UNIOESTE – Universidade Estadual do Oeste do Paraná

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Informática

Curso de Bacharelado em Informática

Desenvolvimento de uma aplicação VoIP para Dispositivos Móveis Dotados com a Tecnologia Bluetooth

Felipe Ricardo Zottis

CASCAVEL

2009

FELIPE RICARDO ZOTTIS

Desenvolvimento de uma Solução VoIP para Dispositivos Móveis Dotados com a Tecnologia Bluetooth

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Informática, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Luiz Antonio Rodrigues

CASCAVEL

2009

FELIPE RICARDO ZOTTIS

Desenvolvimento de uma Solução VoIP para Dispositivos Móveis Dotados com a Tecnologia Bluetooth

Monografia apresentada como requisito parcial para obtenção do Título de *Bacharel em Informática*, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Luiz Antonio Rodrigues (Orientador)
Colegiado de Informática, UNIOESTE

Prof. Marcio Seiji Oyamada (Co-Orientador)
Colegiado de Informática, UNIOESTE

Prof. Anibal Mantovani Diniz
Colegiado de Informática, UNIOESTE

Cascavel, 23 de novembro de 2009.

“Se não puder voar, corra. Se não puder correr, ande. Se não puder andar, rasteje, mas continue em frente de qualquer jeito.”

Martin Luther King

AGRADECIMENTOS

Por esta grande conquista, agradeço primeiramente à minha família, fator mais importante, pois foram eles que me propiciaram todo o apoio necessário nos diversos aspectos possíveis.

Aos amigos de infância da cidade de Santo Antonio do Sudoeste. Foram estes que, além de minha família, me proporcionaram períodos alegres e marcantes quando visitei minha cidade natal. Dos momentos que passei com eles, nunca esquecerei.

Agradeço fortemente aos meus diversos amigos e colegas que conquistei em Cascavel que por muitos momentos me proporcionaram momentos de alegria e descontração. Com eles, foi possível “descansar” e motivar minha mente em horas difíceis como também esquecer a distância dos familiares.

Por fim deixo um agradecimento aos meus dois amigos e orientadores Luiz Antonio Rodrigues e Marcio Seiji Oyamada que sempre estiveram dispostos a me auxiliar nesta empreitada no que fosse necessário.

Lista de Figuras

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

AMS	*Application Management Software*
API	*Application Programming Interface*
CDC	*Connected Device Configuration*
CLDC	*Connected Limited Device Configuration*
CORBA	*Common Object Request Broker Architecture*
CPU	*Central Processing Unit*
EJB	*Enterprise JavaBeans*
FHSS	*Frequency Hopping Spread Spectrum*
FP	*Foundation Profile*
GCF	*Generic Connection Framework*
GHZ	*Gigahertz*
GPS	*Global Positioning System*
GUI	*Graphical User Interface*
HCI	*Host Controller Interface*
HTTP	*Hypertext Transfer Protocol*
IMP	*Information Module Profile*
IP	*Internet Protocol*
ISM	*Industrial, Scientific and Medical*
J2EE	*Java 2 Enterprise Edition*
JME	*Java Micro Edition*
J2SE	*Java 2 Standard Edition*
JCP	*Java Community Process*
JDK	*Java Development Kit*
JIT	*Just-in-time*
JSR	*Java Specifications Request*
JVM	*Java Virtual Machine*
KB	*Kilobyte*
KVM	*Kilobyte* Virtual *Machine*
L2CAP	*Logical Link and Adaption Protocol*
MB	*Megabytes*
MIDP	*Mobile Information Device Profile*
MMAPI	*Mobile Media API*
OBEX	*Object Exchange Protocol*
PABX	*Private Automatic Branch Exchange*
PBP	*Personal Basis Profile*
PDA	*Personal Digital Assistant*
PP	*Personal Profile*
RAM	*Random Access Memory*
RFCOMM	*Radio Frequency Communication*

Lista de Abreviaturas e Siglas

RMI	*Remote Method Invocation*
RTT	*Round Time Trip*
SIG	*Special Interest Group*
SDP	*Service Discovery Protocol*
UUID	*Universally Unique Identifier*
VoIP	*Voice over Internet Protocol*
XML	*Extensible Markup Language*
WAV	*WAVEform audio format*

Sumário

Resumo

Devido à evolução tecnológica constante e crescente, muitas aplicações inovadoras podem ser desenvolvidas periodicamente permitindo que sejam adaptadas ao estilo de vida de cada indivíduo ou organização, agilizando assim suas tarefas cotidianas. Evidencia-se então uma busca por flexibilidade e mobilidade onde naturalmente os assuntos ligados à agilidade em comunicações são essenciais para o Homem, mais precisamente na área da telefonia. A tecnologia de transmissão de voz sob IP (*VoIP – Voice Over Internet Protocol*) busca proporcionar aos usuários da telefonia uma comunicação eficiente e barata. A telefonia via IP é uma tecnologia emergente que resulta da combinação da rede de telecomunicações e da infra-estrutura da Internet. A utilização de transmissão de voz sobre IP constitui de uma técnica para digitalização do áudio, neste caso voz, para que seja transmitido via rede de computadores. Sendo assim, é de interesse deste projeto a construção de um *Softphone* embarcado para dispositivos móveis, capaz de transmitir áudio, em tempo real, via *Bluetooth*, para um servidor responsável por realizar chamadas de voz em uma rede VoIP. Por se tratar de uma tarefa complexa, este trabalho tem como por objetivo avaliar a capacidade de programação, processamento e transmissão de dados dos dispositivos celulares mais comuns no mercado para verificar a viabilidade de integração com uma rede VoIP tradicional.

Palavras-chave: Dispositivos Móveis, JME, *Bluetooth*, VoIP.

Capítulo 1

Considerações Iniciais

1.1 Introdução

A maioria das empresas existentes no mercado necessita prover serviços de telefonia aos seus funcionários para que possam atender seus clientes ou trocar informações importantes com outras organizações de maneira ágil. Para suprir esta demanda, a empresa necessita instalar diversos pontos telefônicos, em muitos casos, um para cada funcionário. Este fato, agregado aos custos de uma chamada telefônica, causa elevadas despesas a uma organização.

Por esta razão, as grandes empresas no mundo todo começam a substituir seus sistemas telefônicos tradicionais PABXs (*Private Automatic Branch Exchange)* por soluções VoIP (*Voice over Internet Protocol*). Esta tecnologia de transmissão de voz sobre IP (*Internet Protocol*) busca proporcionar aos usuários da telefonia uma comunicação eficiente e barata, pois suas chamadas telefônicas possuem um baixo custo por tempo de ligação.

Tendo em vista o cenário descrito, o foco do presente trabalho é desenvolver uma aplicação para permitir que celulares comuniquem-se com a rede de telefonia VoIP. Como a maioria destes aparelhos existentes no mercado possui a tecnologia de transmissão de dados via *Bluetooth*, é possível que estes acessem a rede VoIP através de um ponto de acesso sem fio à ela, garantindo assim certo grau de mobilidade.

A comunicação com a rede telefônica VoIP pode ser provida pelo *Asterisk*, uma ferramenta de código aberto que não requer pagamento de licença para sua utilização. O *Asterisk* pode ser instalado em uma máquina de caráter servidor, a qual através de um dispositivo *Bluetooth* devidamente instalado poderia receber requisições de chamadas dos dispositivos celulares encontrados na área de abrangência do mesmo. Sendo assim, é possível suprir vários dispositivos com apenas uma máquina, restringindo-se apenas as limitações do *Bluetooth*.

Entretanto, antes de desenvolver uma solução que possa ser integrada ao serviço de VoIP é preciso identificar se há suporte por parte dos dispositivos e da tecnologia de transmissão baseada no *Bluetooth*. Nesse sentido, este trabalho procurou identificar as tecnologias, suporte da linguagem de programação, capacidade de processamento e taxa de transmissão oferecidas pelos dispositivos atuais e necessárias para uma comunicação por voz sobre IP.

1.2 Justificativa

Construir uma ferramenta que possa agregar o acesso à rede VoIP em dispositivos celulares é conveniente e importante à medida que empresas do mundo todo buscam soluções baratas para seus meios de comunicação. Juntamente a este fato, pode-se aludir também que a maioria dos funcionários de uma organização possuem um celular, tornando desnecessária a aquisição de aparelhos telefônicos. Somente no Brasil, segundo a Anatel (Agência Nacional de Telecomunicações), existem 154 milhões de dispositivos celulares em uso o que corresponde a pouco mais de 80% da população do país [17].

1.3 Objetivos

1.3.1 Objetivo Geral

Construir uma aplicação na linguagem JME que forneça transmissão e recepção de voz via *Bluetooth* integrando assim dispositivos celulares no processo de comunicação utilizando VoIP.

1.3.2 Objetivos Específicos

- Estudar a plataforma de desenvolvimento para dispositivos móveis JME.
- Estudar a tecnologia de comunicação sem fio *Bluetooth*.
- Implementar a aplicação para o dispositivo móvel.
- Realizar testes para verificar a capacidade de processamento e de transmissão dos dispositivos.

1.4 Trabalhos Relacionados

No trabalho de Souza [23] é realizado um estudo sobre a segurança de uma rede de comunicação VoIP. Para este projeto, o autor fez uso da ferramenta *Asterisk* como estudo de caso para demonstrar quais as falhas de segurança que uma rede VoIP pode apresentar, bem como soluções plausíveis para o problema.

Bazotti [18] aborda o desenvolvimento de uma aplicação para dispositivos móveis que possibilite a transmissão de voz sobre o protocolo IP (*Internet Protocol*). Em seu projeto, o autor faz uso de dispositivos dotados da tecnologia *Wi-Fi* para conexão com a rede *wireless.*

1.5 Organização do Texto

Dada a introdução, o texto segue a seguinte estrutura:

No Capítulo 2 é detalhado como se dá a programação Java para dispositivos móveis: as APIs, máquina virtual, arquitetura, etc.

O capítulo 3 trata da tecnologia *Bluetooth*, suas características, capacidades, modos de rede, protocolos e APIs Java responsáveis pela manipulação das funcionalidades *Bluetooth* de um dispositivo.

O Capítulo 4 define a arquitetura da solução proposta e implementada, como também os resultados obtidos.

O Capítulo 5 trata dos testes realizados com a aplicação desenvolvida, fazendo também uma análise crítica.

Por fim, no capítulo final, uma conclusão sobre o projeto é realizada.

Capítulo 2

Programação Java para Dispositivos Móveis

A linguagem de programação Java foi desenvolvida pela *Sun Microsystems* para proporcionar o desenvolvimento de sistemas independentes do conceito de hardware. Esta característica é obtida através do uso de uma Máquina Virtual Java (JVM – *Java Virtual Machine*), a qual recebe de um compilador da linguagem um conjunto de *bytecodes* e os interpreta a fim de proceder com a execução de um programa. Uma JVM é considerada um interpretador da linguagem [1].

A portabilidade é uma das principais características da linguagem, isto é, a capacidade de executar um programa desenvolvido na linguagem Java em diferentes máquinas com diferentes sistemas operacionais sem a necessidade de recompilação. Esta portabilidade é obtida a partir do momento em que a JVM assume que os *bytecodes*, os tipos de dados e a codificação utilizada pela linguagem Java são independentes do sistema operacional. Sendo assim, a JVM possui uma funcionalidade fundamental de definir como se dá a codificação das classes da linguagem, ou seja, como os *bytecodes* são organizados para formar uma classe [1] [2].

A primeira versão da linguagem (1.0.2) possuía um kit de desenvolvimento (JDK – *Java Development Kit*) adequado para aplicações que requeriam interface gráfica, tais como janelas ou *frames*, bem como *applets* para que um navegador Web execute uma aplicação utilizando a JVM de maneira segura. A segunda versão (1.1) fora lançada para suprir algumas deficiências anteriormente encontradas na linguagem, tais como controle limitado de interface de usuário e um número restrito de modelos de segurança. Nesta versão um novo método de chamada de execução de processos fora adicionado, o *Remote Method Invocation* (RMI) bem como o modo de compilação *Just-in-time* (JIT). A *Sun Microsystems*

decidiu realizar mudanças significativas na seguinte versão da linguagem, Java 2, decidindo por dividi-la em três categorias, cada uma centralizada em um segmento específico de mercado [1] [2]:

- J2SE (*Java 2 Standard Edition*): utilizada para desenvolvimento de aplicações Java ou também *applets* [2]. Seu uso é voltado para o ambiente de aplicações em máquinas *desktop* convencionais, onde há memória, um poder de processamento e armazenamento relativamente adequado para tal.
- J2EE (*Java 2 Enterprise Edition*): utilizada para o desenvolvimento de aplicações com ênfase no desenvolvimento de servidores utilizando-se *Enterprise JavaBeans* (EJBs), bem como aplicações *Web*, CORBA e XML (*Extensible Markup Language*) [2].
- JME (*Java Micro Edition*): utilizada para o desenvolvimento de aplicações para dispositivos com menor disponibilidade de memória e processamento, como celulares e *palmtops* [2].

A linguagem Java dividida em três categorias proporciona inúmeras facilidades aos programadores, como APIs (*Application Programming Interface*) prontas para a construção de determinadas aplicações. Porém, o amplo JDK existente na J2SE entra em contraste com um conjunto muito menor existente na JME, o que pode ocasionar certas dificuldades a um programador que venha a migrar para esta, como, a gerência de memória do dispositivo [1]. Uma vez que o presente trabalho fará uso de dispositivos que se enquadram na especificação JME, esta será detalhada na próxima seção.

2.1 JME

A plataforma JME da *Sun Microsystems* tem como principal foco o desenvolvimento de aplicações para dispositivos dotados de memória e poder de processamento relativamente menor do que as máquinas *desktops* comumente utilizadas por pessoas ou empresas. Os dispositivos celulares possuem capacidades computacionais diferentes, o que depende do modelo ou do fabricante. Assim, torna-se difícil desenvolver aplicações que sejam comumente executadas entre diferentes aparelhos. Ao invés de constituir de uma simples entidade, a JME forma uma coleção de especificações que definem um conjunto de

plataformas, sendo que cada qual possui características que combinam com um determinado dispositivo utilizado pelo usuário [3].

2.1.1 Organização

Historicamente o desenvolvimento de softwares fora voltado para máquinas dotadas de *display*, *mouse*, teclado, quantidades suficientes de memória, bem como armazenamento não volátil de grandes volumes de informações. Com a evolução constante da tecnologia de comunicação e computação, dispositivos menores são lançados gradualmente no mercado. Tais aparelhos, na maioria dos casos, possuem armazenamento não-volátil inadequado ou incapacitado, memória física limitada e são incapazes de realizar grandes processamentos. Estas características são indesejáveis para o desenvolvimento de softwares para estes dispositivos, ocasionando um desafio às grandes empresas ou comunidades responsáveis pela manutenção ou abordagem de novas linguagens ou técnicas de programação para suprir tal demanda [2] [3].

A JME deve ser capaz de suprir computacionalmente diferentes tipos de aparelhos, possuindo estes características diferentes de hardware. O principal desafio da Comunidade Java (JCP - *Java Community Process*) é obter um padrão Java de linguagem e programação para dispositivos dotados de características diferentes e sem padrão de hardware. A JCP é constituída de líderes de grandes empresas os quais se envolvem nas definições de versões futuras da linguagem Java, bem como novas funcionalidades e padrões [2] [3].

A JCP definiu a plataforma de execução da Java e um conjunto de classes que podem operar em cada dispositivo. Dá-se o nome de *configuração* a este processo, onde também é definida a JVM para cada dispositivo. Conseqüentemente a JCP criou o *perfil*, onde um conjunto de classes, adicional ao pré-estabelecido pela configuração, é definido para o desenvolvimento de aplicações para certo grupo de dispositivos. As três camadas aplicadas sobre sob o sistema operacional do aparelho são demonstradas na Figura 2.1 [4]:

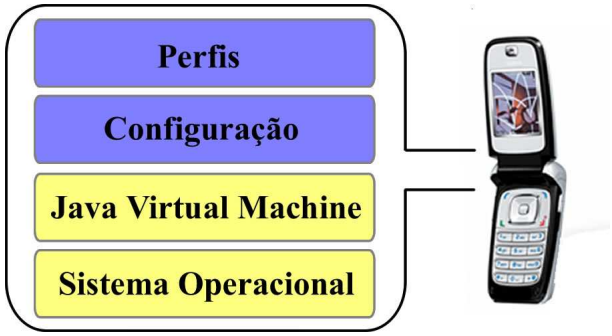

Figura 2.1: Pilha de Camadas sob o Sistema Operacional

Em JME a JVM é estreitamente ligada à configuração do dispositivo, sendo estes desenvolvidos de maneira unívoca para que possam disponibilizar as capacidades essenciais de cada aparelho. Algumas diferenças entre famílias de dispositivos são verificadas pelo acréscimo ou decréscimo de APIs especificas para cada perfil [4].

2.2 KVM

Para que aplicações desenvolvidas na linguagem Java possam ser executadas em diferentes computadores, uma máquina virtual (JVM) é necessária para realizar a interpretação do código da aplicação. A KVM (*Kilobyte Virtual Machine*) fora desenvolvida com este mesmo propósito, porém para dispositivos com poder de processamento ou capacidades de armazenamento, volátil ou não, menores que os dos computadores tradicionais. Uma KVM possui características voltadas para operarem em dispositivos que não dispõem de muitos recursos computacionais, podendo esta carregar classes de uma estrutura de diretórios em um sistema de arquivos local [3].

A KVM realiza a interface entre o sistema operacional nativo do dispositivo e suas aplicações. Sendo assim, é a responsável pelo ciclo de vida destas realizando ações de inicialização, gerenciamento de execução e finalização das mesmas [2].

2.3 Configuração

Uma configuração é constituída de uma especificação com todas as características físicas do dispositivo a qual o software desenvolvido será implantado. Uma KVM é também inclusa em cada configuração, bem como um conjunto de classes Java para fornecer um

ambiente de desenvolvimento de software para o dispositivo [3]. Uma especificação deve possuir requisitos básicos como [3]:

- O tipo e a quantidade de memória disponível;
- A velocidade e o tipo do processador;
- O tipo de conexão de rede existente no aparelho;

Esta especificação normalmente é provida pela empresa fornecedora do aparelho e é fundamental pare que o programador desenvolva corretamente uma aplicação. A JME define duas configurações, a CLDC (*Connected Limited Device Configuration)* e a CDC (*Connected Device Configuration)* [2] [3].

2.3.1 CLDC (Connected Limited Device Configuration)

Projetada para dispositivos móveis pessoais, a CLDC não requer muitos recursos computacionais, por serem limitados, como o próprio nome diz. Nesta categoria enquadram-se celulares e *pagers*. Devido a isto e como o presente trabalho focaliza aparelhos com as características da configuração CLDC, somente os requisitos e APIs desta configuração serão detalhados.

Geralmente nestes aparelhos as seguintes características devem ser encontradas como requisitos mínimos [8] [9]:

- Capacidade mínima de 160 KB de memória não-volátil para a máquina virtual (KVM) e bibliotecas CLDC.
- Capacidade mínima de 32 KB de memória RAM para funcionalidades da máquina virtual, como por exemplo, Java *heap*.

A configuração CLDC assume que o dispositivo possua ao menos um sistema operacional ou *kernel* disponível para a interface entre as aplicações e o *hardware* do aparelho [9].

Conforme [6] os pacotes da Tabela 2.1 estão disponíveis para a configuração do CLDC:

Tabela 2.1: Pacotes incluídos no CLDC [6]

Pacote	Fornece
java.io	Classes para manipulação de entrada e saída de dados

	via *streams*
java.lang	Classes fundamentais para programação Java
java.lang.ref	Suporte para referências fracas
java.util	Classes para coleção de dados
javax.microedition.io	Classes para o GCF (*Generic Connection Framework*)

O pacote java.lang, dentre os mencionados na Tabela 2.1, possui a classe *Thread* para o gerenciamento de processos realizados de maneira concorrente. Como a aplicação *a* ser desenvolvida no presente trabalho exige que dados de áudio sejam enviados e recebidos ao mesmo tempo (características de um telefone) a classe *Thread* será utilizada para o gerenciamento desta funcionalidade.

2.3.2 CDC (Connected Device Configuration)

Diferentemente do CLDC, para estes dispositivos, os requisitos como o tamanho da memória é maior. Por isso, devem possuir 2 MB ou mais de memória física, além de processadores mais robustos e com maior poder de processamento, geralmente na faixa de 32 bits [2] [3]. Nesta categoria enquadram-se aparelhos GPSs e computadores de bordo para automóveis. Para estes dispositivos considera-se que exista uma conexão de rede com uma grande largura de banda [8].

2.3.3 Threads

A plataforma de desenvolvimento JME possui um tipo de *Thread* para cada uma das suas duas configurações. Na CLDC a *Thread* inclui somente um sub-conjunto de métodos encontrados na classe *Thread* da J2SE, sendo eles [3] [21] [22]:

- `activeCount()`: Retorna o número de *Threads* ativas ao mesmo instante.
- `currentThread()`: Retorna o contexto da *Thread em execução* corrente.
- `getPriority()`: Retorna a prioridade atual da *Thread em execução.*
- `isAlive()`: Retorna o *status* de execução da *Thread* corrente.
- `join()`: A *Thread* que invoca este método espera por outra *sincronizar.*

- `run()`: Utilizado quando a *Thread* em contexto foi construída fazendo uso de *Runnable.*
- `setPriority()`: Determina a prioridade da *Thread.*
- `sleep()`: Paraliza a execução por um tempo (em milissegundos) determinado como parâmetro do método.
- `start()`: Inicializa a execução da *Thread* em contexto. Ação realizada pela KVM que invoca o método *Run* da *Thread.*
- `yield()`: Faz com que a execução da *Thread* em contexto seja paralisada para que outras obtenham a CPU do dispositivo.

Diferentemente da encontrada na configuração CLDC, a *Thread* da CDC é mais robusta, pois inclui todas as funcionalidades fornecidas na sua respectiva classe na J2SE [21].

2.4 Perfil

Um perfil é definido como uma extensão de APIs para uma configuração base de um aparelho, ou seja, especifica um conjunto de bibliotecas requeridas para desenvolvimento de aplicações para uma família de dispositivos [1] [8].

Cada dispositivo é qualificado em um determinado perfil por sua empresa fornecedora, atendendo assim as expectativas de mercado sobre o produto. Os perfis em cada grupo de dispositivos são definidos comumente por representantes da indústria juntamente com a JCP [2]. Em termos gerais, para cada segmento de perfil um conjunto de APIs básicas é especificado para que assim uma mesma aplicação possa ser executada em aparelhos com configurações diferentes (porém com o mesmo perfil), garantindo então, juntamente com a KVM a portabilidade em dispositivos móveis. Existem vários tipos de perfis já definidos pela JCP [2]:

- O *Mobile Information Device Profile* (MIDP) cujas características são voltadas para dispositivos qualificados como CLDC. O MIDP é munido das principais APIs necessárias para o desenvolvimento das aplicações mais populares encontradas atualmente [5]. Devido a esta característica, este perfil se enquadra nas requisições do presente trabalho e, portanto, será detalhado posteriormente.

- O *Information Module Profile* (IMP) também possui as características para o desenvolvimento de aplicações em dispositivos CLDC. É denominado como um subconjunto do perfil MIDP [5].
- O *Foundation Profile* (FP) é caracterizado para o desenvolvimento de aplicações em dispositivos CDC. Este perfil pode ser utilizado para a construção de outros perfis, sendo também muito utilizado quando necessário à construção de aplicações que não requerem o uso de interface gráfica, bem como conectividade de rede estável e confiável [5] [8].
- O *Personal Profile* (PP) também é caracterizado por ser baseado em dispositivos característicos CDC. Porém possui APIs voltadas para dispositivos que requerem interfaces gráficas derivadas da Java SE. Este perfil possui um conjunto de classes úteis para a interface gráfica com o usuário [5] [8].
- O *Personal Basis Profile* (PBP) é similar ao Personal Profile, porém com a adição de classes para implementação de interfaces simples para usuário, podendo exibir somente uma janela de visualização por vez. Também é voltado para aparelhos CDC [5] [8].

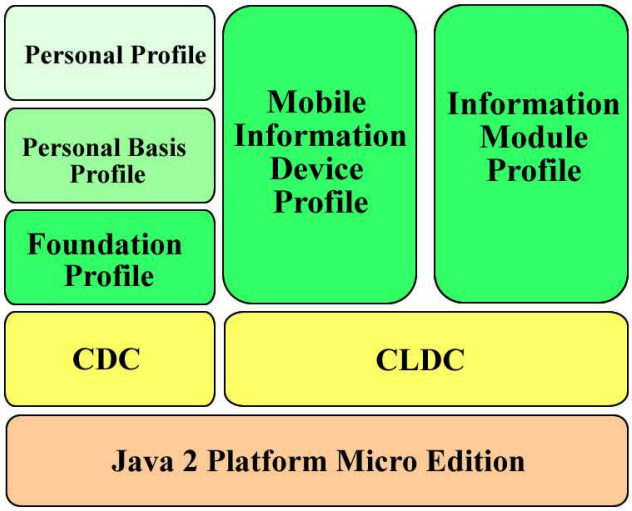

Figura 2.2: Perfis JME [5]

2.5 MIDP

O perfil MIDP 1.0 (*Mobile Information Device Profile*) constitui um conjunto de bibliotecas e APIs para o desenvolvimento de aplicações que utilizam os recursos encontrados em uma plataforma móvel CLDC, como o uso de interfaces gráficas ou conexões com o protocolo HTTP (*Hypertext Transfer Protocol*) [3] [5] [8].

Uma aplicação MIDP é denominada um *MIDlet* e pode utilizar as APIs definidas pelo perfil ou as do conjunto CLDC do dispositivo [5] [6]. O MIDP define um modelo de aplicação para que os recursos limitados de um dispositivo móvel sejam compartilhados entre várias *MIDlets*, podendo estas interagirem entre si. Ele define o que vem a ser um *MIDlet*, como este é organizado, o ambiente para sua execução e também como deverá uma aplicação se portar para que o dispositivo possa gerenciar seus recursos [7].

Atualmente o perfil MIDP encontra-se na versão 2.1, e forma uma extensão do conjunto de APIs fornecidos pela configuração CLDC [7]. Seu conjunto de APIs possui suporte para o desenvolvimento de aplicações que necessitam [8]:

- Interfaces gráficas (GUI);
- Download e execução de maneira segura dos *MIDlets*;
- Armazenamento permanente de dados no dispositivo.

2.5.1 Requisitos de Hardware

Os requisitos listados nesta sessão formam um adicional ao especificado pelo CLDC (*JSR-30)* conforme a *Sun Microsystems*. Para suportar o perfil MIDP 2.1, o dispositivo deve suprir os seguintes requisitos mínimos listados na Tabela 2.2.

Tabela 2.2: Requisitos do perfil MIDP 2.1 [3] [6] [20]

Display	96 pixels verticais e 54 horizontais
Profundidade do *display*	1 bit
Forma do *pixel*	Aproximadamente 1:1
Entrada de dados	Teclado ou *Display touch screen*
Memória	256 KB de memória não-volátil para componentes

	MIDP 8 KB de memória não-volátil para dados persistentes de aplicações 128 KB de memória volátil para ambiente Java
Conectividade	*Wireless* para envio e recebimento com largura de banda limitada
Som	Capacidade de reproduzir sons, seja por meio de software ou por *hardware* dedicado

Os requisitos para a versão 2.1 do perfil MIDP modificam-se da primeira versão somente nas características de memória e de som, pois aplicações mais complexas podem ser desenvolvidas nesta versão. Além da memória requerida pela configuração CLDC, deve existir ao menos 254 KB livres de memória não-volátil e 128 KB de memória RAM para o ambiente Java [6].

2.5.2 Pacotes Incluídos

A Tabela 2.3 lista os pacotes disponíveis no perfil MIDP 2.1 [6].

Tabela 2.3: Pacotes disponíveis no perfil MIDP 2.1

Pacote	Fornece
`javax.microedition.lcdui`	Fornece classes para interfaces de usuário
`javax.microedition.midlet`	Define aplicações MIDP e interações entre elas, bem como o ambiente de sua execução
`javax.microedition.rms`	Fornece armazenamento persistente
`javax.microedition.lcdui.game`	Fornece funcionalidades para aplicações características de jogos
`javax.microedition.media`	Fornece classes para aplicações que envolvem o uso de áudio. Provê ao

	desenvolvedor a capacidade de adicionar *tones* e arquivos em formato *WAV*[1]
`javax.microedition.pki`	Fornece tratamentos de certificados

2.6 MIDlets

Os *MIDlet*s são aplicações desenvolvidas com base nas APIs especificadas pelo perfil MIDP. Para um programador experiente da linguagem Java, o desenvolvimento de *MIDlet*s não constitui um desafio, pois a base do desenvolvimento destes, a linguagem JME, é oriunda do Java. Um *MIDlet* possui seu ciclo de vida inteiramente controlado pelo *Application Management Software* (AMS), sendo este implementado e controlado pela KVM do dispositivo [2].

Não existe restrições quanto ao número de *MIDlet*s que podem ser gerenciadas pelo AMS ao mesmo tempo. Este por sua vez, pode liberar recursos do dispositivo finalizando *MIDlet*s inativas ou paralisando-as. O AMS também é o responsável por determinar qual aplicação será visualizada pelo usuário e quais não serão, pois somente uma pode ser exibida por vez [15].

A programação de um *MIDlet* é similar à criação de aplicações J2SE. Porém o conjunto de APIs é bastante reduzido. Como fora explicitado anteriormente, o desenvolvimento de um *MIDlet* é voltado para dispositivos menores, que possuem capacidade limitada de processamento. Assim, algumas APIs do J2SE não podem ser executadas [2] [3] [6].

Um *MIDlet* deve possuir ao menos uma classe que seja derivada da `javax.microedition.midlet.MIDlet`, pois esta possui métodos que implementam ações necessárias para a KVM gerenciá-la. Os métodos pré-definidos na classe *MIDlet* são [2] [15]:

- `startApp()`: chamado pelo AMS para inicializar ou reiniciar um *MIDlet*. Após a chamada deste método, o *MIDlet* entra no estado ativo e com isso o AMS lhe atribui recursos para sua execução.

[1] *WAVEform audio format*. Formato padrão de arquivos de áudio da Microsoft e IBM para armazenamento de áudio.

- `pauseApp()`: invocado quando a execução do *MIDlet* necessita ser paralisada temporariamente. Um exemplo da invocação deste método é quando um jogo em atividade pelo usuário necessita ser interrompido para que uma chamada telefônica do dispositivo seja atendida.
- `destroyApp()`: este método coloca o *MIDlet* em estado destruído sinalizando-o para que seja finalizado. Durante este processo, o *MIDlet* deve liberar os recursos utilizados e armazenar seus dados persistentes.

Os três métodos apresentados acima não possuem valores de retorno. Somente o método `destroyApp()` necessita um valor boleano como argumento para sua invocação, o qual é designado como verdadeiro se a finalização do *MIDlet* é incondicional, ou falso se o *MIDlet* pode emitir uma exceção à KVM, informando que não é o momento certo de ser finalizado [2]. A Figura 2.3 exibe os possíveis estados de um *MIDlet*.

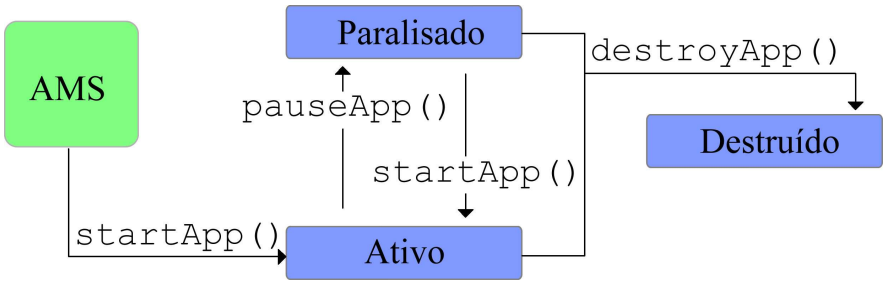

Figura 2.3: Ciclo de vida de um *MIDlet* [7]

Ao contrário das aplicações desenvolvidas em J2SE, um *MIDlet* não deve possuir o método `public static void main()`, pois o AMS fornece um método padrão de inicialização necessárias à CLDC do dispositivo para inicializar o *MIDlet* [2].

2.6.1 MIDlet Exemplo

A Figura 2.4 exibe um *MIDlet* exemplo que constitui um primeiro contato com a tecnologia JME, onde somente a mensagem "*Hello World*" é exibida ao usuário.

```
1  package hello;
2
3  import javax.microedition.midlet.*;
4  import javax.microedition.lcdui.*;
5
6  public class HelloWorld extends MIDlet implements CommandListener {
7
8      private Display display;
9      private TextBox textBox;
10     private Command quitCommand;
11
12     public void startApp() {
13         display = Display.getDisplay(this);
14         quitCommand = new Command("Quit", Command.SCREEN, 1);
15         textBox = new TextBox("Hello World", "My first MIDlet", 40, 0);
16         textBox.addCommand(quitCommand);
17         textBox.setCommandListener(this);
18         display.setCurrent(textBox);
19     }
20
21     public void pauseApp() {
22     }
23
24     public void destroyApp(boolean unconditional) {
25     }
26
27     public void commandAction(Command choice, Displayable displayable) {
28         if (choice == quitCommand) {
29             destroyApp(false);
30             notifyDestroyed();
31         }
32     }
33 }
```

Figura 2.4: *MIDlet* "*Hello World*"

Quando o AMS do dispositivo inicializa a aplicação "*Hello World*", o método `startApp()` é o primeiro a ser invocado, e com isto, uma instância do objeto *Display* é criada. Através do comando `getDisplay()` o AMS permite que a aplicação seja exibida ao usuário [2].

O método `commandAction()` contém parâmetros para validar eventos que ocorrem durante a execução do *MIDlet*. O comando requisitado pelo usuário é passado como primeiro parâmetro, *choice*. O segundo parâmetro constitui de um objeto *Displayable*, o qual foi denominado como uma referência ao objeto *TextBox*, através do comando `setCurrent(textBox)`. Neste exemplo, somente a ação de sair da aplicação, denominada *`quitCommand`*, é aceita na chamado de `commandAction` [2].

O *MIDlet HelloWorld* apresentado na Figura 2.5 apresenta ao usuário o resultado ilustrado na Figura 2.5.

Figura 2.5: Resultado exibido ao usuário

2.7 MMAPI

Os dispositivos da geração atual de aparelhos celulares possuem o recurso de realizar *streamming* multimídia. Sendo assim possuem a capacidade de reprodução de arquivos de áudio e vídeo armazenados no próprio aparelho como também da internet. A API *Mobile Media* (MMAPI) desenvolvida pela *Sun Microsystems* possui a finalidade de gerenciar a operação de *streamming* multimídia de um dispositivo menor [16]. Como o presente estudo requer a gravação, reprodução e posteriormente a transmissão de áudio por parte de um dispositivo celular, a MMAPI será aqui detalhada.

A MMAPI constitui um pacote opcional da plataforma JME e é suportada por dispositivos que implementam quaisquer perfis baseados nas configuração CLDC e CDC. Esta API possui a finalidade básica de permitir que uma *MIDlet* acesse os recursos de áudio

e vídeo de um determinado dispositivo. Com isso, aplicações de reprodução ou gravação multimídia podem ser desenvolvidas [16].

A relação de dispositivos das principais marcas do mercado com suporte a MMAPI está listada na Tabela 2.4 [16].

Tabela 2.4: Dispositivos que suportam MMAPI

Fornecedor	**Dispositivo**
Alcatel	One Touch756
BenQ	AX75 (MIDP 1.0), C70, C75, CF75/76, CL75, CX70/EMOTY,CX75, M75, S75, SL75, SXG75
Motorola	C975, E1000, A1000, A630, A780, A845, C380, C650, E398, E680, SLVR, T725, V180, V220, V3, V300, V303, V360, V400, V500, V525, V550, V551, V600, V620, V635, V8, V80, V980, i730
Nokia	Todas as séries 40, séries 60, e séries 80
Samsung	E310, E380, E710, D400, P705, D410, 176X192 Series, E810, E310
Sony-Ericson	W900, Z600, T610, T616, T618, V600, W800, K608, W550, W600, z520, D750, Z800, K600, K750, K300, K500, K700, J300, V800, Z500, S700, Z1010

A MMAPI é suportada por quaisquer dispositivos que implementam o perfil MIDP 2.0, mas pode estar presente também em alguns dispositivos com a versão 1.0 do perfil [16].

2.7.1 Arquitetura

Um *MIDlet* pode receber dados multimídia de várias localizações possíveis inclusive da internet. Desta forma, o dispositivo necessita realizar a decodificação dos dados do

streamming para o formato de áudio ou vídeo necessitado para assim reproduzi-los efetivamente ao seu usuário [16].

Para a obtenção deste objetivo, o MMAPI possui duas classes especificas para realizar este processo: *Player* e *DataSource*. Estas duas classes trabalham de maneira conjunta para permitir que o dispositivo reproduza arquivos multimídia [16] [26].

A classe *DataSource* fornece meios de acesso ao dado multimídia requisitado, localizando e gerenciando uma conexão com este. Durante a criação de um *DataSource*, o caminho do arquivo multimídia deve ser informado como parâmetro de criação [16].

Cada objeto *DataSource* é composto por um ou mais fluxos de entrada e saída de dados, que são conhecidos como *SourceStream*. Estas *streams* constituem a interface de comunicação e transferências de dados com o objeto *Player* utilizado no *streamming* [16] [26].

Um objeto *Player* é o responsável por receber um dado multimídia e decodificá-lo para posteriormente iniciar sua execução para o usuário. O *Player* é o responsável por utilidades básicas de um dado multimídia, como controlar o volume deste [16].

Os três principais métodos invocados pelo *Player* para gerenciar a execução multimídia são [16]:

- `start()`: inicializa a execução do arquivo multimídia assim que possível.
- `stop()`: paralisa uma execução multimídia corrente.
- `close()`: finaliza a execução da multimídia, liberando os recursos de memória utilizados por este.

O único modo de criação de um *Player* é através dos métodos da classe *Manager*. Esta classe é a interface de comunicação entre um objeto *DataSource* e um *Player*, possuindo três maneiras diferentes para que esta última seja criada [16]:

- `createPlayer(DataSource)`: onde uma instância de *DataSource* é informada como parâmetro de criação.
- `createPlayer(String localização)`: através deste método é possível informar diretamente a localização do dado multimídia, ou seja, não necessita a criação de um *DataSource*.

- `createPlayer(InputStream is, String tipo)`: este método cria um *Player* a partir de um *stream* de entrada *InputStream* do pacote de APIs `java.io`. O segundo parâmetro requisitado pelo método refere-se ao tipo do conteúdo a ser reproduzido pelo *Player*, por exemplo, um arquivo *wav*.

O ciclo de vida de um *Player* possui a finalidade de fornecer um controle sobre o tempo consumido para cada operação deste. Os cinco estados definidos para o ciclo de vida de um *Player* são detalhados abaixo [16] [26]:

- *Unrealized*: durante este estado, o *Player* não possui informações suficientes sobre os recursos necessários para executar uma multimídia.
- *Realized*: neste estado o *Player* já possui todas as informações sobre recursos necessários para a sua operação. Pode ser necessário o *Player* realizar uma comunicação com um servidor ou a leitura de um arquivo para obter tal informação.
- *Prefetched*: após o modo *Realized*, o *Player* pode requisitar uma série de tarefas antes de começar uma execução. Ações como preenchimento de *buffers* e obtenção de recursos extras podem ser necessárias antes de o Player inicializar uma execução. Um *Player* retorna a este estado quando tem sua execução paralisada.
- *Started*: o *Player* entra neste estado após a passagem pelo estado *Prefetched*, invocando seu método `start()`. A entrada neste estado define o *Player* como em execução.
- *Closed*: quando o *Player* entra neste estado significa que sua execução foi finalizada. Sendo assim, os recursos utilizados por ele são liberados. A partir deste estado não é possível adentrar nenhum dos demais.

Os possíveis estados de um objeto *Player* podem ser mais bem evidenciados na Figura 2.6:

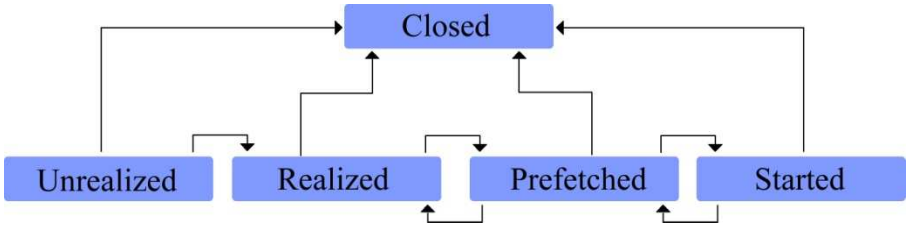

Figura 2.6: Possíveis estados de um *Player*

Para um melhor entendimento de como um *Player* pode ser criado, um exemplo do uso desta classe será apresentado. No exemplo proposto, um *Player* é criado e encarregado de executar um arquivo de som em formato *wav* armazenado no dispositivo. A Figura 2.7 constitui um exemplo simples de como uma classe *Player* pode ser criada a partir da classe *Manager*.

```
1   package Player;
2
3   import javax.microedition.midlet.*;
4   import javax.microedition.lcdui.*;
5   import javax.microedition.media.Manager;
6
7   public class Player extends MIDlet {
8
9       public void startApp() {
10          Player player = (Player) Manager.createPlayer(
11                  getClass().getResourceAsStream("/media/audio/sound1.wav"),
12                  "audio/x-wav");
13          player.start();
14      }
15
16      public void pauseApp() {
17      }
18
19      public void destroyApp(boolean unconditional) {
20      }
21  }
22
```

Figura 2.7: *MIDlet Player*

O arquivo de áudio chamado sound1.wav armazenado em media/áudio/ é reproduzido a partir da criação de uma *InputStream* para o mesmo. A linha 10 merece uma atenção especial nesta demonstração A instância do `Player` deste trecho é criada a partir de uma

string contendo a localização do arquivo de áudio requisitado. Desta mesma maneira, se o arquivo estivesse localizado em um repositório na Internet, o seu endereço de localização seria “HTTP:///<url-para-arquivo>:///<localizacao-arquivo>. O comando `player.start` indica o início da reprodução do arquivo por parte do Player.

Para a aplicação proposto no presente projeto, o *Player* deve realizar a gravação de trechos de áudio para serem enviados a uma ponte de transmissão. A classe *RecordControl* é utilizada para isto, sendo que ela possui métodos para inicializar e parar uma gravação [16] [26]. Para que um *Player* possa realizar uma gravação de áudio, este deve ser instanciado através do método `createPlayer(“capture://audio”)` da classe *Manager*, onde o parâmetro informado em sua criação consiste do formato multimídia a ser utilizado [26]. Os seguintes formatos são suportados [26]:

- `capture://image` (captura de imagem);
- `capture://video` (gravação de vídeo);
- `capture://devcam0` (gravação da câmera de vídeo do dispositivo, se este a possuir);
- `capture://devcam1` (gravação da segunda câmera de vídeo do dispositivo, se este a possuir);

Quando nenhum formato multimídia é informado na criação do *Player*, este assume o tipo padrão especificado pelo dispositivo [16].

Qualquer dado gravado deve ser armazenado para então ser reproduzido pelo dispositivo. A classe *RecordControl* fornece dois métodos para especificar ao *Player* onde armazenar estes dados [16]:

- `setRecordStream(OutputStream stream)`: envia os dados recebidos pelo Player diretamente à uma stream de saída, que pode ser um endereço na rede ou um arquivo local.
- `setRecordLocation(String localização)`: armazena os dados localmente no caminho do dispositivo especificado pela String de parâmetro.

O uso da classe *RecordControl* para a gravação de um segmento de áudio pode ser observado na Figura 2.8:

```
39 Player capturePlayer = Manager.createPlayer("capture://audio");
40 capturePlayer.realize();
41 RecordControl recordControl;
42 recordControl = (RecordControl) capturePlayer.getControl("RecordControl");
43 ByteArrayOutputStream output = new ByteArrayOutputStream(1024);
44 recordControl.setRecordStream(output);
45 recordControl.startRecord();
46 Thread.sleep(10000);
47 recordControl.stopRecord();
48 recordControl.commit();
49 capturePlayer.close();
```

Figura 2.8: Exemplo de uso da classe *RecordControl*

O *Player* responsável pela gravação do áudio é criado e preparado, com *realized*, para poder ser inicializado. Após a criação do objeto *recordControl*, uma variável, do tipo vetor de bytes com tamanho 1024, é criada para armazenar os dados recebidos pelo Player. O método `setRecordStream` de *recordControl* determina a variável *output* para armazenamento. A gravação do áudio é iniciada pela execução do comando `startRecord()`, e em seguida, uma *Thread* determina um *sleep* (*delay*) de 10000 milissegundos (10 segundos) de gravação, vindo a terminar com a execução de `stopRecord()`. O comando `commit()` de *recordControl* efetua a persistência dos dados recebidos.

Capítulo 3

Bluetooth

A tecnologia *Bluetooth* foi desenvolvida a fim de proporcionar comunicação sem fio entre dois ou mais dispositivos capazes de enviar e receber ondas de rádio de curto alcance [11]. Sem a necessidade de cabos para comunicação entre aparelhos, o *Bluetooth* proporciona um certo grau de mobilidade no uso de um dispositivo.

Entre as principais características desejáveis no processo de desenvolvimento da tecnologia *Bluetooth* é possível citar como fundamentais e essenciais o seu custo reduzido e o baixo consumo de energia. Desta forma, evidencia-se uma terceira característica importante e conseqüente das duas anteriores: a possibilidade de implantação em dispositivos com poucos recursos, tais como celulares, *pagers*, *palms* e *notebooks* [23].

O raio de alcance para esta forma de comunicação é uma variável dependente das necessidades requeridas e das características de cada aparelho. Sendo assim, três classes foram padronizadas para representar as diferentes capacidades de alcance, representadas na Tabela 3.1 [10]:

Tabela 3.1: Classes de Alcance da Tecnologia *Bluetooth* [10]

Classes	Alcance (metros)	Potência (*miliwatts*)
Classe 1	100 a 300	100.00 mW
Classe 2	10 a 33	2.5 mW
Classe 3	1 a 3	1.0 mW

Atualmente a grande maioria dos dispositivos utilizados por usuários comuns são dotados de interface *Bluetooth* de Classe 2, podendo estes comunicarem-se com outros aparelhos em um raio de até 10 metros de diâmetro sem interferência física [23].

As especificações da tecnologia *Bluetooth* são propostas pelo *Bluetooth Special Interest Group* (SIG) o qual também é responsável direto pela padronização da tecnologia, permitindo que seja utilizada comumente no mundo todo [11].

3.1 Freqüência de Transmissão

A tecnologia *Bluetooth* utiliza a faixa de transmissão ISM (*Industrial, Scientific and Medical*), que opera com freqüências na faixa de 2.400 GHz até 2.483 GHz. Esta freqüência esta situada entre as ondas de transmissões de televisão e satélites e, por ser uma faixa de freqüência aberta, não necessita de licença para ser usada [11].

A ISM foi criada para que dispositivos que emitem muita interferência de rádio sejam nela agregados, impedindo assim que suas transmissões de dados não causem interferências em outras aplicações sensíveis a ruídos. Sendo uma faixa de freqüência aberta, a ISM pode ser utilizada por quaisquer aplicações comerciais que troquem informações por *wireless*, podendo estas ser desde redes locais sem fio até sistemas de segurança de empresas [12].

O *Bluetooth* utiliza em suas transmissões de dados a técnica de modulação FDMA (*Frequency Hopping Spread Spectrum* – FHSS). Neste método a freqüência de 2.400 a 2.483GHz (utilizada pela ISM) é dividida em 79 canais de 1MHz cada, onde os dados serão enviados ou recebidos [12].

Estes canais pré-selecionados são utilizados pelo transmissor e pelo receptor de maneira síncrona para realizarem uma comunicação. O *Bluetooth* que inicia uma conexão realiza uma alternância de canais, dentre a seqüência estabelecida, para realizar a transmissão de um pacote único, ou seja, cada um destes será transmitido por um canal diferente. A Figura 3.1 mostra esta alternância de freqüências.

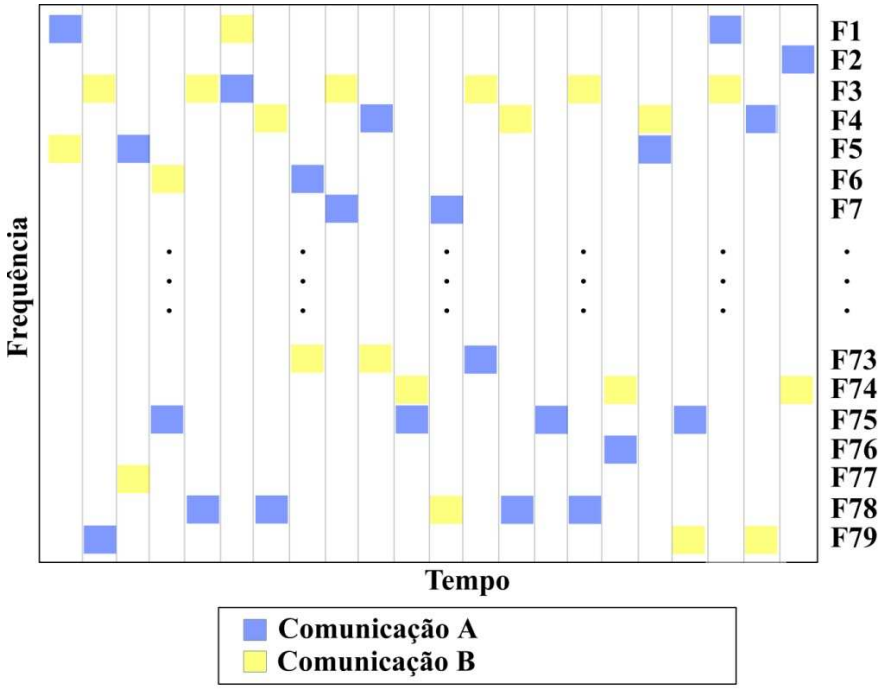

Figura 3.1 – Alternância de freqüências em duas comunicações [23]

Como cada dispositivo pode enviar ou receber dados ao mesmo tempo (*full-duplex*) a transmissão varia entre *slots* de entrada e de saída. Um *slot* é constituído por um canal dividido em períodos de tempo de 625 µs (microssegundos). Cada troca de canal realizada pelo transmissor deve ser ocupada por um *slot*, evidenciando assim 1.600 trocas por segundo. Isto é realizado para evitar que o canal escolhido para a transmissão já esteja em uso por outra aplicação no raio de operação do dispositivo *Bluetooth*. Desta maneira, ruídos ou interferências provenientes de outras aplicações são praticamente evitadas, pois a probabilidade de um canal utilizado para comunicação de um dispositivo com outro já estar em uso é remota. [11] [13].

3.2 Redes Bluetooth

As redes *Bluetooth* são denominadas *piconets*, e são formadas quando dois ou mais dispositivos iniciam uma comunicação através de uma conexão. O dispositivo que inicia a conexão é denominado *master* e possui a responsabilidade de regular a transmissão de dados e o sincronismo dos dispositivos da rede. O aparelho que aceita uma conexão é denominado de *slave* [11] [13].

Em uma *piconet* pode-se ter no máximo 8 dispositivos trocando informações entre si (1 *master* e 7 *slaves*). Estas redes podem conectar-se com outras semelhantes, formando uma *scatternet.* Este esquema de conexão acontece quando um, ou mais, dispositivo de uma *piconet* conecta-se com um aparelho de outra *piconet*. Um *master* só pode ostentar esta função em uma única rede. Os *slaves* podem estar em mais de uma rede ao mesmo tempo [11] [13].

Como o *Bluetooth* utiliza transmissão de dados pela freqüência FDMA, torna-se remota a probabilidade de sobreposição de um canal utilizado. Sendo assim, várias *piconets* podem operar ao mesmo tempo em uma mesma localização [11]. O esquema de uma *scatternet* é demonstrado na Figura 3.2:

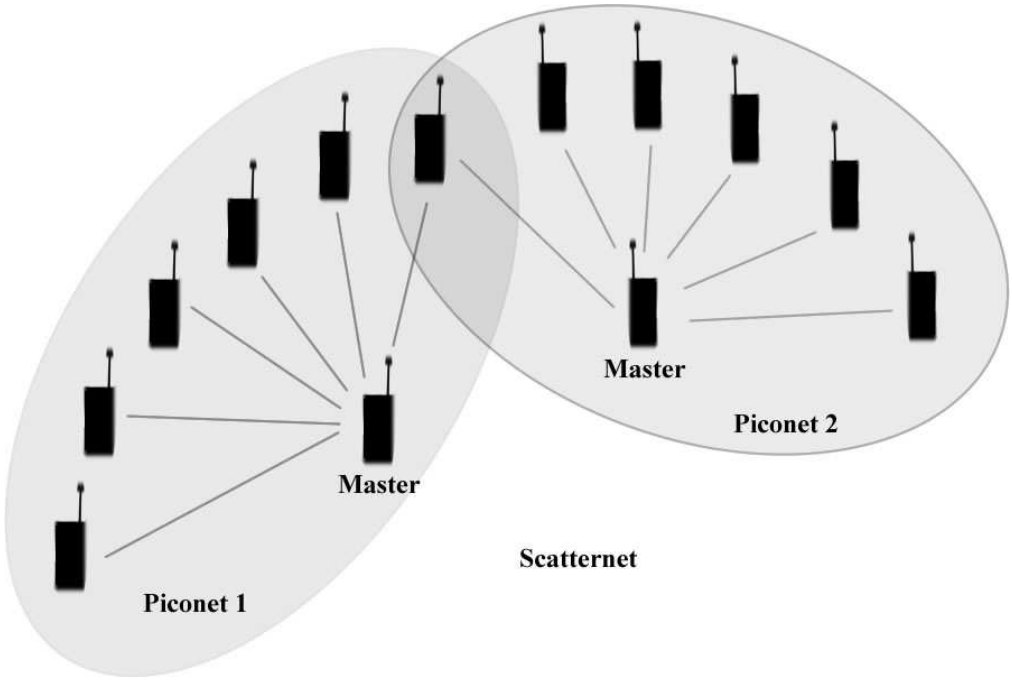

Figura 3.2: Topologia de rede *Bluetooth* [13]

3.3 Pilha de Protocolos

A arquitetura *Bluetooth* implementa uma pilha de protocolos para permitir que dispositivos de diferentes fabricantes possam se comunicar entre si [6]. Com o auxílio da Figura 3.3 é possível analisar os protocolos do *Bluetooth*.

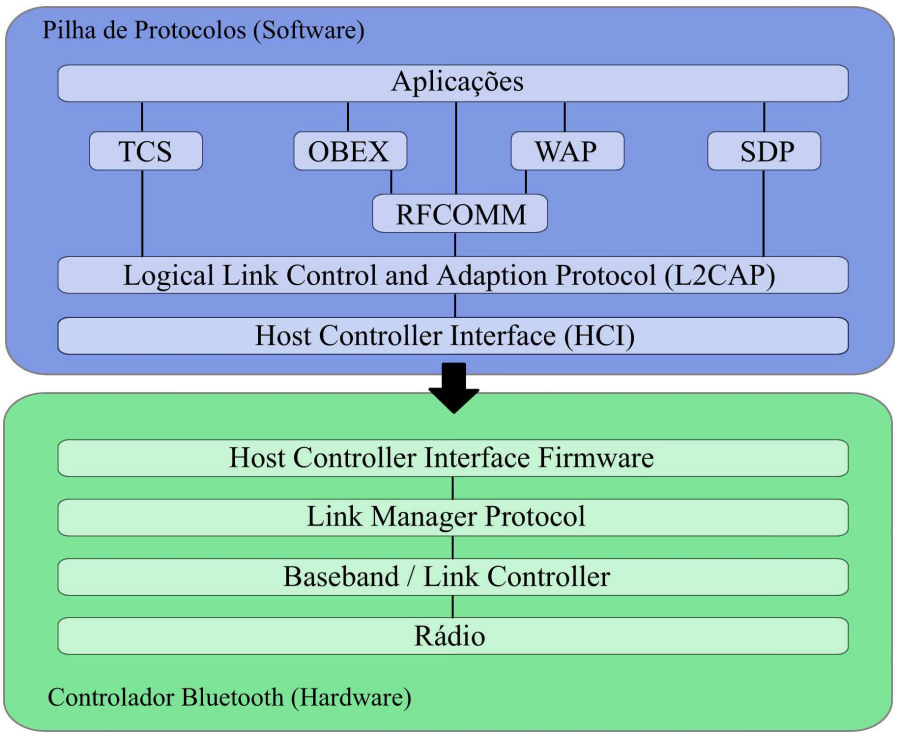

Figura 3.3: Pilha de protocolos *Bluetooth* [6]

A pilha de protocolos é constituída de muitas camadas sendo estas separadas por implementações de componentes em *hardware* e *software*. A comunicação destes é realizada pelo HCI (*Host Controller Interface*). Este componente é responsável pelo acesso ao *hardware* e seus registradores. É possível visualizar sua importância na pilha de protocolos através da Figura 3.3. Todas as camadas abaixo do HCI são implementadas em *hardware* e as acima dele em *software* [6] [24].

A camada do protocolo L2CAP (*Logical Link and Adaption Protocol*) realiza a interface entre as camadas de transporte e as mais altas da pilha de protocolos, possibilitando a transmissão de dados entre estas [6]. O L2CAP permite que os protocolos mais altos da pilha e as aplicações transmitam e recebam pacotes de dados L2CAP de 64 KB [24]. Outros protocolos realizam interface com o L2CAP:

- SDP (*Service Discovery Protocol*): Utilizada pelo dispositivo para realizar a busca por serviços na rede *Bluetooth* [6] [19]. Esta camada será detalhada na seção 3.4, pois a aplicação a ser implementado necessitará fazer uso desta propriedade.

- OBEX (*Object Exchange Protocol*): Protocolo desenvolvido para permitir a transferência de objetos. Utiliza um modelo cliente-servidor e fornece a base para a transferência independentemente dos mecanismos de transporte, ou APIs (*Application Programming Interface*), utilizados. O RFCOMM é o único transporte para a camada OBEX [19] [24].
- RFCOMM (*Serial Cable Emulation Protocol*): Fornece a emulação das portas seriais sobre o protocolo L2CAP. Possui a capacidade de suportar até 60 conexões de canais RFCOMM simultâneas [19].

3.4 SDP (*Service Discovery Protocol*)

O Processo de descobrir quais serviços estão disponíveis para um cliente utilizar é regido através de um protocolo de mensagens denominado SDP (*Service Discovery Protocol* – Protocolo de Descoberta de Serviços). Este protocolo fornece meios para as aplicações de caráter cliente descobrirem a existência de serviços fornecidos por um servidor. O cliente faz uma requisição SDP à uma maquina servidora. Esta mantém uma lista com todos os registros de serviços disponíveis, bem como suas características, atributos, protocolos necessários e mecanismos para utilizar o serviço. No entanto, se o cliente requisitar utilizar um serviço, deve abrir uma conexão separada do servidor, pois o SDP somente contém informações de descobertas de serviços e não como utilizá-los [11] [19].

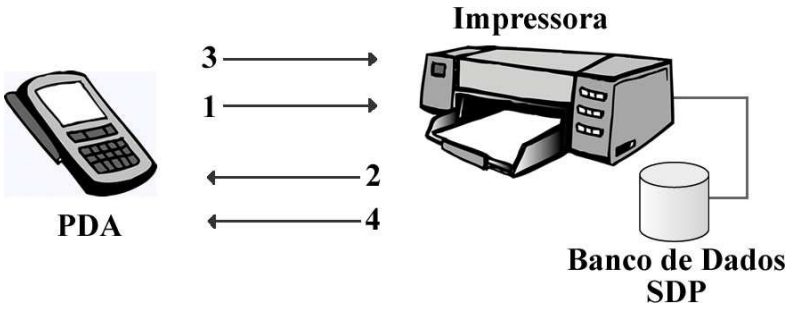

Figura 3.4: Exemplo de requisição de serviço [11]

A Figura 3.3 demonstra como um PDA (*Personal Digital Assistant* – Assistente Digital Pessoal) como *master*, requisitando o serviço de impressão à uma impressora, denominada como unidade *slave*. O processo constitui necessariamente das seguintes etapas [11]:

1. O PDA envia um sinal de *inquiry* para a impressora. Este sinal é constituído de mensagens que um dispositivo, neste caso o PDA, envia pela área de abrangência do seu *Bluetooth*, procurando por dispositivos para se conectar.
2. A impressora responde o PDA com seu respectivo endereço *Bluetooth.* Se houvessem outros dispositivos na área, estes também responderiam ao PDA.
3. O PDA envia então uma requisição de conexão ao endereço retornado pela impressora.
4. A impressora retorna sua lista de serviços, localizada no seu banco de dados SDP, ao PDA. Este por sua vez, armazena os serviços disponíveis em seu banco de dados SDP. Desta forma, o dispositivo pode disponibilizar os serviços da impressora ao seu usuário.

Assim sendo, dada esta estrutura de buscas por serviços pela área de operação do dispositivo *Bluetooth* do celular, é possível utilizar este método no presente trabalho. A disponibilidade do serviço VoIP oferecido ao usuários será melhor detalhada em capítulos posteriores.

3.5 JSR 82: APIs JAVA para Bluetooth

A JSR 82 (*Java Specifications Request*) definida pela JCP contém as APIs *Java* para programação *Bluetooth.* Com esta API é possível construir aplicações que necessitam utilizar a comunicação sem fio do dispositivo [6] [14]. Os seguintes pacotes são encontrados na especificação JSR 82 [6] [25]:

- `javax.bluetooth`: este pacote fornece o conjunto de APIs para funcionalidades essenciais para dispositivos *Bluetooth*, como a descoberta de dispositivos.
- `javax.obex`: neste pacote encontra-se a API do protocolo de comunicação *Bluetooth* OBEX.

Dependendo da configuração do dispositivo, ambos os pacotes podem estar nele contidos por padrão [25].

Sendo assim, o escopo da JSR 82 está em fornecer três funcionalidades essenciais à uma aplicação que requer uso de *Bluetooth*, sendo elas [25]:

- Descoberta de dispositivos: através dos métodos `startInquiry()` e `retrieveDevices()` da classe *DiscoveryAgent* é possível encontrar os dispositivos disponíveis no alcance da rede *Bluetooth*. O método `startInquiry()` realiza uma busca por dispositivos localizados no raio de operação do dispositivo (geralmente 10 metros). O método `retrieveDevices()` não realiza uma busca, mas retorna uma lista de dispositivos previamente encontrados por um método `startInquiry()` realizado.
- Registrar e encontrar serviços: Para registrar um serviço, um servidor deve atribuí-lo a um UUID (*Universally Unique Identifier*), desta forma o processo torna-se disponível para ser localizado e utilizado por um dispositivo na rede *Bluetooth*. A classe *DiscoveryAgent* também possui o método `searchServices()` para localizar um serviço disponível na rede.
- Transferência de dados: é possível realizar a transferência de arquivos utilizando um dos três protocolos: OBEX, L2CAP ou RFCOMM. Um exemplo demonstrativo do uso do protocolo RFCOMM para transferência de dados será exibido na Seção 3.6.

3.6 Comunicação via Bluetooth

Existem diferentes formas e protocolos para realizar a transferência de informações entre dispositivos *Bluetooth*, como o RFCOMM ou o OBEX que são definidos para que aparelhos *Bluetooth* de diferentes fabricantes possam realizar transferência de dados entre si [6].

As Figura 3.5 exibe a implementação de um método para um *MIDlet*, de caráter cliente, que realiza a transferência de um vetor de inteiros para um dispositivo *Bluetooth*, já localizado por um método `startInquiry()`. O protocolo utilizado para transferência de informações no exemplo das Figuras 3.4 e 3.5 é o RFCOMM.

```
45      private void handleConnection(StreamConnection conn) throws IOException {
46          OutputStream out = conn.openOutputStream();
47
48          try {
49              int[] tokens = new int[] { 1, 3, 5, 7, endToken };
50              for (int i = 0; i < tokens.length; i++) {
51                  out.write(tokens[i]);
52                  out.flush();
53                  log("write:" + tokens[i]);
54              }
55          } catch (Exception e) {
56              log(e);
57          } finally {
58              log("Close connection.");
59              if (conn != null) {
60                  try {
61                      conn.close();
62                  } catch (IOException e) {
63                      e.notify();
64                  }
65              }
66          }
67      }
```

Figura 3.5: Exemplo de transmissão via *Bluetooth* (cliente)

O processo de envio de dados dá-se de maneira semelhante a uma transmissão de *sockets* TCP. Um canal de transporte é aberto através da invocação do método `openOutputStream()` da classe *StreamConnection*, que é a responsável pelo fluxo de informações de uma conexão. O envio das informações é realizado com uma chamada ao método `write()`, e o recebimento através do método `read()`. Um exemplo de código de leitura pode ser visto na Figura 3.6.

```
122     private void handleConnection(final String url) {
123         Thread echo = new Thread() {
124             public void run() {
125                 StreamConnection stream = null;
126                 try {
127                     stream = (StreamConnection) Connector.open(url);
128                     InputStream in = stream.openInputStream();
129                     OutputStream out = stream.openOutputStream();
130                     while (true) {
131                         int r = in.read();
132                         log("Read " + r + ", write it back.");
133                         out.write(r);
134                         out.flush();
135                         if (r == stopToken) {
136                             break;
137                         }
138                     }
139                 } catch (IOException e) {
140                     log(e);
141                 } finally {
142                     log("Bluetooth stream closed.");
143                     if (stream != null) {
144                         try {
145                             stream.close();
146                         } catch (IOException e) {
147                             log(e);
148                         }
149                     }
150                 }
151             }
152         };
153         echo.start();
154     }
```

Figura 3.6: Exemplo de transmissão via *Bluetooth* (servidor)

A Figura 3.6 exibe a implementação do método de uma aplicação que executa em um dispositivo *Bluetooth* de caráter servidor. Neste método é possível observar o uso de uma *Thread* a qual permanece aguardando por novas conexões. Assim que uma conexão é realizada a *Thread* executa até que um *token* final lhe é transmitida.

Capítulo 4

Sistema Proposto

Com embasamento nos capítulos anteriores é possível propor um sistema cuja arquitetura de transmissão de dados composta por dois dispositivos celulares, de características diferentes ou não, possam se localizar fora do espaço de abrangência do seu dispositivo *Bluetooth* para que iniciem uma conversação de voz entre si. A arquitetura descrita constitui de três elementos principais:

- O Servidor Central: sistema responsável por armazenar os cadastros dos dispositivos que estão disponíveis, *online,* para iniciarem nova conexão e conseqüentemente uma conversação.
- As Pontes de Conexão e Transmissão: este componente constitui-se de um computador munido de transmissão *Bluetooth* e interligado com o servidor através da rede (local, *intranet* ou *Internet*). A ponte de conexão deve gerenciar, cadastrar e excluir um dispositivo por ela gerenciado e também realizar a transferência dos dados de áudio oriundos dos dispositivos envolvidos na transação. As pontes devem estar interligadas por redes de conexão, por cabeamento ou via *wireless*.
- O dispositivo celular, *Palmtop* ou PDA: para que o aparelho possa ser aplicado ao sistema proposto, é necessário que ele possua os seguintes requisitos mínimos:
    - Dispositivo para conexão *Bluetooth*;
    - Plataforma JAVA;
    - API JSR-82;
    - Perfil MIDP 1.0;

- CLDC 1.0;

Para que este elemento possa realizar e receber conexões para iniciar uma conversação, ele deve possuir duas características diferentes de funcionamento:

- Como cliente: o dispositivo age desta forma a partir do momento em que o usuário requisita uma nova conexão com outro dispositivo cadastrado no servidor.
- Como servidor: o dispositivo fica encarregado de receber uma nova conexão de outro dispositivo.

O esquema da arquitetura do sistema aqui proposto é apresentado na Figura 4.1, onde é possível observar os três elementos que constituem o projeto na totalidade, o servidor, as pontes de conexão e transmissão e os dispositivos utilizados para conversação. Cada um destes componentes possui uma implementação de *software* exclusiva para suas funcionalidades.

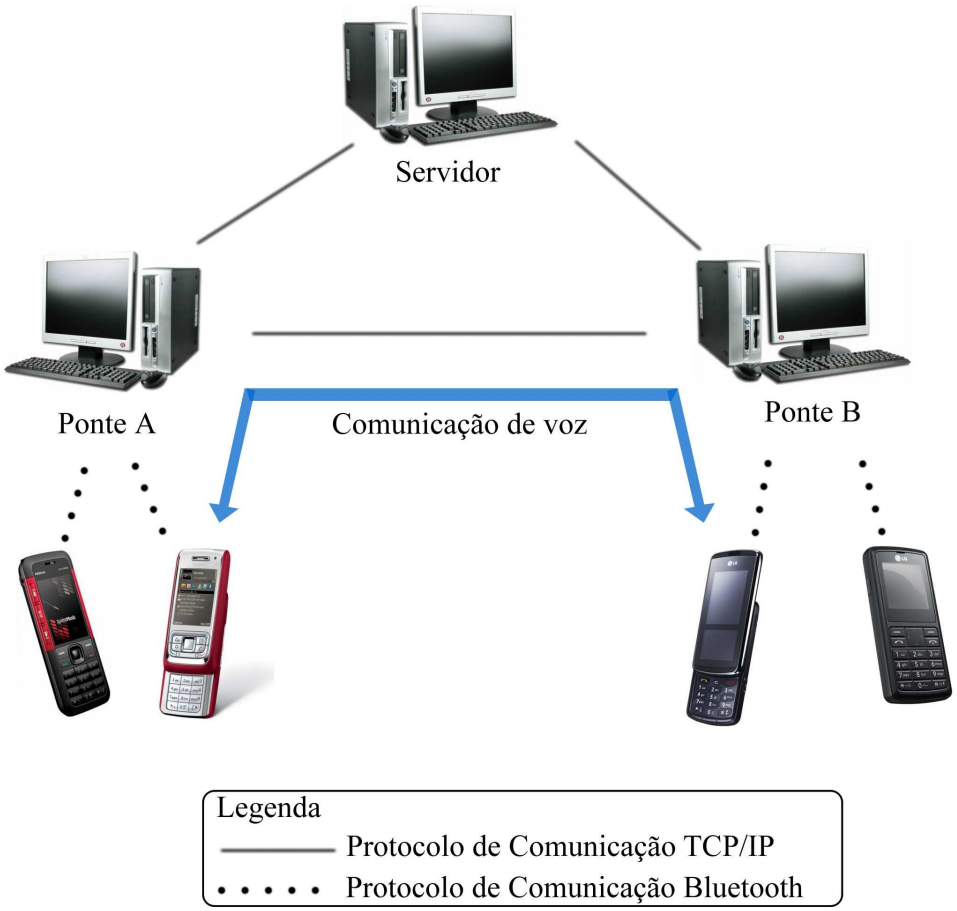

Figura 4.1: Esquema do sistema proposto

A Figura 4.1 demonstra o que se almeja com a aplicação implementada em sua totalidade. É possível observar que deve haver, no mínimo, um servidor e duas pontes diferentes. A conversação exigida deve ser compreendida entre dois dispositivos localizados em áreas de alcance *Bluetooth* de pontes diferentes, ou seja, os aparelhos na área de abrangência da Ponte A devem se comunicar com os da área da Ponte B. Deve-se salientar que para dispositivos localizados em uma mesma ponte a funcionalidade de conexão não foi implementada, visto que não faz sentido existir uma comunicação tão próxima entre dois aparelhos.

Para melhor entendimento, as funcionalidades de cada componente do esquema da Figura 4.1 serão detalhadas nas seções subseqüentes.

4.1 Estabelecimento da Conexão

Para o funcionamento do sistema proposto, os componentes demonstrados na Figura 4.1 devem possuir a capacidade de conectar-se entre si, sendo que para cada um deles uma implementação de *software* diferente é necessária. Desta forma, esta seção tem por finalidade explicar cada elemento do esquema apresentado.

- Passo 01: dispositivo realiza busca por uma ponte localizada no raio de ação do seu *Bluetooth*. Quando este processo é finalizado, uma lista das pontes encontradas é apresentada na tela do dispositivo, desta maneira o usuário pode selecionar uma para realizar conexão e então ser cadastrado no servidor. Este esquema é demonstrado na Figura 5.2:

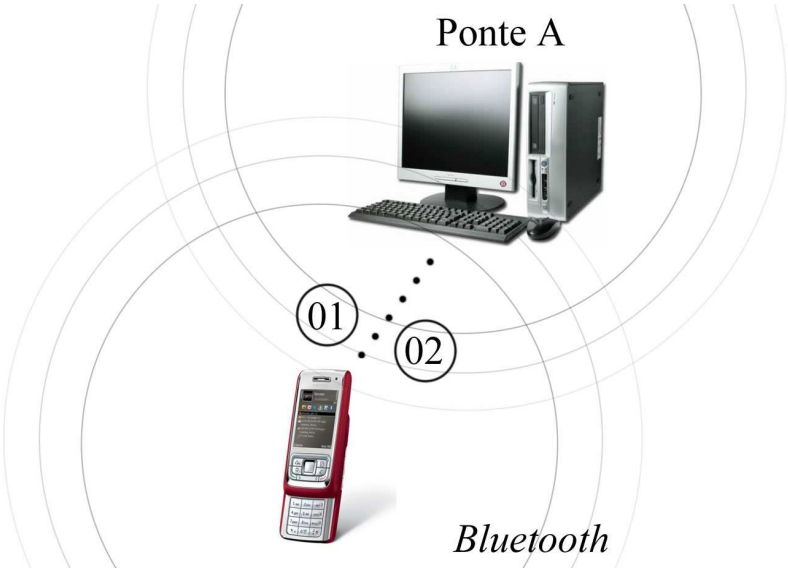

Figura 4.2: Dispositivo efetua busca por pontes.

Conforme o esquema apresentado na Figura 4.2 é possível apresentar as seguintes funcionalidades:

- Ação 01: o dispositivo procura por pontes localizadas em sua área de abrangência *Bluetooth*. Para permitir este processo, o *Bluetooth* da ponte deve estar configurado para ser encontrado por dispositivos de sua área. Desta maneira, a ponte pode fornecer seu serviço de conexão para a rede *Bluetooth* em que está inserida via

UUID (*Universally Unique Identifier*). O dispositivo por sua vez, realiza uma busca por serviços e assim que encontrar efetua a conexão com a ponte, via *StreamConnection*, para então utilizar de seus serviços.

- Ação 02: assim que encontradas as pontes o usuário do dispositivo pode acionar uma ponte específica para efetuar sua conexão junto ao servidor. Este processo é realizado através da informação de uma ação específica à ponte, onde esta por sua vez ao recebê-la, iniciará a comunicação com o servidor para efetuar o cadastro do dispositivo conectado. Para efetuar esta ação, fora definida uma classe chamada *Packet*, a qual possui as seguintes informações:
  - Nome do Cliente: atributo nominativo do dispositivo.
  - Endereço *Bluetooth* do dispositivo: informação essencial para que uma ponte realize uma conexão com o dispositivo alvo da transação.
  - Endereço IP da ponte: através desta informação a conexão entre pontes torna-se possível.
  - Nome da Ponte: atributo utilizado para informar em qual ponte o dispositivo está conectado.
  - Ação a ser realizada: variável do tipo inteiro que define qual procedimento a ponte ou o servidor irá realizar. O valor 1 (um), por exemplo, informado a uma ponte, requisita que ela efetue uma conexão ao servidor e então um cadastro do seu cliente.

➢ Passo 02: a Ponte A (do dispositivo que origina a conexão), conecta-se ao servidor para então efetuar o cadastro do dispositivo nela conectado. Assim que este processo finaliza, uma lista de dispositivos (clientes) cadastradas no servidor é retornada ao dispositivo que originou a conexão. Este processo é melhor entendido com a figura abaixo:

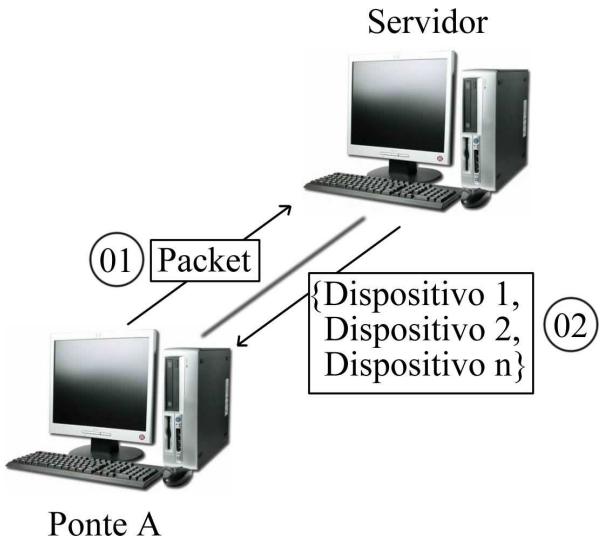

Figura 4.3: A Ponte A comunica-se com o Servidor

Dado o esquema da Figura 4.3, é possível detalhar as seguintes ações necessárias ao funcionamento do sistema.

- Ação 01: a Ponte A efetua uma conexão TCP/IP com o servidor, para isto, o endereço IP desta máquina deve ser informado a todas as pontes da arquitetura. Depois de realizada a conexão, um objeto *Packet* contendo os dados do cliente é submetido ao servidor, que o recebe e adiciona em uma estrutura de lista, *List*. É importante salientar que o servidor confere se o cliente já está cadastrado em alguma outra ponte, se já o estiver, o novo *Packet* recebido substitui o encontrado na lista presente. Esta característica é verificada devido ao fato de um dispositivo poder migrar de áreas de abrangência de pontes.
- Ação 02: após o cadastro no servidor, o cliente pode requisitar a lista dos dispositivos em que ele pode se conectar, denominados dispositivos alvos. Uma relação de objetos do tipo *Packet* é então enviada ao dispositivo, para que assim o usuário selecione um dispositivo específico a se conectar.

- Passo 03: o dispositivo recebe a lista de clientes conectados ao servidor, o usuário pode então, selecionar um deles para efetuar conexão. Esta é requisitada pelo envio de uma ação à ponte conectada. Este processo é melhor detalhado na Figura 4.4:

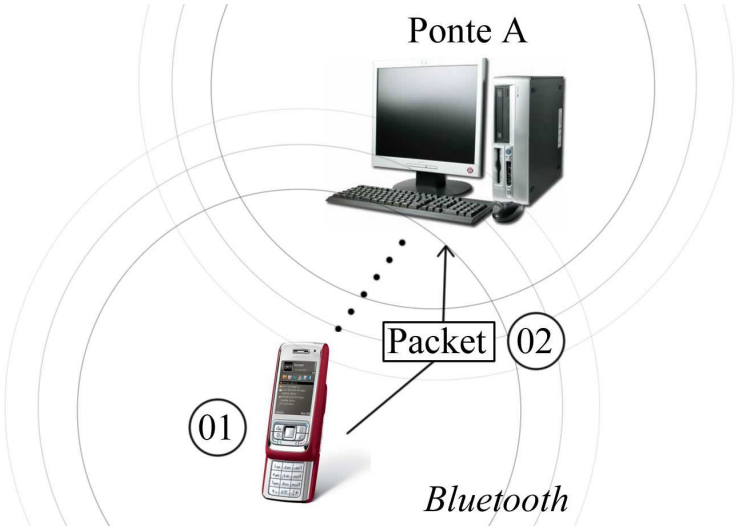

Figura 4.4: Usuário seleciona um cliente para conexão.

- Ação 01: o dispositivo recebe a lista de clientes cadastrados no servidor e a exibe na tela para que o usuário selecione um a se conectar.
- Ação 02: assim que selecionado o cliente para conexão, um *Packet* deste é enviado novamente à ponte para que esta efetue uma conexão com a ponte em que o dispositivo alvo está conectado. Esta conexão torna-se possível devido ao atributo de Endereço IP da ponte encontrado no objeto *Packet*.

➢ Passo 04: neste passo, a Ponte A recebe a requisição de conexão enviada pelo dispositivo de origem e estabelece uma conexão com a ponte destinatária informada no objeto *Packet* recebido. Este processo é melhor compreendido pela visualização da figura seguinte:

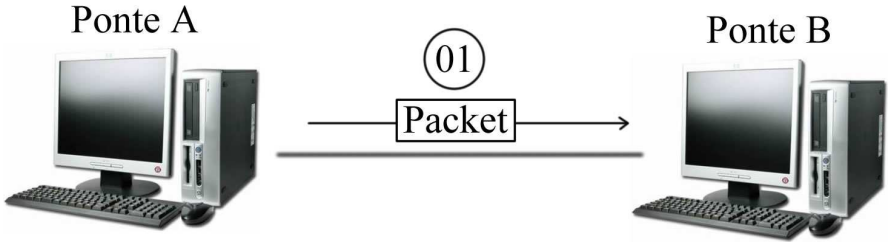

Figura 4.5: Ponte A estabelece conexão com Ponte B

- Ação 01: assim que a Ponte A recebe o objeto *Packet* enviado pelo dispositivo, esta obtém o atributo Endereço IP da ponte para requisitar uma conexão TCP/IP com a ponte destinatária, Ponte B. Assim que a conexão for estabelecida o *Packet* é repassado da Ponte A para B, para que esta possa através do atributo de Endereço *Bluetooth* do Dispositivo, do objeto transmitido, iniciar uma conexão com o dispositivo alvo requisitado.

➢ Passo 05: nesta etapa, a Ponte B recebe o objeto *Packet* da Ponte A e com uso do atributo Endereço *Bluetooth* do Dispositivo, inicia uma conexão com este, onde a confirmação de estabelecimento desta deve ser retornada à Ponte A e ao dispositivo de origem. A Figura 4.6 representa esta fase:

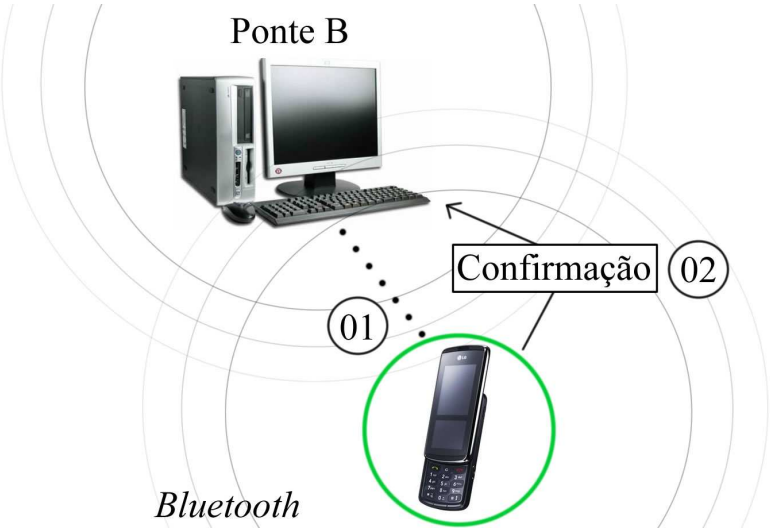

Figura 4.6: Ponte B estabelece conexão com dispositivo alvo.

- Ação 01: a Ponte B emite uma requisição de conexão com o dispositivo alvo através do endereço *Bluetooth* encontrado no objeto *Packet* recebido da Ponte A. O seguinte comando é efetuado para realizar esta ação:

```
stream = (StreamConnection) Connector.open(url);
```

Onde, *stream* representa um objeto do tipo *StreamConnection* e `Connector` é o responsável por efetuar a conexão através do endereço *url* do dispositivo alvo. Para que esta conexão seja efetuada, é necessário que o *software* do dispositivo emita uma *thread* com um UUID de serviço para permanecer aguardando novas conexões.

- Ação 02: assim que a conexão é efetuada uma confirmação é enviada à Ponte A para que informe o dispositivo de origem que está autorizado a iniciar uma transmissão de áudio ao aparelho alvo.

Assim que o Passo 05 é finalizado, com a confirmação de conexão por parte do dispositivo alvo, o esquema da Figura 4.1 pode ser verificado, ou seja, os dispositivos já estão conectados entre si, através das pontes de conexão, para iniciarem uma transmissão de voz. Esta propriedade será detalhada na seção seguinte.

4.2 Desenvolvimento da Estrutura

Para cada um dos três componentes do sistema proposto foi desenvolvida uma aplicação diferente. Um CD-ROM contendo o código fonte da implementação do projeto está contido como Apêndice A. O nome e as propriedades de cada uma delas são explicados abaixo:

- *SoftList*: *MIDlet* principal do dispositivo. Realiza a interface com o usuário.
- *Bridge*: Aplicação *desktop* desenvolvida em J2SE para gerenciar as conexões do dispositivo, tanto com o servidor quanto com outras pontes.
- *Server*: Aplicação *desktop* desenvolvida também em J2SE para realizar o armazenamento dos clientes conectados e suas respectivas pontes.

As próximas seções fazem o detalhamento de cada uma destas aplicações, bem como a estrutura de comunicação entre as classes internas de cada uma constatada no final do projeto.

4.2.1 Aplicação SoftList

Um *MIDlet* principal, denominado *SoftList*, foi definido como *software* central do dispositivo. Nesta aplicação, um conjunto de funcionalidades fora desenvolvido para permitir a integração entre os componentes do sistema. As principais ações construídas foram:

- Busca e conexão por pontes disponíveis: esta ação realiza uma busca pelos serviços de conexão oferecidos pelas Pontes de Conexão e Transmissão. Depois de finalizada a pesquisa, uma lista das pontes encontradas é retornada para seleção de conexão do usuário.
- Registro no servidor: submete uma ação à aplicação da ponte (*Bridge*) para que esta efetue o registro do dispositivo junto ao *software* do servidor (*Server*).
- Exibição dos clientes disponíveis para conexão: submete uma ação à aplicação *Bridge* para que esta requisite ao servidor a lista de dispositivos nele registrado.
- Conexão com dispositivos: permite que o usuário selecione um dos dispositivos da lista retornada do servidor para efetuar conexão.
- Gravação e envio de voz: esta funcionalidade é efetuada através de uma *thread* que possui a responsabilidade de gravar e enviar os dados de áudio (*array* de *bytes*) recebidos pelo dispositivo.
- Reprodução de áudio: realiza a reprodução dos dados de áudio recebidos, através de uma *thread*, de outro dispositivo.

As duas últimas funcionalidades mencionadas acima, de gravação e envio de voz e reprodução de áudio, foram construídas com o auxilio de *threads*. A relação destas com o *MIDlet* principal é melhor compreendida com a visualização do diagrama de classes da Figura 4.7:

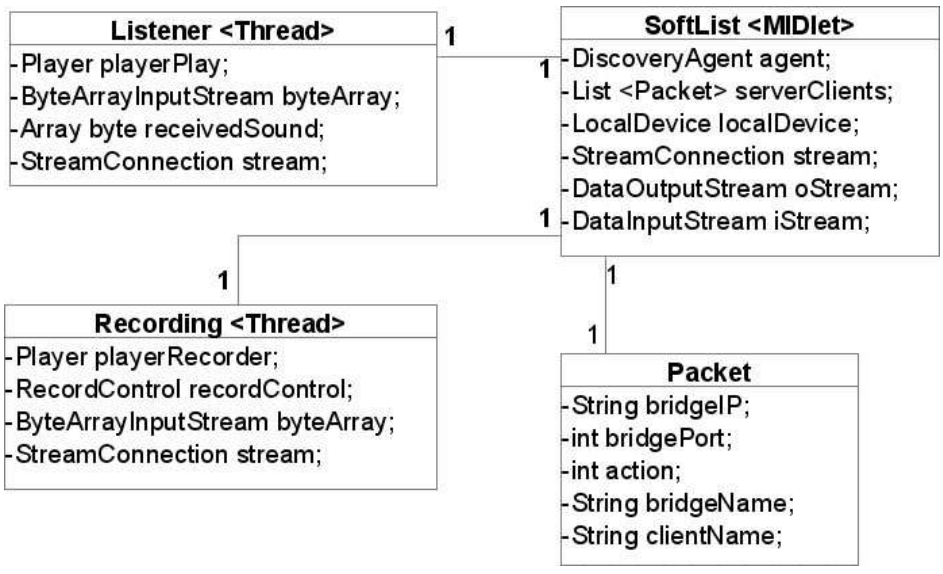

Figura 4.7: Diagrama de classes do componente Dispositivo

A classe *Packet*, como mencionado anteriormente, possui a finalidade de representar um dispositivo e seus dados de conexão com uma ponte. No *MIDlet* principal, é possível observar as variáveis mais importantes envolvidas na implementação deste, sendo elas:

- `DiscoveryAgent agent`: responsável pela busca *Bluetooth* por pontes disponíveis para conexão [06].
- `List <Packet> serverClients`: estrutura do tipo *List* (lista) responsável pelo armazenamento de uma lista de clientes registrados no servidor.
- `LocalDevice localDevice`: possui a finalidade de obter e manter as informações do perfil do dispositivo [06].
- `StreamConnection stream`: responsável pela abertura de conexão com uma ponte.
- `DataOutputStream oStream`: variável de fluxo de transmissão do sentido dispositivo - ponte.
- `DataInputStream iStream`: variável de fluxo de transmissão do sentido ponte - dispositivo.

As duas *threads* evidenciadas no diagrama da Figura 4.7, *Recording* e *Listener*, são designadas para realizarem respectivamente a gravação a reprodução do áudio recebido de outro dispositivo. Estas são criadas somente a partir do momento em que o dispositivo encontra-se registrado no servidor e permanecem executando até que o usuário finalize o *SoftList* com o comando *Exit*.

Ambas as classes descritas acima possuem um objeto *Player*, responsável pela gravação e reprodução do áudio emitido pelo usuário do dispositivo. Pode-se observar também o uso de variáveis auxiliares tanto na *thread Recording* quanto na *Listener*, como o *array* de *bytes* `receivedSound` verificado nesta última.

A conexão com a ponte selecionada pelo usuário é gerenciada com a variável *stream* de fluxo de conexão encontrada tanto em *SoftList* como nas duas *threads Recording* e *Listener*. Com elas, torna-se possível a troca de dados entre o dispositivo e sua respectiva ponte.

4.2.2 Aplicação Bridge

Esta aplicação foi desenvolvida para ser executada nas pontes e tem por intenção receber solicitações de conexões de dispositivos e de pontes de conexão, bem como gerenciar a transferências dos dados de áudio de um dispositivo para outro.

Após uma conexão ser aceita, tanto de uma ponte como de um dispositivo, a *Bridge* emite uma *thread* chamada *BridgeConn* responsável pela recepção das ações emitidas pela aplicação *SoftList* do dispositivo, sendo elas:

- Solicitação de registro no servidor;
- Obtenção da lista dos clientes registrados;
- Conexão com outro dispositivo;
- Exclusão de registro;

A transferência de dados de uma ponte à outra, e conseqüentemente de um dispositivo a outro é realizada por intermédio de duas *threads* específicas, sendo elas:

- *TransBridge*: responsável por receber os dados de áudio oriundos de um dispositivo conectado na ponte e reenviá-los à outra ponte de conexão.
- *TransDevice*: recebe os dados de áudio de uma ponte e reenvia-os ao dispositivo alvo.

Para melhor compreender a relação da classe *Bridge* com suas *threads* de comunicação, um diagrama de classes é apresentado na Figura 4.8:

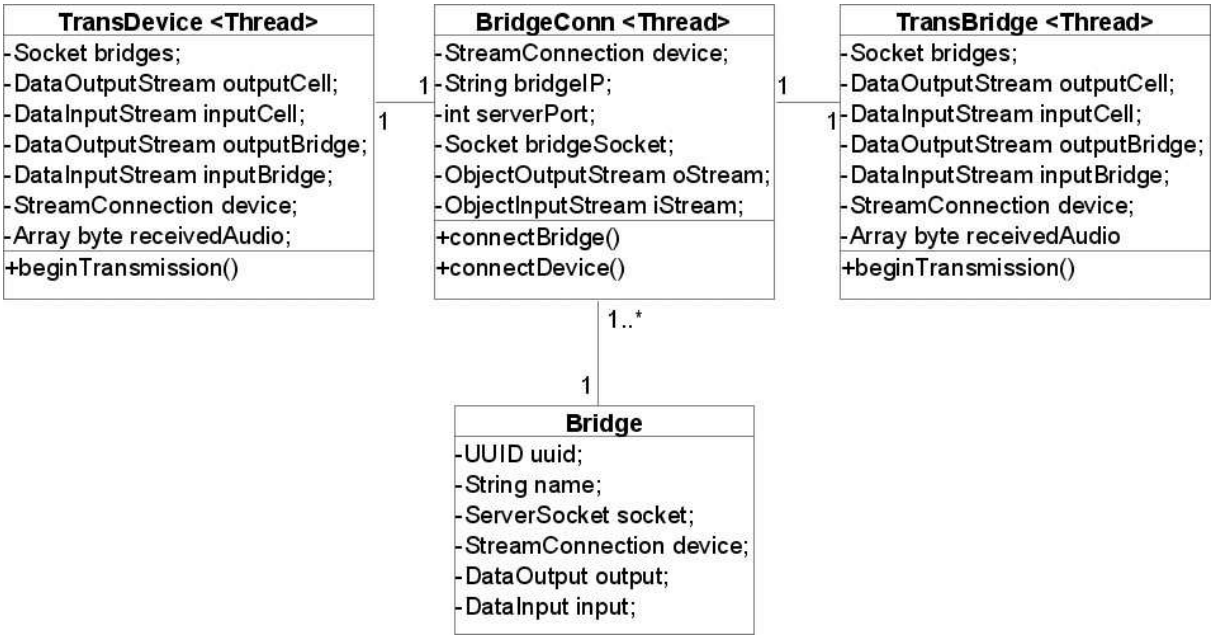

Figura 4.8: Diagrama de classes relacionado à aplicação *Bridge*.

.

Na figura acima é possível observar as relações da classe *Bridge* com suas classes originárias. A *Bridge* permanece esperando por novas solicitações de conexões, tanto de dispositivos como de outras pontes, através de duas variáveis diferentes, `socket` e `device`, que respectivamente são do tipo *ServerSocket* e *StreamConnection*. Assim que uma destas requisições é aceita, uma *thread BridgeConn* é emitida para comunicação com os componentes de origem, dispositivo ou ponte. Outros atributos importantes também são citados, como as variáveis de entrada e saída de dados de uma *stream* de conexão, `input` *e* `output`, assim como o UUID do serviço de conexão *Bluetooth*.

A classe *BridgeConn* armazena a informação de seu endereço na rede (IP), além das *streams* de entrada e saída de objetos serializáveis para comunicação com a aplicação do servidor.

Para que uma conexão seja estabelecida entre duas pontes, é necessário que a *thread BridgeConn* da ponte originária, através do método `connectBridge()` emita uma *thread TransBridge* responsável por enviar uma solicitação de conexão TCP com o endereço da ponte obtido no *Packet* recebido pelo dispositivo. A outra ponte, que recebe a solicitação, aceita a conexão e obtém a *stream* de conexão do dispositivo alvo nela conectado, para então emitir uma *thread TransDevice* que permanece comunicando-se com o dispositivo alvo e a

ponte originária da transação. O processo inverso é realizado pela ponte de destino para que o seu dispositivo possa enviar dados de áudio para a ponte originária da conversação.

As *threads TransDevice* e *TransBrige* possuem cada uma um *buffer* para recepção dos dados de áudio provenientes do dispositivo conectado à ponte como também de outras pontes de conexão. A estrutura utilizada para o *buffer* neste projeto deu-se por um *array* de *bytes*, caracterizando assim que uma *Bridge* possui um tempo determinado, pelo preenchimento do *buffer*, entre a recepção dos dados de áudio e o seu envio.

4.2.3 Aplicação Server

A aplicação *Server* é a responsável pelo armazenamento dos cadastros dos clientes conectados em suas respectivas pontes de conexão. A figura abaixo demonstra o diagrama de classes correspondente à arquitetura de implementação do servidor:

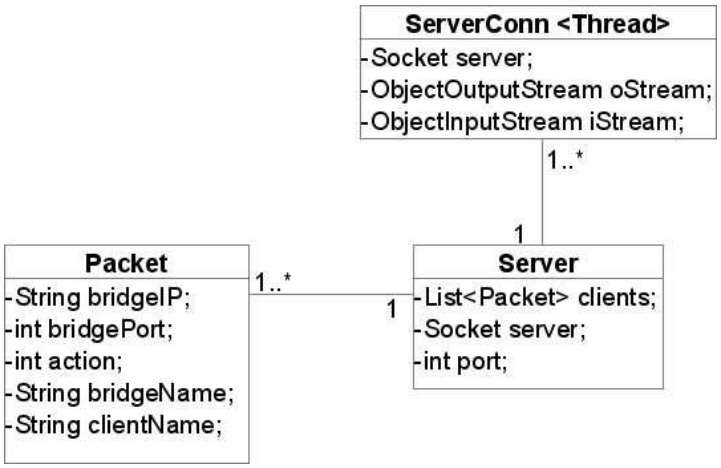

Figura 4.9: Diagrama de classes relacionado à aplicação *Server*.

Na Figura 4.9 pode-se evidenciar como se dá a conjuntura das classes da aplicação do servidor, bem como seus relacionamentos. O funcionamento do servidor é semelhante ao observado anteriormente da arquitetura de uma ponte, ou seja, uma aplicação denominada *Server* permanece ouvindo novas solicitações de conexão TCP. Após aceitar uma destas requisições, uma *thread ServerConn* é lançada para receber as ações emitidas pela ponte de origem. Desta forma, o servidor pode receber conexões de vários clientes (pontes) em um mesmo instante.

A *thread ServerConn* possui acesso à lista de *Packets* (pública) em *Server*, definida como `clients`, e pode assim realizar modificações nela, inserindo ou removendo clientes. Deve-se salientar que quando uma solicitação de registro de dispositivo é realizada, o método de inserção da *thread ServerConn* faz uma busca pelo nome do dispositivo a ser registrado, se este for encontrado, a instância dele é substituída pela atual, garantindo assim que um cliente se registre em outra ponte para iniciar uma conversação, sem necessitar retirar seu registro da atual.

4.2.4 Estrutura da Conexão

Dada a estrutura das classes e suas interações, à medida que o usuário solicita realizar conversação com outro cliente, uma série de passos, já explanados, é realizada para que seja estabelecida uma conexão entre os dois dispositivos. Desta forma, a Figura 4.10 apresenta o esquema final de conexão obtido com os processos comunicativos das estruturas envolvidas no projeto.

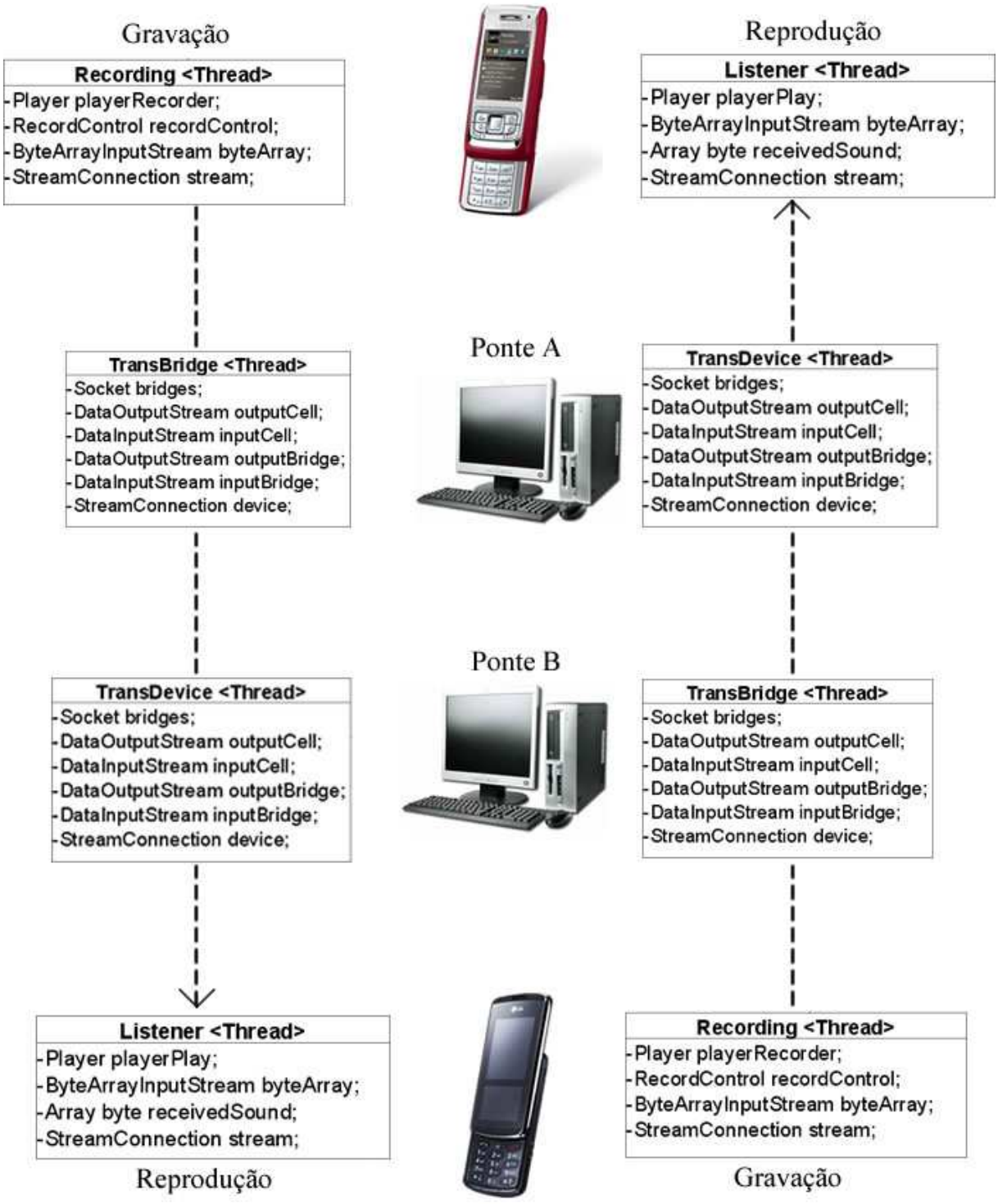

Figura 4.10: Esquema geral de comunicação entre as estruturas.

Com a visualização do diagrama da Figura 4.10, é possível perceber como se estabeleceu a comunicação entre as estruturas implementadas no projeto. A comunicação efetiva é realizada por intermédio das classes acima exibidas.

O exemplo da Figura 4.10 demonstra uma conexão simples entre dois dispositivos celulares. As duas *threads* emitidas por estes para realizarem gravação, envio e reprodução de voz, estabelecem comunicação direta com as *threads* emitidas pelas aplicações *Bridges*. Sendo assim, a transferência de dados de uma ponte para outra é realizada por intermédio das *threads TransBridge* e a comunicação das pontes com seus dispositivos através da *thread TransDevice*.

4.2.5 Envio e Recepção de Voz

Assim que o *SoftList* emite as *threads* de gravação e envio e recepção e reprodução dos dados de áudio, é necessário que exista uma verificação para emitir os comandos “falar” e “ouvir”. Esta propriedade deve ser verificada no presente projeto, pois os dispositivos celulares utilizados em sua construção carecem da propriedade *audio mixing* necessária para um dispositivo possuir dois *Players* ativos (gravação e reprodução) em um mesmo momento. A seção 4.3 denominada Dificuldades Encontradas realizará um detalhamento sobre este assunto.

Para garantir a gravação, envio e recepção dos dados de áudio, é necessário que os dois dispositivos envolvidos na conversação tenham conhecimento de seus estados de execução, isto é, o dispositivo A deve informar ao dispositivo B que iniciará uma transmissão de áudio e vice-versa. Esta característica pode ser futuramente resolvida com uma funcionalidade de informação de estado, melhor compreendida com a visualização da Figura 4.11:

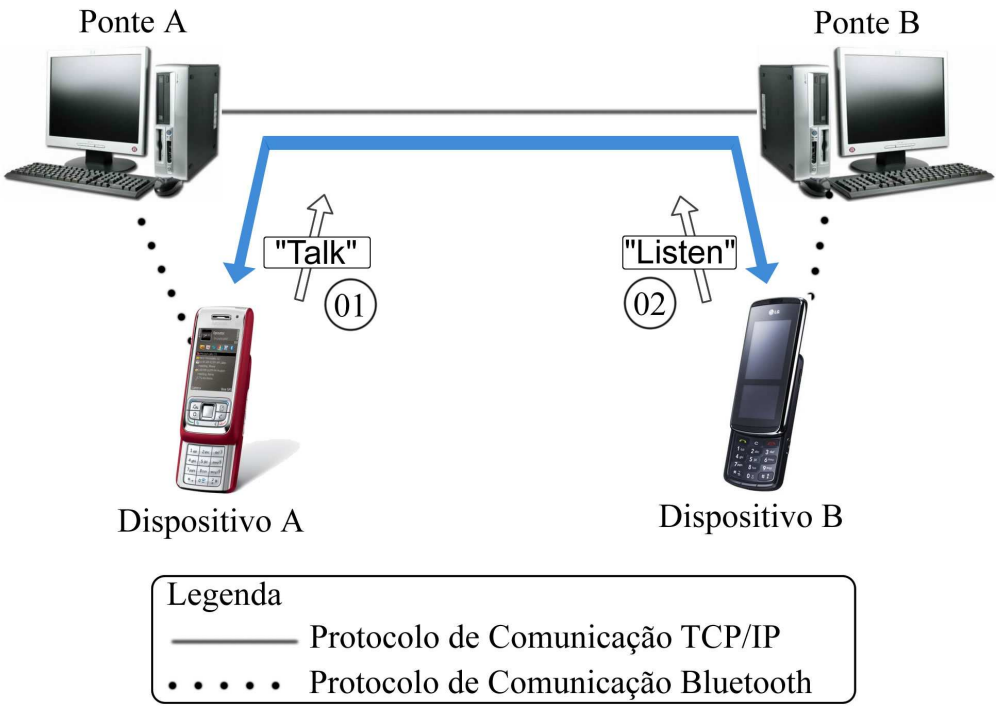

Figura 4.11: Troca de mensagens entre dispositivos.

Os passos executados para o início dos processos de gravação e emissão, ou recepção e reprodução de áudio são:

- Passo 01: O Dispositivo A, originário da conversação, emite o comando "*Talk*" ao Dispositivo B, informando a este que dará início em sua gravação e posterior transmissão de áudio.
- Passo 02: O Dispositivo B por sua vez, ao receber o comando, entrará em estado "Listen", prontificando seu *Player* de reprodução (*thread Listener*) para tratar os dados recebidos de A. O mesmo processo é realizado assim que o Dispositivo B emite o comando "*Talk*" para A.

O *MIDlet* principal, *SoftList*, possui em sua interface com o usuário que possui os dois comandos para controle de gravação (*talk*) e reprodução (*listen*), sendo melhor ilustrados na Figura 4.12:

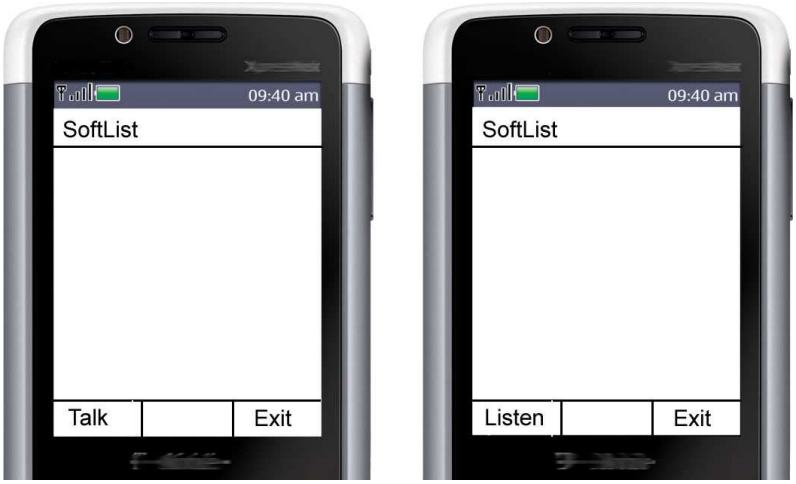

Figura 4.12: Comandos *Talk* e *Listen* exibidos ao usuário.

Como demonstra a Figura 4.12, assim que o usuário pressiona o botão *Talk*, canto inferior esquerdo do *SoftList*, ele é alterado para *Listen*, e vice-versa, permitindo assim que os estados do dispositivo sejam alterados entre "*Talk*" e "*Listen*".

4.3 Dificuldades Encontradas

A experiência de implementar uma arquitetura de conversação entre dois dispositivos localizados em diferentes regiões de abrangência de pontes de conexão *Bluetooth*, deparou-se com um complicador fundamental, a inviabilidade do dispositivo utilizado nos testes reproduzir e gravar dados de voz ao mesmo tempo, uma característica vital para um bom funcionamento de um telefone.

O modelo de celular utilizado nos testes realizados na construção do projeto foi o Nokia 5610 *XpressMusic*. Este aparelho é classificado como série 40 e possui o sistema operacional mono tarefa Nokia OS, ou seja, não é tão robusto como o *Symbian* dos dispositivos da série 60, característica que lhes permitem executar mais de uma aplicação ao mesmo tempo [27]. Além disto, não possuem a propriedade *audio mixing*, o que impede que dois objetos *Player* estejam em execução ao mesmo tempo [26].

Os aparelhos da série 40 possuem apenas a característica de *swap and play*, que consente que mais de um *Player* entre no estado *prefetch*, porém, não é possível que mais de um destes objetos possam executar em um mesmo momento. Uma chamada ao método *start* de um *Player* não paralisa a execução de outro [26].

Os dispositivos da série 60, que possuem o sistema operacional *Symbian*, conseguem executar mais de uma aplicação ao mesmo tempo [27], mas ainda são pouco difundidos entre os usuários.

Durante toda a etapa de implementação do sistema proposto não foram utilizados emuladores de dispositivos celulares. Ao invés disto, a aplicação era compilada e testada efetivamente no aparelho, fato que ocasionou diversos retardos no processo devido às complicações impostas pela arquitetura e tecnologia não tão eficiente do mesmo, bem como a prática de *debug* ser complicada. A opção por não utilizar emuladores deu-se pelo fato destes *softwares* não possuírem as características específicas de *hardware* e de sistema operacional de cada aparelho. Muitos problemas encontrados durante os testes no dispositivo não seriam evidenciados em um emulador, como por exemplo, o uso dos *Players*.

4.4 Testes Efetuados e Resultados Obtidos

Para a realização dos testes foram utilizados dois dispositivos celulares Nokia 5610 da série 40, como também três computadores que representaram duas pontes e um servidor. A comunicação *Bluetooth* das pontes com os celulares foram realizadas através de adaptadores *Bluetooth* USB *Encore*.

Os testes promovidos focalizaram a velocidade de transmissão de dados entre os dispositivos, ou seja, o tempo gasto para isto. Este fator foi calculado considerando-se o tempo gasto entre os processos de:

1. Início da transmissão do trecho de áudio gravado;
2. Recebimento e reenvio pela ponte de origem;
3. Recebimento e reenvio pela ponte de destino;
4. Recebimento das informações no dispositivo de destino;

Considerando este processo, fora definida uma estrutura de comunicação para permitir a obtenção das informações relevantes ao projeto. Dados três computadores localizados na mesma rede interna de comunicação, dois deles foram caracterizados como pontes, executando a aplicação *Bridge*. Para a outra máquina, foi atribuída uma aplicação *Server*, desempenhando assim o servidor.

Para calcular o tempo gasto pela transmissão dos pacotes de áudio, foi implementado um esquema semelhante a um RTT (*Round Trip Time*) onde o contador de milissegundos do dispositivo tem sua contagem inicializada no momento em que o segmento começa a ser transmitido. Após o pacote ser enviado pelas duas pontes e chegar ao dispositivo alvo, este o reenvia para a sua ponte de conexão, que o retransmite para a ponte do celular de origem, que por sua vez, ao recebê-lo novamente paralisa seu contador de milissegundos para obter o intervalo médio entre o envio e a confirmação de recebimento.

Os resultados obtidos são demonstrados na Tabela 5.1, onde cada trecho de áudio foi transmitido dez vezes para a obtenção de uma média dos tempos de emissão e recebimento por parte do dispositivo de origem. O tempo médio de transmissão obtido esta denotado em milissegundos (ms).

Tabela 5.1: Dados da transmissão dispositivo – dispositivo.

Trecho de áudio	*Bytes*	Tempo médio de transmissão	Desvio Padrão
1 segundo	1286	55,7 ms	17,82 ms
2 segundos	3078	71,6 ms	8,42 ms
3 segundos	4614	91,9 ms	18,55 ms

Também foram efetuados testes de transmissões somente entre o dispositivo e sua ponte, visando obter somente seus tempos de transmissão via *Bluetooth*. O esquema realizado é semelhante ao efetuado no teste anterior, porém neste, o pacote é reenviado ao dispositivo assim que ele chega a sua ponte de conexão. Os valores encontrados são exibidos na Tabela 5.2:

Tabela 5.2: Taxas Dados da transmissão ponte – dispositivo.

Trecho de áudio	*Bytes*	Tempo médio de transmissão	Desvio Padrão
1 segundo	1286	24,1 ms	13,11 ms
2 segundos	3078	28 ms	1,65 ms
3 segundos	4614	30,4 ms	7,9 ms

Desta forma, pode-se perceber que o tempo requerido para uma transmissão de pacote de áudio é significativamente baixo, pois a sua incidência no dispositivo alvo dá-se de maneira quase imediata. As Figuras 4.13 e 4.14 apresentam os gráficos com os valores de tempo de transmissão obtidos a cada iteração de transmissão de áudio nos testes realizados.

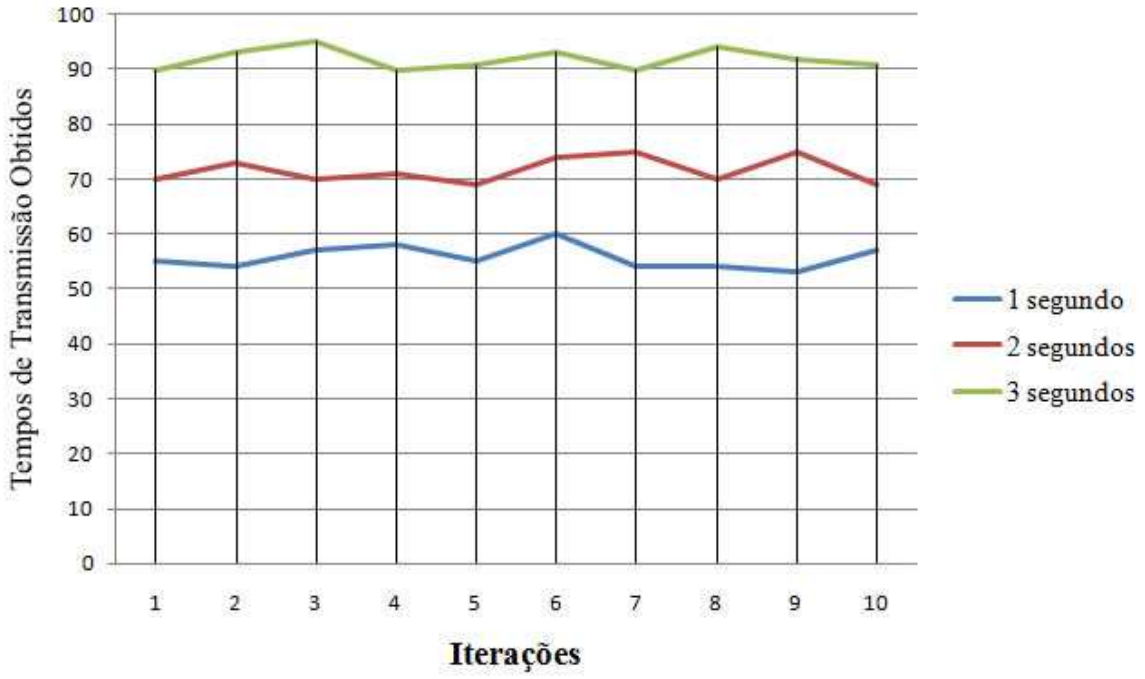

Figura 4.13: Gráfico dos tempos obtidos dispositivo – dispositivo

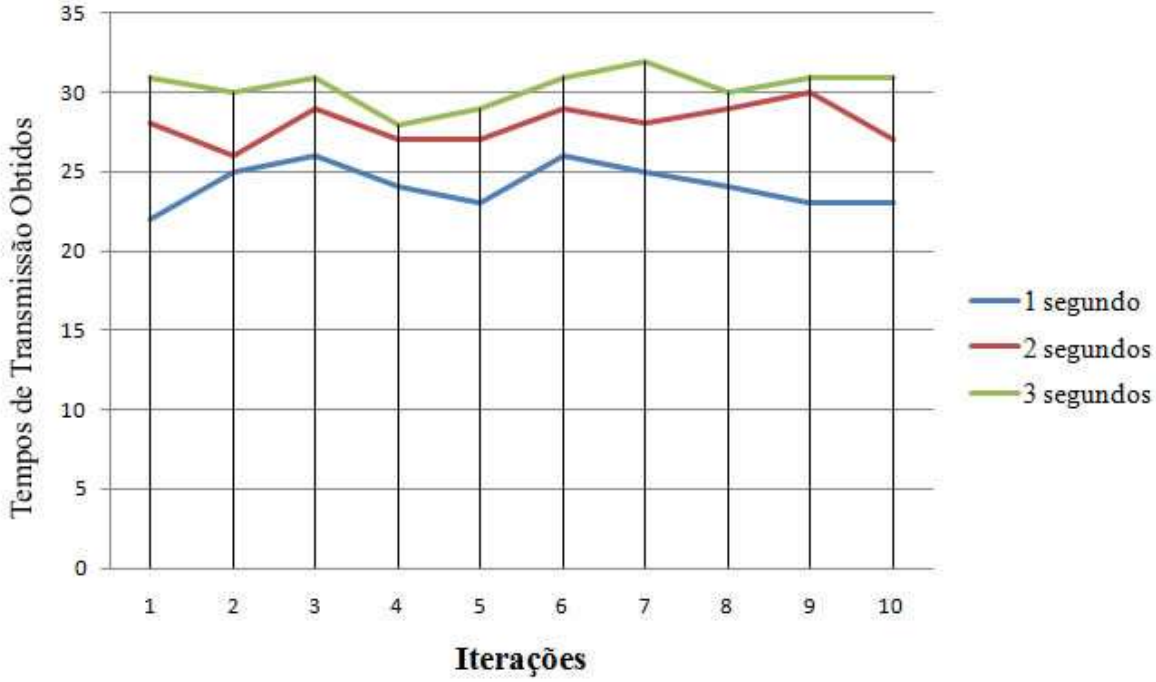

Figura 4.14: Gráfico dos tempos obtidos dispositivo - ponte

Capítulo 5

Considerações Finais

5.1 Conclusões

O fato de o sistema proposto ter sido inteiramente testado no dispositivo durante seu desenvolvimento nos permitiu evidenciar vários obstáculos devido às diferentes configurações, tanto de *hardware* como do sistema operacional, de cada aparelho. Desta forma, foi possível obter um conhecimento mais preciso sobre a tecnologia de desenvolvimento para plataformas móveis JME, bem como das limitações da tecnologia *Bluetooth*.

No sistema proposto a implicação de o pacote de áudio emitido pelo dispositivo necessitar ser transmitido via *Bluetooth*, e por TCP em seqüência, faz com que exista um atraso relacionado tanto nas ações emitidas pelo *SoftList* à sua ponte de conexão, como pela transmissão do áudio. Como verificado nos testes realizados, o atraso de transmissão de um único trecho de áudio não apresentou valores significativos para a aplicação desenvolvida.

As pretensões futuras com este projeto são de agregar sua comunicação a um servidor VoIP, como por exemplo o *Asterisk*. Para alcançar este objetivo, alguns aspectos de implementação devem ser pesquisados e aprimorados para que a conversação não encareça com atrasos que possam ocorrer na transmissão dos pacotes de áudio pela rede.

O fato de a aplicação não possuir um registro apropriado para ser executada no dispositivo impõe uma limitação ao usuário, a de ter que permanecer aceitando as requisições de acessos às propriedades

do sistema e de *hardware* do aparelho. Como por exemplo, a cada intervalo definido para gravação, uma solicitação de aceitação é emitida ao usuário, o que empobrece a comunicação e usabilidade.

Deve-se salientar que as diferentes características para cada dispositivo tornam-se um empecilho para o uso da aplicação. Este fato foi observado através dos modelos dos aparelhos selecionados para efetuar os testes da aplicação desenvolvida, onde estes por sua vez, não possuíam um sistema operacional robusto o suficiente para que a aplicação executasse corretamente. Esta característica ocasionou problemas nos processos de gravação e reprodução de áudio de maneira conjuntas no mesmo dispositivo, pois requererem tecnologias mais avançadas para funcionar corretamente. Desta forma, o uso eficaz da aplicação fica restrito a um conjunto limitado de usuários de aparelhos mais sofisticados.

Deve-se relatar também o problema evidenciado com o dispositivo *Bluetooth* utilizado nos testes da aplicação. Este aparelho apresentou anomalias inexplicáveis vindo a travar tanto nas atividades de coleta dos tempos de transmissão quanto durante o desenvolvimento do projeto.

Pode-se analisar nos testes realizados que a velocidade de transmissão do *Bluetooth* dos aparelhos celulares envolvidos permitiu que uma aplicação de emissão e recepção de pacotes de áudio, voz neste caso, fosse desenvolvida, porém apresentando atrasos na comunicação. Este aspecto pode ser amenizado no desenvolvimento de projetos futuros, através do uso de técnicas mais eficientes de comunicação.

5.2 Trabalhos Futuros

O que se pretende com o desenvolvimento da arquitetura de comunicação apresentada neste projeto, é integrá-la a um servidor VoIP, como o *Asterisk*. Para alcançar este objetivo, alguns aspectos de implementação devem ser pesquisados e aprimorados para que a conversação não encareça com atrasos que possam ocorrer na transmissão dos pacotes de áudio pela rede. Sendo assim, para estudos futuros, propõe-se:

- Adaptar a aplicação para execução em dispositivos móveis multi-tarefa, que permitam a gravação e reprodução simultânea de áudio;
- Estudar os protocolos para VoIP como SIP e H.323;
- Analisar e integrar o sistema a um servidor de VoIP como o *Asterisk*;

Apêndice A

CD-ROM com o código das aplicações

Referências Bibliográficas

[1] Ortiz, C. Enrique.; **Mobile Information Device Profile for Java 2 Micro Edition: Professional Developer's Guide**. Editora: John Wiley& Sons, 2001. 352 pg.

[2] Keogh, J.; **J2ME: The Complete Reference. Berkeley, California** - U.S.A. McGraw-Hill/Osborne, 2003. 721 p.

[3] Topley, K.; **J2ME in a Nutshell**. O'Reilly, 2002. 478 p.

[4] Sun Microsystems.; **J2ME Building Blocks for Mobile Devices: White Paper on KVM and the Connected, Limited Device Configuration (CLDC)**. Disponível em: *http://java.sun.com/products/cldc/wp/KVMwp.pdf*. Acesso em: 02/07/2009.

[5] Isakow, A. e Shi, H.; **Review of J2ME and J2Me-based Mobile Applications**. IJCSNS International Journal of Computer Science and Network Security, VOL 8 No.2, Melbourne, Australia, fev. 2008.

[6] Klingsheim, A. N.; **J2ME Bluetooth Programming**. 2004. 162 f. Tese (Mestrado em Informática) - Universidade de Bergen. 2004.

[7] Sun Microsystems.; **Mobile Information Device Profile (JSR-37): JCP Specification, Java 2 Platform, Micro Edition, 1.0a**. Disponível em: *https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_JCP-Site/en_US/-/USD/ViewFilteredProducts-SimpleBundleDownload*. Acesso em: 24/05/2009.

[8] Debbabi, M. et al.; **Embedded Java Security: Security for Mobile Devices**. 1a Edição. Springer, 2006. 243 pg.

[9] Sun Microsystems.; **Connected Limited Device Configuration: Specification Version 1.1.1**. Disponível em: *https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_JCP-Site/en_US/-/USD/ViewFilteredProducts-SimpleBundleDownload*. Acesso em: 10/06/2009.

[10] Menezes, A. L.; **Estudo da Viabilidade de Aplicação da Tecnologia Bluetooth na Criação de Ambientes Computacionais Ubíquos**. Cascavel, Paraná: Universidade Estadual do Oeste do Paraná, Dezembro, 2008. Monografia.

[11] Harte, L.; **Introduction to Bluetooth - Technology, Market, Operation, Profiles, and Services**. Althos, 2004. 60 pg.

[12] Kleer cut loose.; **ISM Band Coexistence Whitepaper**. Disponível em: *http://www.kleer.com/newsevents/whitepapers.php*. Acesso em: 21/04/2009.

[13] Bluetooth Special Group Interest. **Specification Of The Bluetooth System**. Disponível em: *http://www.bluetooth.com/Bluetooth/Technology/Building/Specifications/*. Acesso em: 01/07/2009.

[14] Knudsen, J.; Li, S.; **Beginning J2ME From Novice to Professional. 3a Edição**. Nova Yorke - U.S.A: Springer, 2005. 421 pg.

[15] Nokia.; **Java™ ME Developer's Library 2.3**. Disponível em: *http://www.forum.nokia.com/document/Java_Developers_Library_v2/?content=GUID-207D7DA7-A460-4E9F-AAAF-D4C96305B03A.html*. Acesso em: 14/05/2009.

[16] Goyal, V.; **Pro Java ME MMAPI:Mobile Media API for Java Micro Edition**. Estados Unidos da América: Apress, 2006. 255 pg.

[17] Anatel.; **"País tem 154,6 milhões de celulares em uso"**. Disponível em: *http://aeinvestimentos.limao.com.br/economia/eco28902.shtm*. Acesso em: 11/06/2009.

[18] Bazotti, E.; **VoIP para Computação móvel**. Cruz Alta, Rio Grande do Sul: Universidade de Cruz Alta, Dezembro, 2007. Monografia.

[19] **Projeto de Documentação do FreeBSD**.; Disponível em: *http://doc.fug.com.br/handbook/*. Acesso em: 15/07/2009.

[20] Java Community Process.; **JSR-118 Maintenance Release 2.1 Change Log**. Disponívem em: *http://jcp.org/aboutJava/communityprocess/maintenance/jsr118/JSR_118_MR2_Changelog.pdf*. Acesso em: 23/06/2009

[21] Giguere, E.; **Using Threads in J2ME Applications. Sun Developer Network (SDN)**. Fev. de 2003. Disponível em: *http://developers.sun.com/mobility/midp/articles/threading2/*. Acesso em: 22/06/2009.

[22] Sun Microsystems.; **Java™ 2 Platform Std. Versão 1.4.2. Sun Microsystems**, 2003. Disponível em: *http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html*. Acesso em: 11/07/2009.

[23] De Souza, C. E.; **Um Estudo de Caso Envolvendo Segurança em Soluções VoIP**. Cascavel, Paraná: Universidade Estadual do Oeste do Paraná, Dezembro, 2008. Monografia.

[24] Bluetooth Special Group Interest. **Bluetooth Protocol Architecture**. Versão 1.0. Disponível em: *http://www.bluetooth.com/NR/rdonlyres/C222A81E-D9F9-48CA-91DE-9C81F5C8B94F/0/Security_Architecture.pdf*. Acesso em: 04/06/2009.

[25] Sun Microsystems.; **Java™ APIs for Bluetooth™ Wireless Technology (JSR 82)**: Especification Version 1.1.1 JavaTM 2 Platform, Micro Edition. Disponível em: *https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_JCP-Site/en_US/-/USD/ViewFilteredProducts-ingleVariationTypeFilter*. Acesso em: 01/06/2009.

[26] Forum Nokia. *Midp:* ***Mobile Media API Developer's*** **Guide**. Disponível em: *http://www.forum.nokia.com/info/sw.nokia.com/id/f4f9f446-c335-4e23-ab39-5c49b003f202/MIDP_Mobile_Media_API_Developers_Guide_v2_0_en.pdf.html*. Acesso em: 02 de ago. 2009.

[27] Carlos Morimoto. **Celulares Nokia**: S40, S60, S80 e Maemo. Disponível em: *http://www.gdhpress.com.br/blog/celulares-nokia*. Acesso em: 29 de out. 2009.