UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GIANCARLO RAMPANELLI

Um Novo Método para Comunicação por Vídeo em Dispositivos Móveis com Sistema Operacional Android

Trabalho de Graduação.

Prof. Dr. Valter Roesler
Orientador

Porto Alegre, junho de 2011.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do CIC: Prof. Raul Fernando Weber
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço ao Prof. Valter Roesler pela orientação e conhecimento transmitidos ao longo de todo o ano no qual foi desenvolvido este trabalho. Agradeço ao colega Leonardo Crauss Daronco pelo conhecimento transmitido na área de codificação e de qualidade de vídeo, aumentando o meu interesse pela área. Agradeço aos colegas André Schulz, Felipe Cecagno e Jens-Michalis Papaioannou, pois tiveram importante participação e forneceram auxílio direto na implementação deste trabalho. Agradeço aos demais colegas do Laboratório de Projetos em Áudio e Vídeo da UFRGS (o PRAV), pois todos contribuíram – direta ou indiretamente – para a realização deste projeto.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
JNI	Java Native Interface
NDK	Native Development Kit
PC	Personal Computer
PCM	Pulse-code Modulation
PRAV	Projetos em Áudio e Vídeo
RTMP	Real Time Messaging Protocol
RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol
SDK	Software Development Kit
SDP	Session Description Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UFRGS	Universidade Federal do Rio Grande do Sul

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho tem o objetivo de apresentar uma nova estratégia para criação de videoconferências e sistemas de interação por vídeo em tempo real em dispositivos móveis com sistema operacional Android. É apresentado um estudo sobre as classes multimídia padrão do *kit* de desenvolvimento de *software* do Android, detalhando as limitações que dificultam o desenvolvimento de sistemas multimídia em tempo real sem a utilização de bibliotecas externas. Várias soluções propostas em trabalhos anteriores são comparadas, ressaltando prós e contras, e conduzindo à necessidade da utilização de uma nova estratégia, detalhada neste trabalho, que é baseada principalmente em bibliotecas alternativas. A ideia proposta é então validada num sistema Android real, mostrando suas vantagens. Dentre as implementações realizadas está a de uma API que encapsula as partes complexas da técnica, tornando simples o desenvolvimento de interação por vídeo no Android. A API - apresentada no capítulo 5.4 - encontra-se disponível em http://www.inf.ufrgs.br/prav/downloads_api_android.html.

Palavras-chave: Interação multimídia de tempo real, Vídeo, JNI.

A New Method for Video Communication on Android-powered Mobile Devices

ABSTRACT

This work aims to present a novel strategy for implementation of real-time interactive multimedia systems on Android-powered mobile devices. A study about the standard multimedia classes of Android's software development kit will be presented, detailing the limitations that hinder the development of real time multimedia systems without using external libraries. Some alternative solutions proposed in previous works will be presented, driving to the need of using a new strategy, detailed in this work, which is based primarily on alternative libraries. The proposed idea is then validated in a real Android system, showing its advantages. Moreover, an API that encapsulates the complex parts of the technique is developed, making it simple to implement video communication on Android. The API - presented in chapter 5.4 - is available in http://www.inf.ufrgs.br/prav/downloads_api_android.html.

Keywords: Real-time interactive multimedia, Video, JNI

1 INTRODUÇÃO

A grande popularidade dos aplicativos de vídeo combinada com um maior crescimento da venda de *smartphones* em comparação ao crescimento da venda de celulares simples (GARTNER, 2010) torna o mercado de programas de vídeo para *smartphones* muito promissor. Enquanto os sistemas operacionais que estão estabelecidos como dominantes de vendas há mais de um ano (como o Symbian, da Nokia, o BlackBerry, da Research in Motion, e o iOS, da Apple) já têm alguns aplicativos avançados de vídeo que, inclusive, funcionam para interação em tempo real, o sistema operacional para *smartphones* da Google (o Android, que está crescendo muito em numero de vendas) é caracterizado por aplicativos de vídeo mais simples. Isso se deve às limitações das APIs fornecidas pelo Android para programação de vídeo e à grande complexidade para criar tais aplicativos sem a utilização dessas APIs (JANSSEN, 2010).

Segundo (GARTNER, 2010), no terceiro trimestre de 2009, o Android dominava apenas 3,5% do mercado mundial de *smartphones* (em vendas para usuários finais). No entanto, em apenas um ano, segundo o mesmo autor, o Android passou a corresponder a 25,5% das vendas e, inclusive, ultrapassou o sistema operacional da Apple (o iOS, que roda no iPhone, por exemplo). O gráfico a seguir, gerado a partir dos dados coletados por (GARTNER, 2010), ilustra o significativo crescimento do Android com base no percentual de vendas de *smartphones* para usuários finais por sistema operacional.

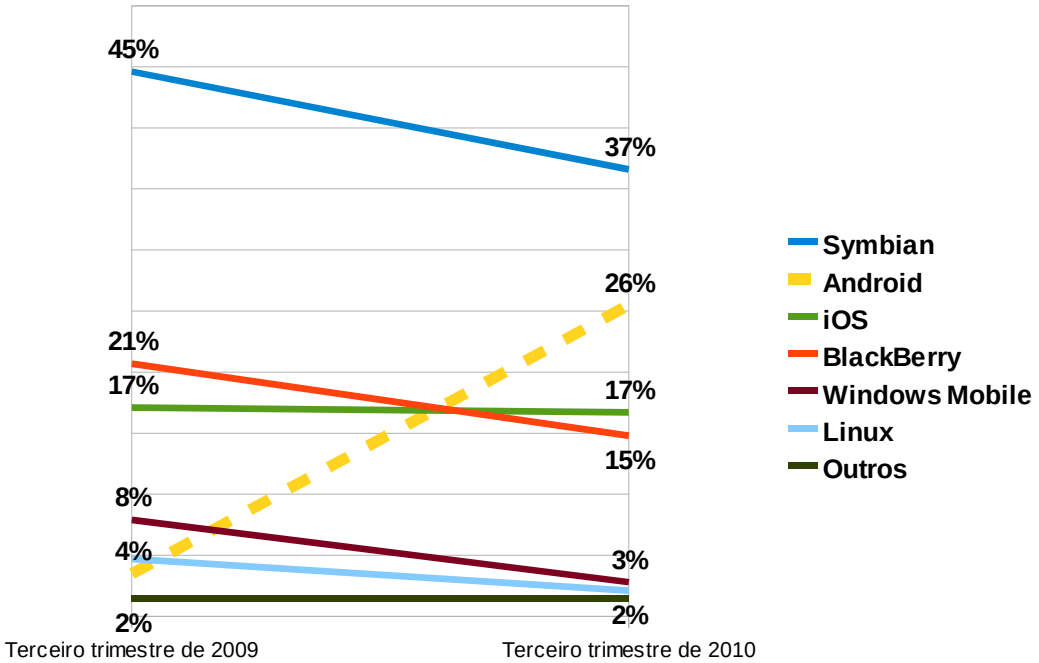


Figura 1.1: Evolução das vendas do Android em comparação aos outros *smartphones*

1.1 Objetivos e Justificativa

As APIs ou classes fornecidas pelo Android para o desenvolvimento de aplicativos de vídeo têm diversas limitações que impossibilitam o desenvolvimento de programas complexos, principalmente no contexto de interação por vídeo em tempo real. O objetivo deste trabalho, portanto, é a análise das alternativas e o desenvolvimento de uma solução para a criação de aplicativos de transmissão e recepção de vídeo em tempo real em dispositivos com Android.

Para justificar as escolhas feitas e a utilização de bibliotecas alternativas, primeiramente será realizada uma análise da Software Development Kit (Kit de Desenvolvimento de Software - SDK) do Android para multimídia, seguida de uma comparação de possíveis estratégias - propostas por trabalhos anteriores - para desenvolvimento de sistemas de interação por vídeo em tempo real com a utilização apenas das classes padrão do Android. Com isso, serão analisadas as limitações dessas estratégias e das APIs do Android.

Por fim, para validar a solução desenvolvida neste trabalho, ela será implementada na forma de aplicativos distintos e na forma de uma API que visa abstrair dos seus usuários (os programadores) as partes mais trabalhosas e menos intuitivas da ideia, para que seja possível a criação de tais aplicativos de vídeo para Android de forma simples e rápida.

1.2 Trabalhos Relacionados

(JANSSEN, 2010) apresenta um estudo das limitações das classes padrão do Android para o desenvolvimento de aplicativos de interação por vídeo em tempo real. Ele levanta diversas possíveis abordagens de uso dessas APIs, analisando cada uma das estratégias para determinar qual é a que teria um melhor resultado. Por fim, implementa a melhor ideia e indica que o resultado não é suficientemente bom para interação por vídeo, mas é o melhor que se pode alcançar utilizando-se apenas as classes padrão. No capítulo 3.1 será apresentada essa estratégia e a sua respectiva análise.

No trabalho de (SONG et al., 2010), os autores estendem o seu trabalho anterior (FU et al., 2010) e implementam um aplicativo para Android que reproduz vídeo a partir de arquivos de vídeo locais, sem utilizar apenas as classes padrão do Android. Para isso, eles utilizam uma ideia inicial semelhante à que será proposta e utilizada neste trabalho: portam a biblioteca FFmpeg[1] para o Android a nível de *kernel*, a fim de realizar a decodificação do vídeo em *software*, e obtém bons resultados. No entanto, eles não abordam nem transmissão de vídeo nem a construção de uma API vídeo.

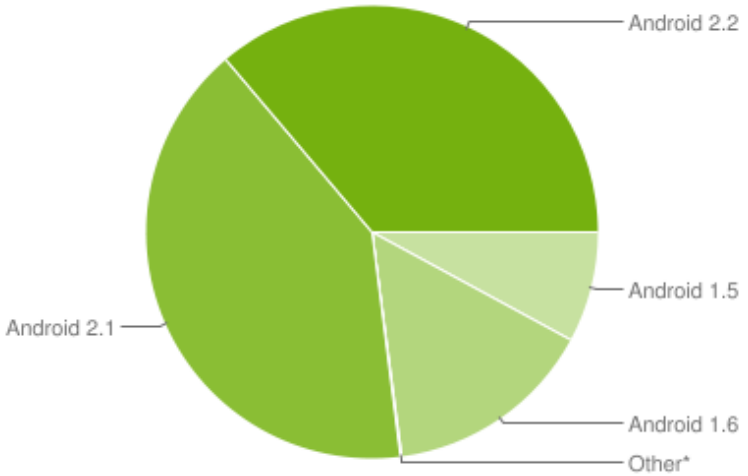
No que diz respeito a trabalhos não científicos, foi lançado, em outubro de 2010, o Adobe Air[2] para Android, o primeiro e único *framework* que apresenta alternativas às classes padrão encontrado. Graças ao suporte a Flash, introduzido na versão mais recente do sistema operacional (a 2.2), o *framework* da Adobe possibilita o desenvolvimento de aplicativos em Flash para Android, incluindo programas de vídeo interativo em tempo real. Com esse *framework*, aplicativos podem ser desenvolvidos em altíssimo nível de abstração (através de interface gráfica) sem necessidade de conhecimento de alguma linguagem de programação. Os programas de vídeo que vêm sendo desenvolvidos com o Adobe Air ainda estão em fase de testes e, portanto, não se

[1]FFmpeg é uma solução que pode ser utilizada como uma biblioteca para codificar e decodificar vídeo em *software* (FFMPEG, 2010).

[2]Adobe Air para Android: http://www.adobe.com/br/products/air

pode afirmar nada sobre a sua qualidade. Uma importante limitação é, contudo, a falta de suporte a versões anteriores à 2.2 do Android (ADOBE, 2010). Desse modo, 63,8% dos usuários de Android não seriam beneficiados, conforme indica o gráfico da Figura 1.2 (com dados de novembro de 2010).

Além do *framework* da Adobe, foram encontrados oito aplicativos de vídeo para Android com características relevantes para serem considerados como trabalhos relacionados a este. Quatro deles, (SIPDROID, 2011), (MOVICHA, 2010), (FRING, 2010) e (TANGO, 2010) são voltados para conversas por vídeo. Os outros quatro, (MOBILESOFT.KR, 2010), (WUZHENHUA PLAYER, 2010), (ARCMEDIA, 2010) e (REDIRECT INTELLIGENCE CORP, 2010) utilizaram alguma alternativa às classes padrão no seu desenvolvimento, visto que suportam formatos não compatíveis com as classes padrão. No entanto, todos são de código fechado e, portanto, não há como saber de que forma foram programados, com exceção do (SIPDROID, 2011), cuja estratégia será analisada no capítulo 3.2.



Platform	Distribution
Android 1.5	7.9%
Android 1.6	15.0%
Android 2.1	40.8%
Android 2.2	36.2%

Figura 1.2: Acessos ao Android Market[1] por versão (PLATFORMVERSIONS, 2010)

1.3 Conceitos Básicos Sobre Transmissão de Vídeo

As diferentes técnicas para transmissão de vídeo atuais podem ser classificadas em três grupos:

- Transmissão por *download* clássico: conhecida também como *download-and-play*, requer, basicamente, uma máquina cliente capaz de realizar o *download* completo de um arquivo de vídeo armazenado em um servidor *web* de mídia para, em seguida, renderizar o arquivo recebido com um reprodutor de mídia (TSCHÖKE, 2001)

- Transmissão por *download* progressivo: consiste em uma evolução da técnica de *download* clássico, e tem apenas uma necessidade adicional em relação à ela: um reprodutor de mídia que processe os dados que estão sendo baixados e os transforme em imagem e som enquanto o arquivo ainda não foi completamente transferido (NUNES, [200-])

- Transmissão por *streaming*: enquanto as técnicas por *download* utilizam TCP na camada de transporte e HTTP na de aplicação e transmitem os vídeos a uma taxa

[1]Android Market (http://www.android.com/market/) é a loja virtual de aplicativos para Android desenvolvida pela Google.

de transferência maximizada, a técnica de *streaming* caracteriza-se, principalmente, pelo uso de protocolos mais adequados para o envio de dados visando a sua renderização em tempo real, como o UDP (na camada de transporte) e o RTSP (na de aplicação), por exemplo, e por transmitir os vídeos com a mesma taxa com que eles foram codificados, entregando, desse modo, um fluxo contínuo à máquina final. Para isso, pode ser necessário ajustar a qualidade de codificação do vídeo (modificando a taxa de quadros por segundo, a resolução, ou outros atributos), de modo a permitir que máquinas com diferentes larguras de banda de rede recebam os dados a tempo de renderizá-los. Há, inclusive, um conjunto de estudos formais para se avaliar a qualidade do vídeo em sistemas desse tipo (DARONCO, 2009). Alguns dos fatores que mais influenciam na qualidade percebida são a resolução da imagem, a taxa de quadros por segundo e a sincronização entre o áudio e a imagem. A técnica de transmissão por *streaming* pode ser subdividida em (KUROSE; ROSS, 1999):

- *Streaming* de arquivo de vídeo armazenado: é caracterizado pela utilização de um servidor especializado em *streaming* que realiza a transmissão de vídeos a partir de arquivos de vídeo conforme a requisição de clientes (KUROSE; ROSS, 1999).
- Vídeo ao vivo não interativo: a principal diferença entre uma transmissão de vídeo ao vivo e um *streaming* de um arquivo de vídeo está na impossibilidade do usuário avançar ou retroceder a exibição quando o vídeo está sendo capturado e transmitido ao vivo. Por exemplo, é como assistir a um canal de televisão (KUROSE; ROSS, 1999).
- Vídeo interativo em tempo real: é o sistema em que pessoas se comunicam por áudio e vídeo. Nesses sistemas, o atraso na transmissão (que corresponde à diferença de tempo entre o instante da filmagem do quadro e o instante da exibição do quadro na máquina destino) também é um fator de qualidade importante (KUROSE; ROSS, 1999). Alguns milissegundos de atraso já causam incômodo, pois podem fazer que um participante interrompa outro, por exemplo, ao pensar que ele havia terminado por completo a sua fala após o que foi, na verdade, apenas uma pequena pausa na sua locução. Segundo (KUROSE; ROSS, 1999), atrasos menores que 150 milissegundos não são percebidos pelos participantes, atrasos entre 150 e 400 milissegundos são toleráveis e atrasos maiores que 400 milissegundos podem tornar a experiência frustrante. Um sistema de interação por vídeo em tempo real pode ser dividido em dois módulos: um módulo origem e um módulo destino. O módulo origem é composto de:

  1. um submódulo que captura áudio e vídeo não comprimidos do microfone e da câmera, respectivamente;
  2. um submódulo que codifica áudio e vídeo;
  3. um submódulo que envia pela rede os dados codificados;

  O módulo destino é composto de:

  1. um submódulo que recebe da rede dados de áudio e vídeo codificados;
  2. um submódulo que decodifica áudio e vídeo;

3. um submódulo que exibe os dados decodificados para o usuário.

Essa divisão em módulos e submódulos é ilustrada pela Figura 1.3.

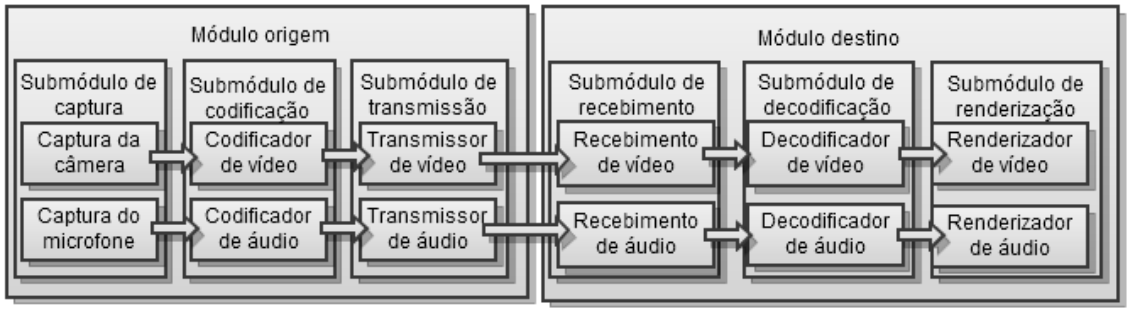


Figura 1.3: Módulos de um sistema de interação por vídeo em tempo real

2 DESENVOLVIMENTO DE PROGRAMAS DE VÍDEO PARA ANDROID

O Android é um sistema operacional de código aberto voltado principalmente para dispositivos móveis, desenvolvido numa colaboração entre Google e outros membros da Open Handset Alliance, e baseado no núcleo do Linux. Aplicativos para Android são escritos com a linguagem de programação Java e executam sobre a máquina virtual Dalvik. O desenvolvimento de aplicativos é feito com o Kit de Desenvolvimento de Software para Android (Android SDK). Esse *kit* fornece diversas ferramentas úteis (como depuradores de código, emuladores e APIs) além de oferecer plataformas que permitem a compilação de aplicativos (ABLESON; COLLINS; SEN, 2008).

É possível, também, desenvolver módulos de aplicativos Android nas linguagens C e C++, visto que o Android executa sobre o núcleo do Linux (FU et al., 2010). Para isso, é necessário utilizar a Native Development Kit (Kit de Desenvolvimento Nativo - NDK) para Android. Entretanto, não é possível executar um aplicativo totalmente desenvolvido utilizando a NDK - é necessário que, pelo menos, o ponto de partida do aplicativo seja escrito em Java. A interação entre o código Java e o código nativo é feita através da JNI[1] (Java Native Interface), interface que permite que um código que está executando em uma máquina virtual interaja com um código nativo (FU et al., 2010).

As classes de referência da SDK do Android que podem ser importantes para o desenvolvimento de um sistema de interação através de áudio e vídeo podem ser classificadas como segue:

- Para o módulo de origem: MediaRecorder, Camera e AudioRecord;
- Para o módulo de destino: MediaPlayer e AudioTrack.

Existem outras classes relacionadas, porém elas apresentam apenas pequenas extensões ou encapsulamentos das classes acima, sem importância para este trabalho. Também existem classes relacionadas a envio e recebimento de dados genéricos pela rede que podem ser utilizadas em conjunto com as classes acima.

A seguir, as cinco classes supracitadas serão detalhadas, e, com base nas características de cada uma, serão apresentadas, no capítulo 3, estratégias de desenvolvimento de um sistema de vídeo interativo que utilize apenas classes padrão do Android, bem como as limitações de cada estratégia. Ao final do capítulo são apresentadas duas tabelas que comparam as classes.

[1]Trabalhar com a JNI pode ser considerado uma tarefa complexa pois, além de requerer do programador o domínio de mais de uma linguagem de programação, suas funções são, em geral, pouco intuitivas. Outro exemplo de dificuldade é que a JNI não realiza o gerenciamento dos recursos de memória utilizados pela parte nativa do programa. Logo, o programador precisa ter mais atenção a esse aspecto visto que é de sua responsabilidade a liberação desses recursos (LIANG, 1999).

2.1 Classe MediaRecorder

A classe MediaRecorder realiza a captura de áudio e vídeo (através do microfone e da câmera, respectivamente) e codifica os dados utilizando o *hardware* disponível no aparelho. Os dados codificados podem ser salvos em arquivo ou transmitidos pela rede por meio de *sockets*[1]. A classe também permite a exibição de uma pré-visualização local da captura para o usuário. A utilização do MediaRecorder acontece da seguinte forma (HASHIMI; KOMATINENI, 2009):

```
1  //Declarar uma instância da classe:
2  MediaRecorder mMediaRecorder = new MediaRecorder();
3
4  //Escolher as fontes de áudio e vídeo:
5  mMediaRecorder.setAudioSource(<fonte_de_audio>);
6  mMediaRecorder.setVideoSource(<fonte_de_video>);
7
8  //Escolher o destino (arquivo ou socket)
9  mMediaRecorder.setOutputFile(<destino>);
10
11 //Iniciar a captura:
12 mMediaRecorder.prepare();
13 mMediaRecorder.start();
14
15 //Exibir o preview da captura:
16 mMediaRecorder.setPreviewDisplay(<superficie_local>);
```

Código básico de utilização da classe MediaRecorder

É importante ressaltar que, apesar do funcionamento da classe ser bastante simples e intuitivo, não existe nenhum mecanismo de acesso direto aos dados codificados. Essa característica impossibilita que os dados sejam encapsulados diretamente em um protocolo de aplicação antes de serem enviados. Para realizar essa tarefa usando a classe MediaRecorder, é necessário que um *socket* local seja aberto e intermedeie os dados entre a classe e o receptor local, que no caso poderia ser um encapsulador[2].

Além disso, a classe está totalmente vinculada aos codificadores em *hardware* disponíveis no dispositivo, que são poucos. Não tendo acesso aos dados decodificados, não é possível aplicar sobre os mesmos um processo de codificação diferente do padrão, inviabilizando o uso da classe para este fim.

2.2 Classe Camera

A classe Camera realiza a captura de vídeo (a partir da câmera) e permite, bem como a classe MediaRecorder, uma pré-visualização local da captura para o usuário. Ela pode ser utilizada de duas formas: para tirar fotografias (ou seja, capturar um quadro, codificá-lo e salvá-lo em arquivo) ou para configurar um *callback* que é executado a

[1]Na realidade, a classe MediaRecorder aceita apenas um descritor de arquivo como saída, e não *sockets*. E, em Java, *sockets* não têm descritores de arquivos associados (ao contrário de C). No entanto, foi descoberto um meio de fazer isso em Java: para poder utilizar um *socket* (local ou remoto), é necessário fazer uso da classe padrão ParcelFileDescritptor. Através do método fromSocket(Socket), é possível instanciar um ParcelFileDescriptor. Após instanciá-lo é possível acessar o descritor de arquivo do *socket* com o método getFileDescriptor() da classe ParcelFileDescriptor.

[2]Relembrando, a utilização de *sockets* com a classe MediaRecorder não é direta, pois não se pode obter um descritor de arquivo associado a um *socket* em Java diretamente.

cada quadro capturado, e que recebe dados não codificados. A utilização da classe acontece como segue (HASHIMI; KOMATINENI, 2009):

```
17 //Obter acesso à câmera:
18 Camera mCamera = Camera.open();
19
20 //Escolher a superfície de preview:
21 mCamera.setPreviewDisplay(<superficie_local>);
22
23 //Configurar o callback (opcional):
24 mCamera.setPreviewCallback(<contexto>);
25
26 //Iniciar a captura e o preview:
27 mCamera.startPreview();
28
29 //Tirar foto (opcional):
30 takePicture(<Camera.ShutterCallback>,
31             <Camera.PictureCallback_raw>,
32             <Camera.PictureCallback_jpeg>);
33
34 //Implementar o callback que recebe o quadro não codificado (opcional):
35 @Override
36 public void onPreviewFrame (byte[] data, Camera camera){}
```

Código básico de utilização da classe Camera

2.3 Classe AudioRecord

A classe AudioRecord realiza a captura de áudio (a partir do microfone), e os dados capturados não codificados são colocados em um *buffer* principal. Para utilizar os dados capturados pode-se fazer uma cópia do *buffer* principal para um *buffer* secundário. O funcionamento da classe AudioRecord segue as seguintes etapas (HASHIMI; KOMATINENI, 2009):

```
37 //Declarar uma instância da classe:
38 AudioRecord mAudioRecord = new AudioRecord(
39    MediaRecorder.AudioSource.MIC,
40    <taxa_de_amostragem>, <numero_de_canais>,
41    <bits_por_amostra>, <tamanho_do_buffer_principal>);
42
43 //Iniciar a captura:
44 mAudioRecord.startRecording();
45
46 //Transferir o conteúdo do buffer principal (quando desejado) para um buffer secundário:
47 record.read(<buffer_secundario>,
48    <indice_inicial_do_buffer_secundario>,
49    <numero_de_bytes_a_transferir>);
```

Código básico de utilização da classe AudioRecord

2.4 Classe MediaPlayer

A classe MediaPlayer fornece funções de alto nível para a renderização de vídeo e áudio. Permite a escolha de origem entre arquivo local, servidor *web* e *streaming* por RTSP. Configurada a origem do vídeo, a classe decodifica o áudio e o vídeo utilizando

os recursos de *hardware* disponíveis e exibe-os para o usuário (ABLESON; COLLINS; SEN, 2008). As etapas necessárias são as seguintes (HASHIMI; KOMATINENI, 2009):

```
50 //Declarar uma instância da classe:
51 MediaPlayer mMediaPlayer = new MediaPlayer();
52
53 //Escolher a origem do vídeo:
54  mMediaPlayer.setDataSource(<origem_do_video>);
55
56 //Iniciar o processo de leitura (local ou remota), decodificação e renderização:
57 mMediaPlayer.prepare();
58 mMediaPlayer.start();
```

Código básico de utilização da classe MediaPlayer

Outras funções importantes da classe são as opções para avançar, retroceder, pausar e parar o vídeo. Tais funções só têm efeito em três casos:

- Quando o vídeo é de um arquivo local
- Quando o vídeo é de um arquivo armazenado em um servidor *web* HTTP, visto que a classe dá suporte a *download* progressivo
- Quando o vídeo é de um *streaming* de um arquivo de vídeo armazenado em um servidor de *streaming* RTSP

O quarto e último uso suportado pela classe (porém sem as opções de avançar, retroceder e pausar) é a recepção de vídeo ao vivo com o protocolo RTSP. As funcionalidades restantes da classe são de menor importância, como modificar o volume, obter a duração e a resolução do vídeo, etc (MEDIAPLAYER, 2010).

A simplicidade e a facilidade de uso, juntamente com a invocação do *hardware* para executar a tarefa computacionalmente custosa de decodificação, tornam a classe MediaPlayer ótima para o desenvolvimento de aplicativos básicos de vídeo (HASHIMI; KOMATINENI, 2009). No entanto, quando se trata de desenvolver aplicativos diferenciados e complexos, como aplicativos de interação por vídeo em tempo real, ou aplicativos que dão suporte a uma maior quantidade de *codecs* e formatos, a classe se torna muito limitada, e a necessidade de estratégias e/ou *frameworks* alternativos fica evidente (SONG et al., 2010).

2.4.1 Limitações da Classe MediaPlayer

As principais limitações da classe MediaPlayer são:

- Impossibilidade de decodificar vídeos codificados com *codecs* que não são suportados pelo *hardware* do dispositivo (SONG et al., 2010), visto que a classe não suporta decodificação em *software*
- Para vídeos não locais, há suporte apenas aos protocolos HTTP e RTSP (HASHIMI; KOMATINENI, 2009). Não é possível utilizar outro protocolo (seja ele de um padrão existente ou não)
- Impossibilidade de acessar, via programação, os dados dos quadros que vão sendo decodificados. Logo, o programador não pode desenvolver suas próprias funções para trabalhar com os quadros, ficando limitado às funções básicas

citadas. Assim, todos os aplicativos de vídeo ficam parecidos e com pouca flexibilidade, mudando apenas nas partes não relacionadas diretamente à decodificação e à exibição de vídeos

- Atraso na transmissão de vídeo ao vivo, o que tornaria frustrante uma aplicação de interação em tempo real. Tal afirmação foi comprovada através do seguinte experimento:

  1. Uma câmera foi conectada a um PC, tendo seu conteúdo capturado através do *software* VideoLAN Media Player (VLC)[1];
  2. O fluxo de dados proveniente do VLC foi direcionado para um servidor de *streaming* RTSP (Darwin Streaming Server[2]) especificado por um arquivo SDP (Session Description Protocol);
  3. O fluxo RTSP foi recebido em um *smartphone* Motorola Milestone (Android 2.0.1) utilizando a classe MediaPlayer.
  4. O atraso médio obtido durante o experimento com o Motorola foi de 10 segundos. Ao receber o mesmo fluxo em um VLC instalado em outro PC da mesma rede, o atraso foi de aproximadamente 1 segundo. Além do experimento proposto, também foi realizada uma comparação do RTSP do Android com o RTSP do VLC utilizando diversos links RTSP públicos na Internet. O mesmo resultado anterior foi verificado.

2.5 Classe AudioTrack

A classe AudioTrack serve para renderizar áudio, e funciona de maneira análoga à classe AudioRecord - para renderizar uma amostra de áudio, deve-se transferi-la de um *buffer* secundário para o *buffer* principal da classe. As etapas necessárias são as seguintes (HASHIMI; KOMATINENI, 2009):

```
59 //Declarar uma instância da classe:
60 mAudioTrack = new AudioTrack(<tipo_de_stream>,
61    <taxa_de_amostragem>,
62    <numero_de_canais>,
63    <bits_por_amostra>,
64    <tamanho_do_buffer_principal>,
65    <modo_(estatico_ou_stream)>);
66
67 //Caso o modo utilizado seja o stream, chamar o método play():
68  mAudioTrack.play();
69 //Nada será renderizado ainda pois o buffer principal está vazio.
70
71 //Transferir o conteúdo do buffer secundário (quando desejado) para o buffer principal:
72 mAudioTrack.write(<buffer_secundario>,
73    <indice_inicial_do_buffer_secundario>,
74    <numero_de_bytes_a_transferir>);
75
76 //Caso o modo utilizado seja stream, cada novo dado recebido no buffer principal será
77 reproduzido sem a necessidade de chamar o método play();
78 //Caso o modo utilizado seja estático, chamar o método play() para renderizar os dados
79 (quando desejado):
```

[1]VLC: http://www.videolan.org/vlc/

[2]Darwin Streaming Server: http://dss.macosforge.org/

```
80 mAudioTrack.play();
```

Código básico de utilização da classe AudioTrack

Tabela 2.1: Comparação entre as classes relacionadas ao módulo de origem

Classe	Captura	Acesso aos dados	Codificação	Transmissão
MediaRecorder	Áudio e vídeo	Indireto	Em *hardware*	Por *socket*
Camera	Apenas vídeo	Direto	Apenas para JPEG	Não transmite
AudioRecord	Apenas áudio	Direto	Não codifica	Não transmite

Tabela 2.2: Comparação entre as classes relacionadas ao módulo de destino

Classe	Renderização	Entrada	Decodificação
MediaPlayer	Áudio e vídeo	Arquivo local, HTTP e RTSP. Não aceita dados decodificados como entrada	Em *hardware*
AudioTrack	Apenas áudio	Array com dados PCM decodificados	Não decodifica

3 ESTRATÉGIA PARA VÍDEO COM AS CLASSES PADRÃO

Neste capítulo será realizada uma comparação de possíveis estratégias para desenvolvimento de sistemas de interação por vídeo em tempo real com a utilização apenas das classes padrão do Android. Com isso, serão analisadas as limitações dessas estratégias e das APIs do Android.

3.1 Estratégia 1: de (JANSSEN, 2010)

É a técnica implementada em (JANSSEN, 2010).

3.1.1 Na Origem

Essa estratégia se baseia na classe MediaRecorder para realizar captura de áudio e vídeo, e salva pequenas amostras (de 2 segundos, por exemplo) do conteúdo capturado codificado em arquivos temporários. Durante a captura os dados do arquivo vão sendo enviados ao ponto remoto através de uma classe do Android para transmissão de dados genéricos. Ao finalizar a gravação da captura no arquivo temporário, a classe MediaRecorder insere no início do arquivo um cabeçalho e metadados do contêiner escolhido (3GPP ou MPEG-4). Esses dados, por sua vez, também são enviados para o ponto remoto, fechando o ciclo e reiniciando o processo.

3.1.2 No Destino

O ponto remoto, por sua vez, recebe os dados de áudio e vídeo provenientes da origem e armazena-os em um arquivo temporário. Quando o cabeçalho e os metadados chegam, o ponto remoto finaliza o arquivo, e nesse instante é possível reproduzir o seu conteúdo através da classe MediaPlayer. O novo arquivo temporário deve ter nome diferente do anterior, pois o início do recebimento acontecerá antes do término da reprodução do arquivo recebido.

3.1.3 Análise da Estratégia

Segundo o autor, a classe MediaRecorder leva em média 1,3 segundos para ser inicializada, e não é possível inicializar um novo MediaRecorder antes de finalizar o anterior. Logo, cada 2 segundos de conteúdo capturado será seguido de 1,3 segundos sem conteúdo. No destino, a classe MediaPlayer leva em média 0,83 segundos para ser inicializada, o que também representaria um intervalo de tempo sem conteúdo. Entretanto, os tempos de inicialização não se somam como atraso, mas se sobrepõem, como mostra a figura 3.1.

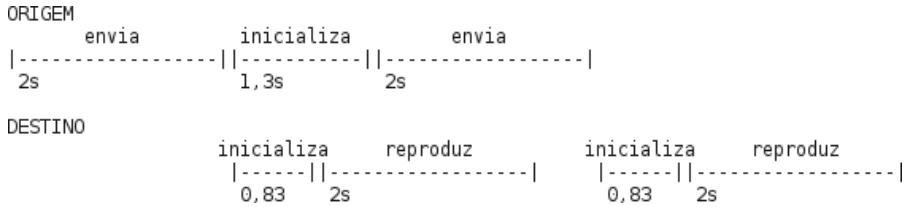


Figura 3.1: Estratégia 1: de (JANSSEN, 2010) - Linha do tempo

Uma análise rápida revela um atraso mínimo de 2,83 segundos, que é o tempo de captura do arquivo temporário na origem somado ao tempo de inicialização do MediaPlayer no destino. Isso sem contar os tempos necessários para a classe MediaRecorder inserir os metadados no início do arquivo na origem, para os metadados serem inseridos no início do arquivo no destino e para transmitir um pacote pela rede. Mas o maior problema dessa estratégia são as constantes pausas seguidas de saltos na exibição do vídeo no destino, devido ao período de inicialização da classe MediaRecorder, quando não se pode capturar nada. O autor reconhece que a técnica não serve para interação em tempo real, mas afirma que é o melhor resultado possível utilizando apenas as classes padrão do Android.

3.2 Estratégia 2: de (SIPDROID, 2011)

É a técnica implementada em (SIPDROID, 2011). Esta técnica utiliza a capacidade da classe MediaPlayer em reproduzir conteúdo ao vivo utilizando o protocolo RTSP.

3.2.1 Na Origem

A classe MediaRecorder é utilizada para capturar vídeo. Os dados codificados são colocados diretamente em um *socket* local, que por sua vez é lido por um outro módulo que implementa o protocolo RTSP. Dessa forma os dados são enviados para o destino.

3.2.2 No Destino

A classe MediaPlayer é inicializada e configurada para receber o conteúdo de um fluxo RTSP.

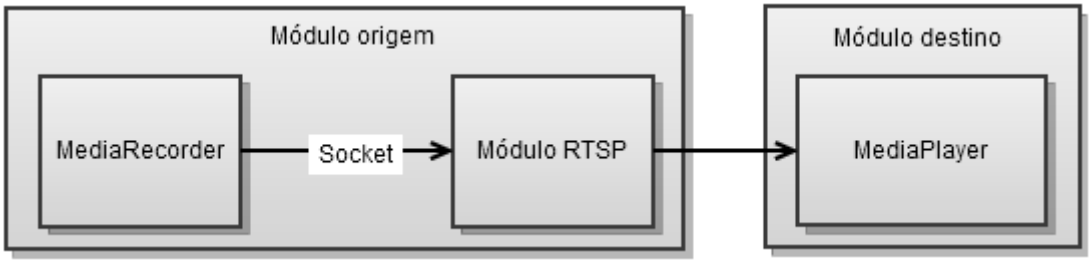


Figura 3.2: Esquema ilustrativo da estratégia 2: de (SIPDROID, 2011)

3.2.3 Análise da Estratégia

O método resulta em um sistema de vídeo ao vivo, mas não em um sistema de interação em tempo real, pois a implementação do RTSP no Android não serve para tempo real, conforme explicado no capítulo 2.4.1. Comparado ao método anterior, nesta estratégia não ocorrem paradas nem saltos na exibição. Porém, segundo os testes de desempenho realizados com a classe MediaPlayer, o atraso é maior do que no método anterior, devido ao baixo desempenho do RTSP utilizado juntamente com a classe

MediaPlayer do Android. Vale deixar claro que o problema encontra-se na implementação de RTSP do Android e não na implementação do Sipdroid.

3.3 Estratégia 3: Camera e AudioRecord

Ao invés de utilizar a classe MediaRecorder para realizar a captura de áudio e vídeo, utiliza-se as classes Camera e AudioRecord. Ambas as classes permitem acesso direto aos dados não codificados. No entanto, não há nenhuma classe padrão do Android que codifique um fragmento de áudio ou vídeo a partir de dados não codificados, e quadros não codificados são grandes demais para serem transmitidos pela rede em tempo viável para vídeo interativo.

Poderia ser implementado um codificador/decodificador em Java. No entanto, codificação e decodificação são processos que exigem um uso intensivo de CPU, ou seja, possuem um alto custo computacional, o que torna a solução inviável, uma vez que a implementação executaria sobre uma máquina virtual em um dispositivo com baixo poder computacional.

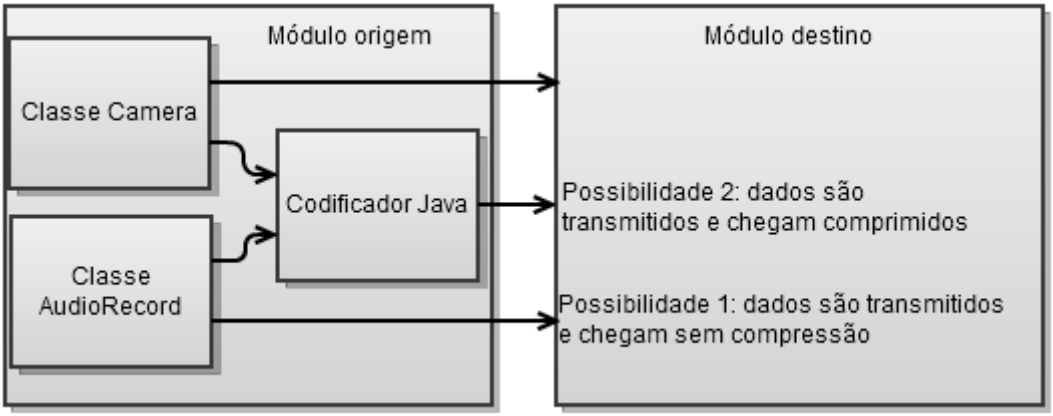


Figura 3.3: Estratégia 3: Camera e AudioRecord

3.4 Estratégia 4: Transmissão Consecutiva de Fotografias JPEG

3.4.1 Na Origem

Utilizar a função de foto da classe Camera, que retorna um quadro codificado em JPEG, e enviar cada quadro independentemente pela rede para o destino.

3.4.2 No Destino

Receber cada quadro e exibi-lo com alguma classe padrão do Android que decodifique e exiba imagens JPEG.

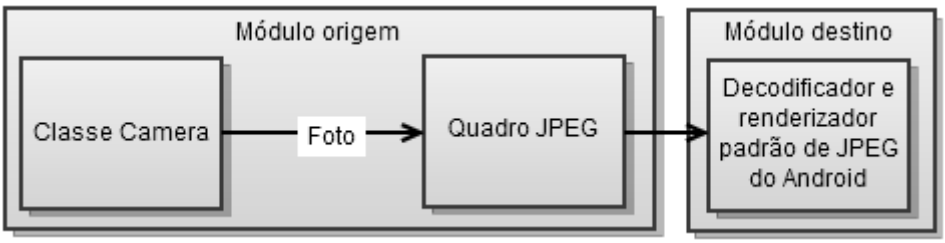


Figura 3.4: Estratégia 4: Transmissão consecutiva de fotografias JPEG

3.4.3 Análise da Estratégia

Codecs de vídeo são tipicamente mais eficazes do que *codecs* de imagens, visto que um vídeo é um conjunto de imagens relacionadas entre si, e a compressão leva em consideração a relação entre essas imagens. O *codec* JPEG apenas comprime uma imagem sem relacioná-la com nenhuma outra, portanto tem menos poder de compressão. Além disso, a função de foto do Android não foi projetada para este uso e é pouco eficiente. Um dos motivos é que quando a função foto é chamada, a pré-visualização do vídeo capturado é interrompida. Para tirar uma nova foto é preciso reiniciar a pré-visualização.

O método em si não foi testado pois, conforme justificado anteriormente, a estratégia é inviável independentemente da implementação. Outro problema da estratégia é não ter um correspondente para o áudio - somente o vídeo seria transmitido.

3.5 Estratégia 5: Reuso das Bibliotecas Nativas Privadas

Como o Android tem código aberto, esta estratégia propõe copiar as bibliotecas nativas (C/C++) internas do sistema operacional que interagem com as classes MediaPlayer e MediaRecorder e com o *hardware* dos aparelhos, inserindo-as em um novo projeto compilável através da NDK. Desse modo se teria acesso aos quadros codificados e decodificados pelo *hardware* e uma classe de transmissão de dados genéricos do Android poderia ser utilizada para realizar a transmissão e recepção dos dados multimídia. A implementação poderia ser organizada da seguinte forma:

- Captura: câmera e microfone seriam acessados diretamente pelo código nativo, utilizando as bibliotecas internas do Android;
- Codificação e decodificação: em *hardware*, utilizando as bibliotecas internas do Android;
- Transmissão e recepção: alguma classe padrão do Android para transmissão e recepção de dados genéricos;
- Renderização: tela e alto falante seriam acessados diretamente pelo código nativo, utilizando as bibliotecas internas do Android.

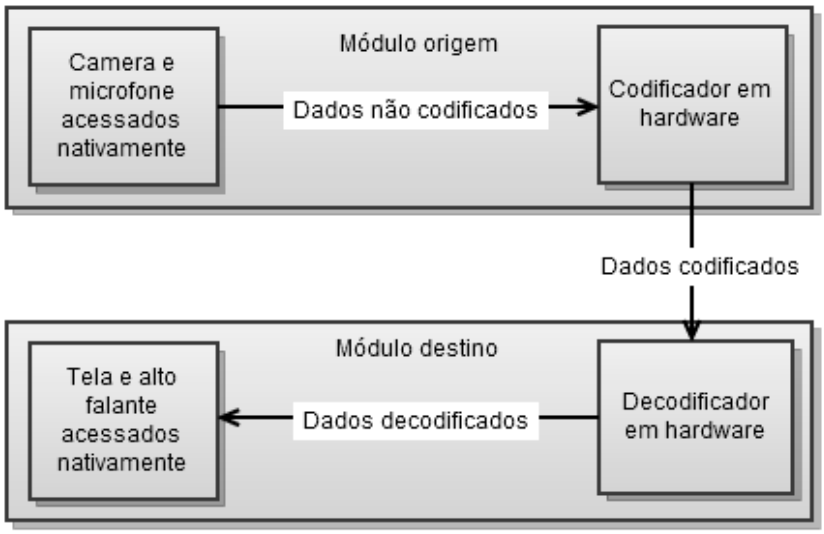


Figura 3.5: Estratégia 5: Reuso das bibliotecas nativas privadas

Esta seria a solução mais eficiente possível, visto que todas as tarefas com alto custo computacional seriam realizadas em *hardware*, e os dados trafegariam pela rede comprimidos. Contudo, as bibliotecas nativas são dependentes de plataforma, ou seja, cada dispositivo tem particularidades de *hardware* que são generalizadas através da implementação de suas bibliotecas nativas. Utilizando a estratégia descrita, seria necessário manter diversas versões da mesma biblioteca, uma para cada dispositivo (e uma para cada versão do Android também, visto que as bibliotecas nativas mudam a cada nova versão do Android). Logo, a manutenção de um aplicativo desenvolvido com esta estratégia seria complexa. Por esse motivo, a manipulação de bibliotecas nativas internas do Android é desencorajado pelos desenvolvedores do Android. Além disso, essa solução é limitada à utilização dos *codecs* suportados por padrão pelos dispositivos.

4 SOLUÇÃO ADOTADA

Pela análise acima, pôde-se concluir que, até o momento, não é possível construir um sistema de interação por vídeo em tempo real utilizando apenas as classes padrão do Android. A solução proposta neste trabalho utiliza algumas das classes padrão do Android apresentadas, mas se baseia principalmente em implementações alternativas, e é compatível com todas as versões do Android utilizadas atualmente (ou seja, versões maiores ou iguais a 1.5), pois todos os métodos das classes padrão utilizados na técnica são pertencentes ao nível de API da versão 1.5 ou menor.

Neste capítulo, essa estratégia será apresentada de forma teórica. Adicionalmente, para representar as explicações textuais e demonstrar algumas das dificuldades encontradas, os trechos teóricos serão intercalados, neste capítulo, por alguns dos trechos de código real que foram utilizados, na prática, para implementar a técnica. Por fim, é apresentada a Figura 4.1, que ilustra a estratégia como um todo, e a tabela 4.1, que compara a solução proposta neste capítulo com as estratégias apresentadas no capítulo 3.

4.1 Módulo de Captura de Vídeo

A classe Camera é utilizada para realizar a captura de vídeo através da configuração de um *callback* que recebe os quadros não codificados. Através dessa classe é possível configurar os parâmetros desejados, tais como resolução e número de quadros por segundo. Os quadros não codificados são apresentados no formato YUV420SP, também conhecido como NV21, único formato de captura suportado por todos os dispositivos Android. Esta solução propõe passar os quadros não codificados para a interface nativa (em C++), através da JNI, para a realização das tarefas de conversão e codificação.

4.1.1 Implementação do Módulo de Captura de Vídeo

Na prática, a utilização da classe Camera teve algumas peculiaridades. Por exemplo, há dispositivos com apenas uma câmera e há outros com duas câmeras. Nesse último caso, é preferível que a câmera selecionada seja a frontal, pois é a mais adequada para interação por vídeo. Percebi, também, a necessidade de tratar um caso específico: em dispositivos Android da marca HTC com câmera frontal, o procedimento para requisitar a utilização da câmera é diferente do caso genérico. O trecho de código Java a seguir demonstra os passos necessários para utilizar a câmera em todos os casos:

```
81 //abre a câmera frontal, caso haja uma. Caso contrário, abre a câmera padrão. Observe que há
82 um tratamento para um caso específico no dispositivo da marca HTC.
83 boolean  isAvailableSprintFFC;
```

```
try {
   Class.forName("android.hardware.HtcFrontFacingCamera");
   isAvailableSprintFFC = true;
} catch (Exception ex) {
    isAvailableSprintFFC = false;
}
if (isAvailableSprintFFC) { //caso específico do HTC
   try {
      Method method =
Class.forName("android.hardware.HtcFrontFacingCamera").getDeclaredMethod("getCamera",
null);
      mCamera = (Camera) method.invoke(null, null);
   } catch (Exception ex) {
      ...
   }
} else { //caso genérico
   mCamera = Camera.open();
   Camera.Parameters parameters = mCamera.getParameters();
   parameters.set("camera-id", 2); //o número 2 abre a câmera frontal, se houver. Não havendo,
abre a câmera padrão.
   mCamera.setParameters(parameters);
}
```

Código Java para requisitar o uso da câmera

Outra descoberta interessante relacionada à utilização da câmera tem a ver com a configuração do *callback* que recebe os dados não codificados. O *callback* pode ser configurado tanto com o método setPreviewCallback quanto com o método setPreviewCallbackWithBuffer. O método setPreviewCallbackWithBuffer, teoricamente, só funciona nas versões do Android superiores ou iguais a 2.2. No entanto, com a utilização de Java Reflection, pode-se chamar esse método em versões mais antigas. Essa descoberta é de grande importância pois esse método, em comparação com o método setPreviewCallback (que é compatível com todas as versões diretamente), reduz pela metade o custo computacional da operação, permitindo assim o dobro da taxa de quadros por segundo. Isso ocorre pois, no método setPreviewCallback, um novo buffer é alocado a cada quadro capturado, causando que o Garbage Collector interrompa a execução do aplicativo a cada quadro, enquanto que com o método setPreviewCallbackWithBuffer, o mesmo *buffer* é utilizado. Além disso, para utilizar o método setPreviewCallbackWithBuffer, é necessário indicar, inicialmente, dois *buffers* com o método addCallbackBuffer (pode-se indicar apenas um, mas percebeu-se um ganho de desempenho indicando dois). O código Java abaixo ilustra essa inicialização:

```
byte[] buffer = new byte[<largura*altura*4>]; //tamanho máximo de um quadro não
codificado
mCamera.addCallbackBuffer(buffer);
buffer = new byte[buffer];
mCamera.addCallbackBuffer(buffer);
mCamera.setPreviewCallbackWithBuffer(this);
```

Código Java para inicializar o método setPreviewCallbackWithBuffer

Para utilizar uma interface nativa em C/C++, é necessário, em Java, declarar o cabeçalho dos métodos nativos que serão implementados em C/C++ e, ainda em Java, carregar os binários correspondentes ao código nativo compilado. No caso desta estratégia, queremos um método C++ que receba do Java o quadro não codificado. Isso é feito da seguinte forma (do lado Java):

```
112 private native void enqueueFrame(byte[] dados); //declaração do cabeçalho da função C++
113 System.load(<caminho para o binário>); //função que carrega o código nativo compilado
```

Código Java para inicializar a utilização de método nativo

O *callback* Java que recebe os dados não codificados e os passa à função nativa "enqueueFrame" pode ser implementado da seguinte forma:

```
114 @Override public void onPreviewFrame (byte[] dados, Camera mCamera) {
115    enqueueFrame(dados); //isto chama a função C++ "enqueueFrame"
116    mCamera.addCallbackBuffer(dados); //é necessário indicar um buffer para o próximo
117 callback
118 }
```

Código Java de *callback* que invoca o método nativo passando o *buffer*

Do lado nativo (C++), é necessário receber os dados passados como parâmetro. Isso pode ser feito como segue (em C++):

```
119 //A função enqueueFrame, em C++, precisa ser declarada de uma forma especial, para ser
120 encontrada corretamente. Abaixo, <package Java> e <classe Java> correspondem ao package
121 e à classe Java nos quais o cabeçalho da função enqueueFrame foi declarado. Observe,
122 também, que há dois parâmetros adicionais (além dos dados): "env" e "obj". Ambos são
123 obrigatórios (passados automaticamente do Java para o código nativo) e correspondem a
124 variáveis auxiliares da JNI para se interagir entre os códigos nativo e Java.
125 jvoid Java_<package Java>_<classe Java>_enqueueFrame(JNIEnv *env, jobject obj,
126 jbyteArray dados) {
127    //O método GetByteArrayElements pertence à JNI e converte os dados vindos do Java para
128 serem utilizados no lado nativo
129    jbyte *javaData = env->GetByteArrayElements(dados, 0);
130   //neste ponto, "javaData" contém, em C++, os dados não codificados do quadro capturado, e
131 está pronto para ser tratado conforme será explicado no capítulo 4.3
132   ...
133 }
```

Código C++ que recebe os dados passados como parâmetro pelo Java

4.2 Módulo de Captura de Áudio

A captura de áudio é feita através da classe AudioRecord. As amostras de áudio são capturadas no formato não codificado PCM, e enviadas para a interface em C++, através da JNI, para a realização das tarefas de conversão e codificação.

4.2.1 Implementação do Módulo de Captura de Áudio

A implementação deste módulo é análoga à do módulo de captura de vídeo, inclusive na parte de passar o quadro não codificado do Java ao C++. Desta vez, porém, não foram encontradas peculiaridades importantes. Vale citar que, para capturar áudio e vídeo ao mesmo tempo com esta solução, cada captura deve ocorrer numa *thread* diferente.

4.3 Módulo de Codificação e de Transmissão de Vídeo

Como Android executa sobre um núcleo do Linux, foi possível gerar uma compilação do FFmpeg através da NDK[1]. O FFmpeg é um projeto que implementa diversos codificadores e decodificadores em *software*, escrito na linguagem C, e pequenas adaptações possibilitam sua compilação para o Android. Dessa forma aproveita-se o desempenho melhor do código nativo, em comparação com o código Java, para executar operações com alto custo computacional. Entretanto, o FFmpeg exige, para diversos *codecs*, que o quadro de vídeo cru de entrada esteja no formato YUV420P, diferentemente do formato de saída da classe Camera. A conversão intermediária necessária entre formatos pode ser realizada em *software* através do SwScale, que é parte do FFmpeg.

Portanto, a estratégia consiste de converter o quadro de YUV420SP para YUV420P e codificá-lo com o FFmpeg antes de transmiti-lo pela rede[2]. O quadro codificado com o *codec* desejado é, então, enviado para o ponto remoto. Isso pode ser feito tanto através do código nativo, via Berkeley Sockets, quanto através do código Java, via Java Sockets. Entretanto, a segunda opção é mais custosa, uma vez que os dados devem voltar pro contexto Java via JNI. Antes do envio, ainda, é possível encapsular os dados no protocolo desejado (como RTP ou RTMP, por exemplo).

4.3.1 Implementação do Módulo de Codificação/Transmissão de Vídeo

Na prática, a implementação deste módulo consiste em invocar as funções de conversão e de codificação do FFmpeg passando como parâmetro o array “javaData” (apresentado no capítulo 4.1), que corresponde ao quadro não codificado proveniente da câmera. Após a codificação, ele pode ser encapsulado no protocolo desejado, pode ser dividido em pacotes menores, ou pode receber qualquer tratamento antes de ser transmitido pela rede ao destino.

4.4 Módulo de Codificação e de Transmissão de Áudio

Novamente, este módulo é análogo ao do vídeo. No caso do áudio, não é necessário convertê-lo antes de codificá-lo, pois o formato de saída do microfone é aceito pelo FFmpeg.

Quanto à transmissão, os dados de áudio podem ser enviados segundo um protocolo diferente do escolhido para o vídeo. É recomendada, nesse caso, a transmissão de variáveis que indicam o instante de tempo de cada quadro, para facilitar a sincronização dos quadros de áudio com os de vídeo no lado do destino.

4.5 Módulo de Recebimento e de Decodificação de Áudio e de Vídeo

No ponto remoto (o lado do destino), o processo inverso é aplicado. Primeiramente os dados de áudio e vídeo devem ser recebidos através de *sockets*, que podem estar tanto no contexto Java quanto no contexto de código nativo. Se os dados estiverem encapsulados em algum protocolo, o processo inverso deve ser aplicado a fim de isolar

[1]Uma versão do FFmpeg configurado para compilação no Android encontra-se pública em: git@github.com:mconf/android-ffmpeg.git

[2]YUV420 é uma forma específica de representar digitalmente o espaço de cor YUV. YUV420P é uma versão planar desta representação, na qual todos os componentes Y aparecem no início da matriz, seguidos por todos os componentes U e, por fim, seguidos pelos componentes V. YUV420SP é uma versão semiplanar - ou entrelaçada - na qual os componentes Y, U e V aparecem misturados.

os dados de cada quadro de vídeo ou amostra de áudio. Os dados são então decodificados com o FFmpeg compilado para código nativo.

4.6 Módulo de Renderização de Áudio

Neste ponto da estratégia, o módulo destino já tem acesso aos dados decodificados do áudio em código nativo, teoricamente prontos para serem renderizados. Entretanto, não há como acessar o *hardware* de som de um dispositivo Android diretamente do código nativo. A solução encontrada é passar os quadros de áudio decodificados para a classe AudioTrack utilizando a JNI. Desse modo, eles são reproduzidos, pois classe AudioTrack toca quadros de áudio que são entregues a ela no formato não codificado PCM.

4.6.1 Implementação do Módulo de Renderização de Áudio

A troca de contexto dos dados entre código nativo e código Java é feita através da JNI. Porém, na prática, ao passar um *buffer* como parâmetro a um método Java invocado a partir do contexto nativo, ou seja, utilizando a JNI diretamente, o Garbage Collector do Android é executado a cada quadro, causando uma enorme queda no desempenho (100ms de interrupção, em média). Alternativamente, foi utilizada uma outra técnica da JNI que apresentou melhor desempenho. A técnica consiste na criação de um *array* em Java que representa um quadro, e de um *array* de mesmo tamanho em código nativo. No código nativo invoca-se funções da JNI que associam os dois *arrays* à mesma posição de memória. Dessa forma, cada atualização do *array* no código nativo com novos dados decodificados é propagada ao *array* do Java, que reproduz os dados de áudio conforme citado anteriormente. Assim, sem cópias ou alocação de memória adicional, a solução mostrou-se a mais eficiente para a realização da tarefa proposta. Os trechos de código seguintes ilustram a implementação deste módulo.

Começando pela declaração, em Java, do *array* e de uma função que irá associá-lo ao *array* do C++:

```
134 byte[] mAudioBuffer = new byte[<tamanho>]; //o tamanho depende dos parâmetros de
135 codificação utilizados
136
137 byte[] getBuffer() {
138    return mAudioBuffer;
139 }
```

Código Java de inicialização do *array* de áudio

Em seguida, o trecho do C++ que associa o *array* do C++ e o do Java à mesma posição de memória:

```
140 //na chamada a seguir, “env” e “obj” são variáveis auxiliares, fornecidas automaticamente
141 pela JNI, que facilitam a interação entre o código nativo e o Java. GetMethodID é uma função
142 da JNI que retorna um identificador de um método Java encontrado de acordo com os
143 parâmetros passados. “getBuffer é o nome do método buscado e “()[B” indica que esse
144 método não pede parâmetros e retorna um “byte[]”
145 jmethodID JavaGetBuffer = env->GetMethodID(obj, "getBuffer", "()[B");
146
147 //em seguida, o método Java “getBuffer” é chamado a partir do C++, e retorna o array
148 mAudioBuffer declarado na linha 134 acima
149 jbyteArray audioBufferJNI = (_jbyteArray*)env->CallObjectMethod(obj, JavaGetBuffer);
```

```
150
151 //finalmente, para poder associar um array do C++ à mesma posição de memória do array do
152 Java, é preciso fazer o seguinte:
153 int8_t * audioBuffer = (int8_t*)jniEnv->GetByteArrayElements(audioBufferJNI, NULL);
154 //neste ponto, o array "audioBuffer" do C++ está associado ao array "mAudioBuffer" do Java.
155 Portanto, qualquer modificação feita no array do C++ será propagada no do Java diretamente,
156 ou seja, sem cópia
```

Código C++ de associação do *array* do C++ com o do Java

No código Java a seguir, é implementada uma função Java que, conforme a explicação da classe AudioTrack (do capítulo 2.5), reproduz os dados de áudio passados a ela, graças ao método "write" da classe AudioTrack. Nesta estratégia, tal função será chamada a partir do C++.

```
157 void renderiza(){
158   //a variável mAudioBuffer abaixo é a mesma que foi declarada na linha 134 e que está
159 associada ao array do C++
160   mAudioTrack.write(mAudioBuffer, 0, mAudioBuffer.length);
161 }
```

Código Java que invoca o método "write" responsável por renderizar o áudio

Por fim, o trecho de código C++ a seguir chama o método Java "renderiza" (apresentado acima) sempre que o *array* "audioBuffer" do lado do C++ receber um novo quadro decodificado entregue pelo FFmpeg, para que o quadro possa ser renderizado. Observe que não é necessário passar o *array* como parâmetro, pois o *array* do Java já conterá os dados do *array* do C++.

```
162 env->CallIntMethod(obj, JavaRenderiza);
163 //Obs: "JavaRenderiza" é o identificador do método Java "renderiza", e deve ser obtido de
164 forma análoga à inicialização do identificador "JavaGetBuffer" apresentada anteriormente
```

Código C++ que invoca o método Java "renderiza" para renderizar um quadro de áudio

Assim, o ciclo de renderização do áudio está completamente explicado. Uma observação final: é recomendada a atribuição de alta prioridade à *thread* de renderização do áudio apresentada neste capítulo, para diminuir as chances de falhas ou pequenas pausas na renderização, que seriam desagradáveis ao ouvido humano e poderiam atrapalhar uma comunicação - mais do que pequenas pausas no vídeo atrapalhariam.

4.7 Módulo de Renderização de Vídeo

Assim como para o áudio, neste ponto da estratégia o módulo destino já tem acesso aos dados decodificados do vídeo em código nativo, teoricamente prontos para serem renderizados. Entretanto, de maneira similar ao áudio, não há como acessar o *hardware* da tela diretamente do código nativo para renderizar os quadros de vídeo.

Neste caso, contudo, não há uma classe do Android que receba dados de vídeo não codificados e renderize-os (como faz a AudioTrack para o áudio). Portanto, a solução proposta no parágrafo anterior (de passar o *array* de áudio para o Java) não se aplica para o vídeo.

A solução encontrada é utilizar o OpenGL ES[1]. Através do OpenGL ES, constrói-se, em código nativo, um retângulo com as dimensões desejadas e aplica-se uma textura ao retângulo contendo o quadro de vídeo, que pode, desse modo, ser exibido na tela.

[1]O OpenGL ES é a versão do OpenGL para sistemas embarcados.

4.7.1 Implementação do Módulo de Renderização de Vídeo

No entanto, na prática, para que a textura seja exibida na tela, é necessário que a função de renderização nativa seja chamada de dentro do método onDrawFrame implementado pela classe padrão GLSurfaceView.Renderer (do lado Java) e retorne para o método onDrawFrame a cada quadro. O trecho de código a seguir ilustra isso. Primeiramente do lado Java:

```
//cabeçalho da função de renderização nativa:
private native void renderizadorNativo();

//método onDrawFrame invocando a função nativa C++ para renderizar um quadro e retornar
ao Java.
@Override public void onDrawFrame(GL10 gl) {
   //convém limpar o buffer do OpenGL ES a cada quadro com o método padrão abaixo:
   gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

   renderizadorNativo();
}
```

Código Java contendo o método onDrawFrame

Agora, do lado C++:

```
jvoid Java_<package Java>_<classe Java>_renderizadorNativo(JNIEnv *env, jobject obj) {
   //este é o ponto indicado para obter um novo quadro decodificado entregue pelo FFmpeg
   ….
Vamos supor que, agora, o array "quadroDecodificado" abaixo já obteve os dados de um
quadro de vídeo decodificado pelo FFmpeg. Logo, basta apenas chamar as duas funções do
OpenGL ES a seguir, e retornar ao método onDrawFrame do Java, que o quadro será
renderizado (os parâmetros e as duas funções do OpenGL ES serão explicados adiante):
   glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0,
                        <largura original>,<altura original>, GL_RGB,
                       GL_UNSIGNED_SHORT_5_6_5,  quadroDecodificado);

   glDrawTexiOES(<posicao X>, <posicao Y>, 0, <largura a desenhar>, <altura a desenhar>);

   return;
}

//Obs: o OpenGL ES e a textura precisam ser inicializados de acordo com a sua documentação
antes de se utilizar as duas funções acima
```

Código C++ que renderiza um quadro nativamente com o OpenGL ES

O OpenGL ES só permite aplicação de texturas RGB (por isso a utilização do parâmetro GL_RGB acima), e o FFmpeg, por sua vez, só decodifica quadros de vídeo para o formato YUV420P, sendo necessária uma conversão intermediária antes de renderizá-los – conversão que pode ser feita com o próprio FFmpeg. No entanto, o OpenGL ES permite a utilização de diferentes tipos de RGB. O que mostrou melhor desempenho é o RGB565, por necessitar de menos memória para representar um quadro decodificado em comparação aos outros tipos de RGB.

Os parâmetros <largura original> e <altura original> correspondem ao tamanho verdadeiro do quadro de vídeo. Já os parâmetros <largura a desenhar> e <altura a desenhar> podem ser diferentes dos tamanhos originais, e servem para ampliar ou reduzir a renderização do quadro. Ou seja, utiliza-se o próprio OpenGL ES para a

realização do redimensionamento. Isso porque foi verificado experimentalmente que o OpenGL ES tem melhor desempenho que o FFmpeg para realizar tal operação. Esse desempenho é ainda maior nas versões do OpenGL ES posteriores à 1.0, nas quais o redimensionamento é feito em hardware.

Pode-se perguntar por que foi utilizada a função glTexSubImage2D ao invés da glTexImage2D. Isso se deve a uma limitação do OpenGL ES: a limitação de, em algumas versões, a função glTexImage2D somente suportar aplicação de texturas sobre retângulos de dimensões potência de dois. Ou seja, para desenhar um vídeo com resolução 320x240 com a glTexImage2D, deve-se utilizar um *array* de tamanho necessário para representar um quadro de pelo menos 512x256 *pixels*, o que causaria uma queda desnecessária no desempenho. Para superar essa limitação, foi elaborada a seguinte solução:

1. Antes de renderizar o primeiro quadro, e apenas uma vez: criar um retângulo com dimensões potência de dois imediatamente maiores que o tamanho do vídeo, com a chamada abaixo:

```
194  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
195          <potência de 2 da largura>, <potência de 2 da altura>, 0, GL_RGB,
196          GL_UNSIGNED_SHORT_5_6_5,
197          <buffer vazio de dimensões potência de 2>);
```

Código C++ que cria um retângulo com dimensões potência de dois em OpenGL ES

2. Então, sempre que se desejar renderizar um quadro, aplicar a textura com as dimensões originais do vídeo sobre parte dessa região com a função glTexSubImage2D. Isso foi demonstrado no código das linhas 176 à 193.

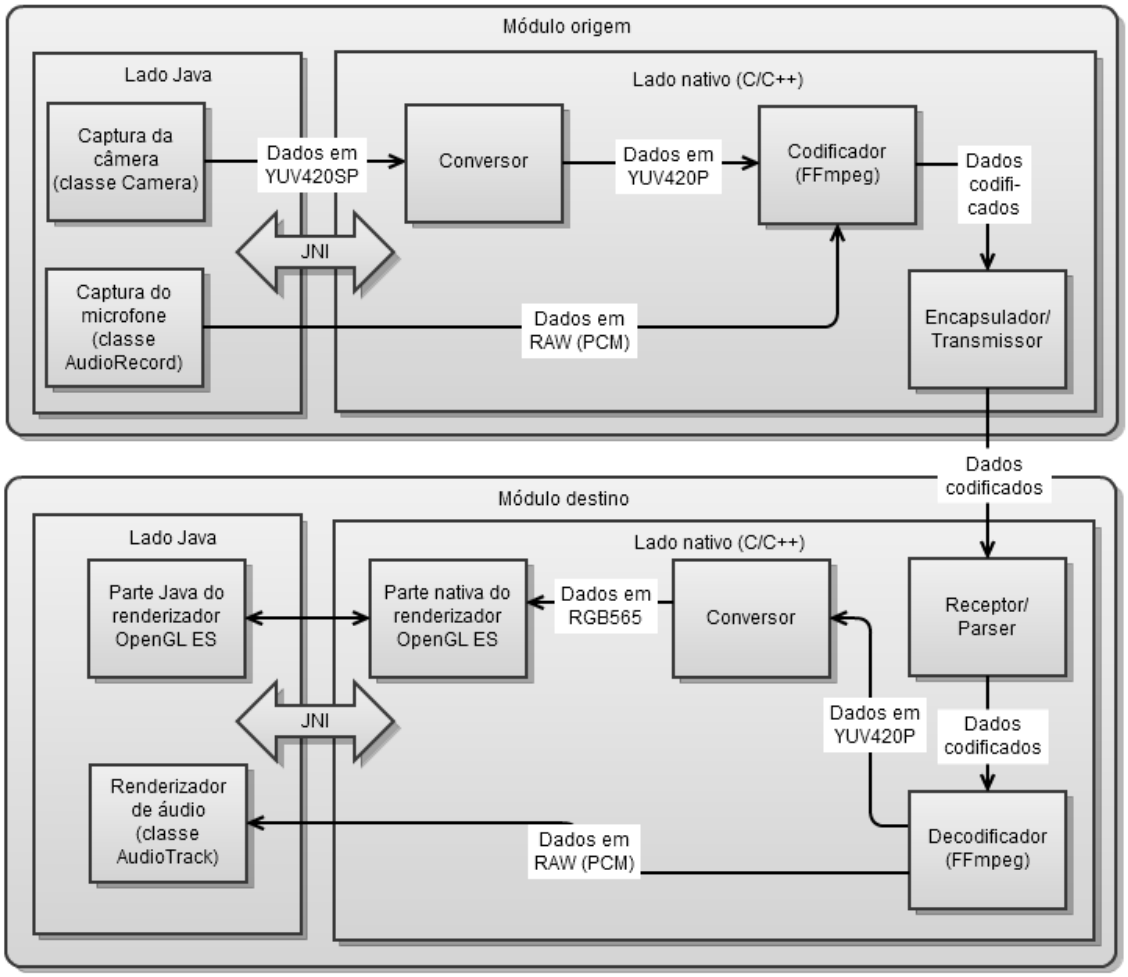


Figura 4.1: Esquema ilustrativo da estratégia proposta neste trabalho

Tabela 4.1: Comparação entre as estratégias do capítulo 3 e a solução proposta

Estratégia	Fluidez na renderização	Tráfego na rede	Atraso	Complexidade de implementação	Manutenção
1	Muito ruim	Baixo	Alto	Média	Muito simples
2	Boa	Baixo	Muito alto	Média	Muito simples
3 (sem codificar)	Muito ruim	Muito alto	Não verificado	Muito simples	Muito simples
3 (com codificador Java)	Muito ruim	Baixo	Não verificado	Muito simples (desconsiderando o codificador)	Muito simples
4	Muito ruim	Alto	Não verificado	Muito simples	Muito simples
5	Muito boa	Baixo	Muito baixo	Muito complexa	Muito complexa
Solução apresentada neste trabalho	Boa	Baixo	Baixo	Complexa	Muito simples

5 EXPERIMENTOS REALIZADOS E RESULTADOS OBTIDOS

A técnica apresentada neste trabalho foi implementada na forma de três aplicativos independentes e na forma de uma API. As implementações (também chamadas de validações, pois foram úteis na validação da estratégia) serão descritas neste capítulo, e seus respectivos resultados serão apresentados.

Tabela 5.1: Especificação dos dispositivos utilizados nos experimentos

Modelo	Processador	Versão do Android	Resolução máxima	Versão do OpenGL ES
HTC Magic	528MHz	1.6	320x480	1.1
Motorola Milestone A853	500MHz	2.0.1	480x854	1.0
Samsung Galaxy S GTI9000B	1000MHz	2.1-update1	480x800	1.1

Após a elaboração da estratégia, mas antes de realizar as implementações, foi realizado um experimento para verificar a capacidade máxima de decodificação e de renderização que um dispositivo móvel com Android pode alcançar através da técnica proposta. O experimento consistiu de um aplicativo para Android que abre e renderiza arquivos locais de áudio e/ou vídeo codificados com qualquer codec e encapsulados com qualquer formato suportado pelo FFmpeg, utilizando-se a técnica proposta neste trabalho. Utilizei um dispositivo Android real HTC Magic e um arquivo de vídeo local com resolução 176x144. A taxa de decodificação alcançada foi de 227 quadros por segundo, ou seja, tempo médio por quadro de 4,41ms. A taxa de renderização obtida foi de 60 quadros por segundo, limitada pela taxa de *refresh* da tela (60Hz). Portanto, o resultado foi positivo e serviu para validar a ideia, conduzindo às implementações abaixo.

5.1 Validação para Interação por Vídeo em Tempo Real

Foi implementado um aplicativo de conversa por vídeo em tempo real para dispositivos móveis Android aplicando a técnica apresentada.

5.1.1 Resultados Obtidos

Para verificar experimentalmente o funcionamento do aplicativo, foi realizada uma conversa por vídeo em tempo real utilizando dois dispositivos Android reais: um Motorola Milestone e um Samsung Galaxy S, ambos conectados à Internet por Internet sem fio. Os parâmetros de codificação de vídeo utilizados no teste foram:

- Resolução: 320x240
- *Codec*: MPEG-4
- Taxa de bits: 256kbps
- GOP: 12
- Taxa de quadros por segundo: 15

Os parâmetros de codificação de áudio utilizados no teste foram:

- *Codec*: MP2
- Taxa de bits: 64kbps
- Frequência: 22050Hz
- Bits por amostra: 16
- Canais: 1

O vídeo foi ampliado localmente com o OpenGL ES para a resolução máxima de cada aparelho (480x854 no Motorola e 480x800 no Samsung). A tabela 5.2 exibe os resultados do teste.

Tabela 5.2: Resultados da validação 1 – Teste 1

Dispositivo	Tipo de medida	Taxa de quadros	Tempo médio por quadro
Motorola	Captura do vídeo	15	1,16ms
Samsung	Captura do vídeo	15	0,72ms
Motorola	Renderização do vídeo	15	16,03ms
Samsung	Renderização do vídeo	14	2,88ms

Enquanto a captura do vídeo teve desempenho semelhante nos dois dispositivos, a renderização do vídeo teve um melhor desempenho no Samsung Galaxy S. A diferença de performance é justificada pela versão do OpenGL ES do Samsung (1.1) em comparação à do Motorola (1.0) - conforme citado anteriormente, a partir da versão 1.1 o redimensionamento é realizado em *hardware*, enquanto nas versões anteriores tal tarefa é realizada em *software*.

Pela análise da tabela também pode-se perceber que a taxa de quadros capturados na origem no Samsung (quinze por segundo) foi mantida no destino no Motorola e ficou próxima do ideal no inverso (do Motorola para o Samsung). O motivo é o menor poder de processamento do Motorola, que não foi capaz de codificar e transmitir quinze quadros em um segundo (deve-se levar em conta que ele também estava decodificando ao mesmo tempo). Desse modo, não chegaram quinze quadros ao destino (o Samsung), por isso os “quatorze quadros” da tabela 5.2.

Adicionalmente, com o mesmo aplicativo, realizou-se uma transmissão de vídeo em tempo real do Samsung para o Motorola unilateralmente. Os parâmetros de codificação do áudio e do vídeo utilizados foram os mesmos do teste anterior, porém com a taxa de quadros do vídeo igual a trinta por segundo. A tabela 5.3 exibe os resultados do teste.

Tabela 5.3: Resultados da validação 1 – Teste 2

Dispositivo	Tipo de medida	Taxa de quadros	Tempo médio por quadro
Samsung	Captura do vídeo	30	0,58ms
Motorola	Renderização do vídeo	29	12,77ms

Pode-se perceber, nos dois testes, a perda de um quadro, já que foram exibidos, no teste 1, no Samsung, quatorze quadros ao invés de quinze, e, no segundo teste, 29 quadros ao invés de trinta. Esse comportamento já era esperado, visto que os parâmetros foram escolhidos de modo a testar a estratégia no seu limite. Logo, os parâmetros poderiam ser reduzidos consideravelmente e, ainda assim, fornecer uma boa qualidade de vídeo.

Portanto, os testes realizados demonstraram a total viabilidade do método para sistemas de vídeo interativo em tempo real.

5.2 Validação para Ensino a Distância

O IVA é um sistema de videoconferência (compatível com PC) voltado a ensino a distância com suporte à transmissão de áudio e vídeo em alta qualidade (ROESLER, 2009). Utilizando a estratégia proposta neste trabalho, foi implementado um aplicativo para Android (ao qual foi dado o nome de EAD@Cel) que permite a interação de usuários a partir de dispositivos móveis por áudio e vídeo com a videoconferência do IVA.

As entidades que compõem o IVA são descritas a seguir, para esclarecer como o EAD@Cel foi integrado ao modelo. A Figura 5.1 ilustra essa integração:

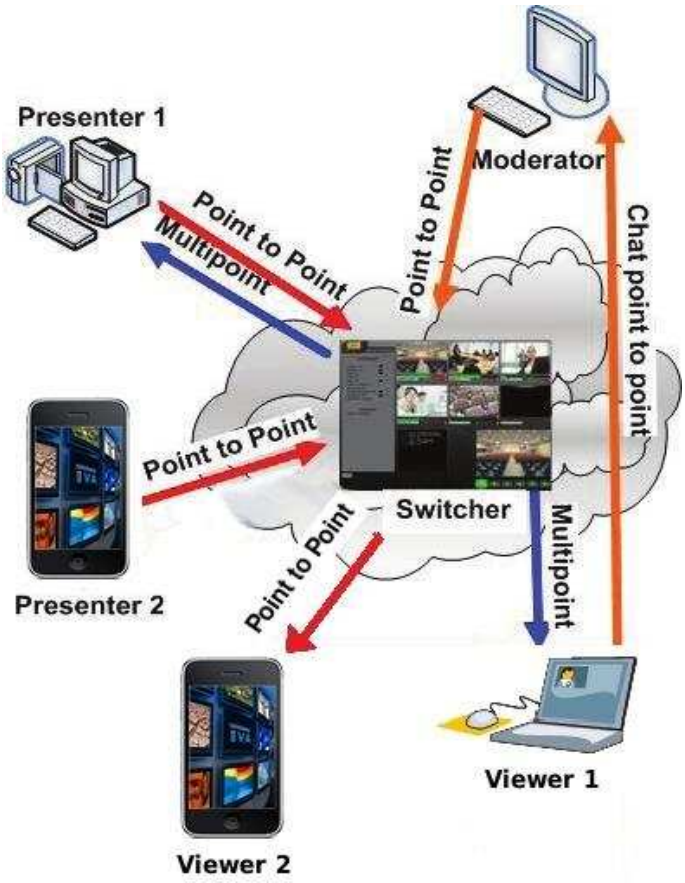


Figura 5.1: Entidades do IVA e o EAD@Cel integrado ao sistema

- Apresentador: esta entidade captura o áudio e o vídeo do professor ou palestrante, transmitindo o sinal à entidade Suíte por *unicast*, também chamada conexão ponto a ponto.

- Suíte: utilizada para gerenciar o fluxo de até seis Apresentadores simultâneos. O seu operador decide qual dos fluxos deve ser transmitido em cada momento. A Suíte é exibida na Figura 5.2 e representada na Figura 5.1 (como Switcher) recebendo o fluxo dos Apresentadores em *unicast* e transmitindo o fluxo escolhido em *multicast* ou conexão multi ponto.

- Visualizador: recebe, decodifica e exibe o fluxo multimídia *multicast*. É similar ao Apresentador, porém sem a possibilidade de transmitir áudio e vídeo.

- Moderador: recebe perguntas de texto dos Apresentadores e dos Visualizadores, responde algumas delas, filtra as mais importantes e, de acordo com o contexto da aula ou palestra, interage com a Suíte ou com o Apresentador para publicar as questões ao vivo.

O EAD@Cel foi integrado ao IVA agindo de dois modos: como um Apresentador ou como um Visualizador (representados na Figura 5.1 como Presenter 2 e Viewer 2). Uma conexão ponto a ponto é utilizada nesse caso devido à falta de suporte a *multicast* em dispositivos móveis.

Figura 5.2: IVA (Entidade Suíte)

5.2.1 Resultados Obtidos

O IVA utiliza um protocolo de comunicação próprio que age sobre UDP e TCP. Como já foi explicado anteriormente, a técnica apresentada neste trabalho permite a utilização de qualquer protocolo de comunicação. Logo, a implementação do EAD@Cel provou essa capacidade e a consequente possibilidade de integração com sistemas existentes para PC. As bibliotecas de implementação dos protocolos de rede foram escritas em C++.

Para medir o desempenho da implementação, o aplicativo Android (EAD@Cel), agindo como Visualizador, recebeu um fluxo de dados codificado em MPEG-4 (vídeo) e MP2 (áudio) gerado a partir de um PC e distribuído para os participantes remotos pelo IVA. Os parâmetros de codificação do vídeo foram: taxa de bits de 1400kbps, resolução de 720x480 e taxa de quadros de 15 por segundo. O áudio foi codificado a uma taxa de 128kbps. O Motorola Milestone executando o aplicativo recebeu e exibiu esta transmissão mantendo a taxa de 15 quadros por segundo, com atraso levemente superior à exibição no PC. A Figura 5.3 mostra o teste em execução.

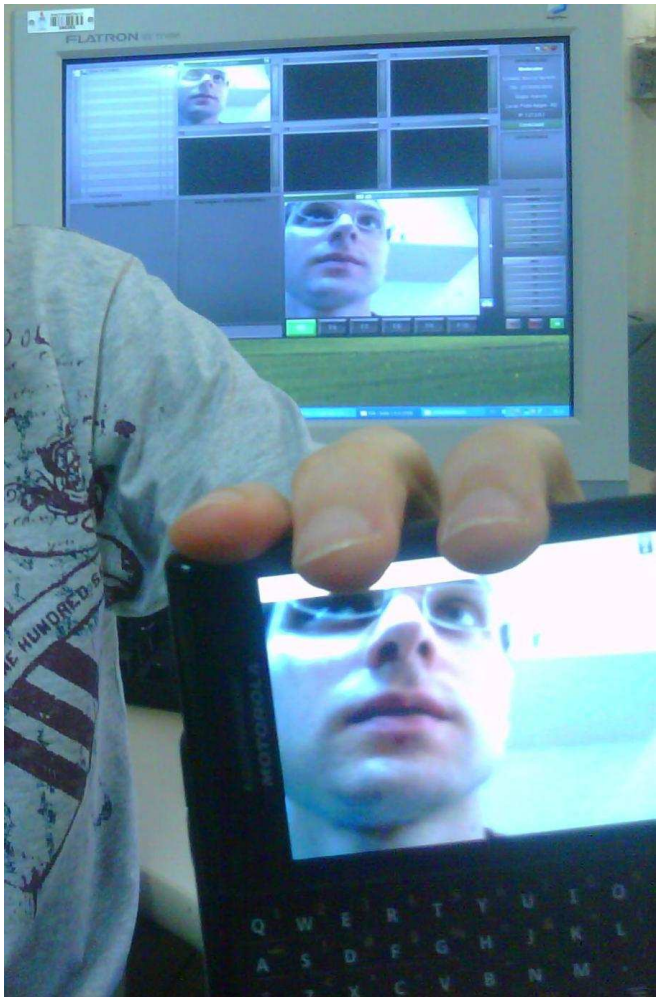


Figura 5.3: EAD@Cel (à frente) integrado ao sistema IVA (ao fundo)

5.3 Validação para Videoconferência

A estratégia também foi integrada ao sistema de videoconferência na *web* chamado BigBlueButton[1]. O BigBlueButton é um sistema de código aberto que permite a interação por vídeo e áudio, bem como conversas por texto, compartilhamento de tela e apresentações. O sistema utiliza a tecnologia Flash para transmissão de conteúdo, e o protocolo utilizado para interação entre clientes e servidor é o RTMP. Os *codecs* utilizados pelo sistema são o Sorenson H.263 para vídeo e o μ-law para áudio. As resoluções possíveis vão de 160x120 até 1280x720, sendo o padrão 320x240.

O aplicativo de integração com o BigBlueButton foi implementado no âmbito do projeto GT-Mconf[2]. Apesar da tecnologia Flash ser suportada em dispositivos Android com versão 2.2 ou posterior, a decisão do projeto foi implementar um aplicativo nativo para Android, compatível com as versões mais antigas.

5.3.1 Resultados Obtidos

A figura 5.4 apresenta um vídeo sendo exibido no aplicativo Android, proveniente da videoconferência do BigBlueButton. Mais uma vez aproveitou-se a capacidade da estratégia de suportar qualquer protocolo. Desta vez, uma biblioteca de implementação

[1]BigBlueButton: http://bigbluebutton.org/
[2]GT-Mconf: http://www.inf.ufrgs.br/prav/projetos_gtmconf.php

do protocolo RTMP, escrita em Java, foi utilizada para a comunicação com o servidor, e tanto áudio quando vídeo foram recebidos no contexto do Java e enviados ao contexto de código nativo através da JNI.

Figura 5.4: Aplicativo Android integrado ao sistema *web* BigBlueButton

5.4 Validação para API ou Framework

Esta implementação consiste de uma API que encapsula todas as partes complexas da técnica apresentada neste trabalho. A API permite implementar aplicativos de interação por vídeo em tempo real para Android de modo simples e rápido.

A API consiste, basicamente, de seis classes Java, quinze módulos C/C++ compilados com a NDK do Android como bibliotecas binárias do tipo *shared object* (extensão .so) e dois elementos de *layout* do Android para permitir a fácil inclusão de superfícies para exibir o vídeo que está sendo codificado e o que está sendo decodificado.

A API permite, basicamente, quatro operações, que podem ser iniciadas em qualquer parte do aplicativo (o programador usuário da API é livre para escolher). As quatro operações são:

- Iniciar uma captura, codificação e transmissão de vídeo
- Iniciar uma captura, codificação e transmissão de áudio

- Iniciar um recebimento, decodificação e renderização de vídeo
- Iniciar um recebimento, decodificação e renderização de áudio

O usuário programador pode escolher em qual dos elementos de *layout* padrão do Android ele quer exibir a renderização do vídeo e/ou a pré-visualização da captura. Por exemplo, caso ele deseje exibir uma imagem grande correspondente ao vídeo do usuário remoto sobreposta por uma pequena imagem correspondente à pré-visualização do usuário local, uma boa opção é utilizar uma Activity para o vídeo do usuário remoto e um Dialog para o local.

5.4.1 Instalação e Utilização da API

A instalação da API dá-se através da simples inclusão das seis classes Java a um projeto qualquer para Android, da adição dos binários .so à pasta “libs/armeabi” do projeto e à inclusão dos dois elementos de *layout* na pasta de *layouts* do projeto.

Caso o usuário programador queira utilizar transmissão ou recepção de áudio ou vídeo no seu aplicativo, ele deverá incluir a seguinte linha no seu “AndroidManifest.xml”:

<uses-permission android:name="android.permission.INTERNET" />

Caso ele queira utilizar captura de câmera no seu aplicativo, deverá incluir a seguinte linha no seu “AndroidManifest.xml”:

<uses-permission android:name="android.permission.CAMERA" />

Caso queira utilizar captura de áudio no seu aplicativo, deverá incluir a seguinte linha no seu “AndroidManifest.xml”:

<uses-permission android:name="android.permission.RECORD_AUDIO" />

Para iniciar uma captura, codificação e envio de vídeo, é necessário apenas escrever o seguinte código:

```
198 //Em seu elemento de layout desejado, adicione as seguintes linhas:
199 setContentView(R.layout.video_capture);
200 VideoCapture videoWindow = (VideoCapture) findViewById(R.id.video_capture);
201
202 //Caso você deseje exibir um preview pequeno e manter a taxa de aspecto original, adicione a
203 seguinte linha (o parâmetro margem é um booleano que indica se o preview será exibido com
204 margem ou não):
205 videoWindow.smallPreview(<margem>);
206
207 //Caso você deseje utilizar parâmetros de captura diferentes do padrão, adicione a seguinte
208 linha:
209 videoWindow.setCaptureParams(<taxa de frames>,<largura>,<altura>);
210
211 //Caso você deseje utilizar parâmetros de codificação diferentes do padrão, adicione a seguinte
212 linha:
213 videoWindow.setEncodeParams(<codec_id>,<bitrate>,<gop>);
214
215 //Caso você deseje enviar para um ip diferente do padrão, adicione a seguinte linha:
216 videoWindow.setIP(<ipByte1>,<ipByte2>,<ipByte3>,<ipByte4>,<porta>);
217
```

```
//Para efetivamente iniciar a captura, a codificação e o envio, inicie a execução seu elemento de layout.

//Para finalizar a captura, a codificação e o envio, finalize o seu elemento de layout ou adicione a seguinte linha:
videoWindow.stopCapture();
```

Código para iniciar uma captura, codificação e envio de vídeo

Para iniciar uma captura, codificação e envio de áudio, é necessário apenas escrever o seguinte código:

```
AudioCapture mAudioCapture;
mAudioCapture = new AudioCapture();

//Caso você deseje utilizar parâmetros de captura diferentes do padrão, adicione a seguinte linha:
mAudioCapture.setCaptureParams(<frequencia>,<canais>,<bits por amostra>);

//Caso você deseje utilizar parâmetros de codificação diferentes do padrão, adicione a seguinte linha:
mAudioCapture.setEncodeParams(<codec_id>,<bitrate>);

//Caso você deseje enviar para um ip diferente do padrão, adicione a seguinte linha:
mAudioCapture.setIP(<ipByte1>,<ipByte2>,<ipByte3>,<ipByte4>,<porta>);

//Para efetivamente iniciar a captura, a codificação e o envio:
mAudioCapture.start();

//Para finalizar a captura, a codificação e o envio, adicione a seguinte linha:
mAudioCapture.stopCapture();
```

Código para iniciar uma captura, codificação e envio de áudio

Para iniciar um recebimento, decodificação e renderização de vídeo, é necessário apenas escrever o seguinte código:

```
//Em seu elemento de layout desejado, adicione as seguintes linhas:
setContentView(R.layout.video_window);
VideoSurface videoWindow = (VideoSurface) findViewById(R.id.video_window);

//Caso você deseje utilizar um codec diferente do padrão, adicione a seguinte linha:
videoWindow.setCodec(<codec_id>);

//Caso você deseje receber de um ip diferente do padrão, adicione a seguinte linha:
videoWindow.setIP(<ipByte1>,<ipByte2>,<ipByte3>,<ipByte4>,<porta>);

//Para efetivamente iniciar o recebimento, a decodificação e a exibição, adicione a seguinte linha (o parâmetro margem é um booleano que indica se o preview será exibido com margem ou não):
videoWindow.start(<margem>);

//Para finalizar o recebimento, a decodificação e a renderização, adicione a seguinte linha:
videoWindow.stop();
```

Código para iniciar um recebimento, decodificação e renderização de vídeo

Para iniciar um recebimento, decodificação e renderização de áudio, é necessário apenas escrever o seguinte código:

```
//Em seu elemento de layout desejado, adicione as seguintes linhas:
AudioRenderer mAudioRenderer;
mAudioRenderer = new AudioRenderer();

//Caso você deseje utilizar um codec diferente do padrão, adicione a seguinte linha:
mAudioRenderer.setCodec(<codec_id>);

//Caso você deseje receber de um ip diferente do padrão, adicione a seguinte linha:
mAudioRenderer.setIP(<ipByte1>,<ipByte2>,<ipByte3>,<ipByte4>,<porta>);

//Para efetivamente iniciar o recebimento, a decodificação e a renderização, adicione a seguinte linha:
mAudioRenderer.start();

//Para controlar o volume, utilize os seguintes métodos:
mAudioRenderer.getVolume();
mAudioRenderer.volumeDown();
mAudioRenderer.volumeUp();

//Para finalizar o recebimento, a decodificação e a renderização, adicione a seguinte linha:
mAudioRenderer.deinitAudio();
```

Código para iniciar um recebimento, decodificação e renderização de áudio

5.4.2 Resultados Obtidos

Para testar a implementação da API, foi implementado, com ela, um aplicativo de conversa por vídeo objetivando ser o mais semelhante possível ao aplicativo apresentado no capítulo 5.1 (teste 1), antes implementado sem a API. Os resultados foram praticamente idênticos, validando a implementação da API.

Adicionalmente, o pequeno número de linhas de código necessárias para tal aplicativo (graças à utilização da API), representa a sua facilidade de uso.

No capítulo 6.1 (Trabalhos Futuros), será apresentada uma possível extensão da API na forma de um *framework* que serviria para dar mais flexibilidade ao usuário programador.

6 CONCLUSÃO

Este trabalho apresentou uma solução para implementação de sistemas de vídeo interativo em tempo real para Android, sendo consistente com a versão 1.5 do Android ou posterior. Os aspectos inovadores da estratégia são, principalmente, a codificação/decodificação de áudio e vídeo em *software* (com o FFmpeg compilado para o Android), a utilização da JNI para manipular os dados de áudio e de vídeo entre a camada nativa e a camada Java e a utilização de OpenGL ES para renderizar o vídeo. Todas essas escolhas são justificadas através de um estudo sobre as classes padrão do Android e suas limitações, seguida de uma comparação de estratégias propostas por trabalhos anteriores e seus resultados. Os testes das diferentes implementações da solução são apresentados, e seus resultados validam a utilização da técnica na implementação desse tipo de sistema.

Um aspecto positivo adicional da técnica é a flexibilidade dada ao desenvolvedor e ao sistema. A solução não envolve a utilização de um protocolo de transmissão específico ou de um *codec* - qualquer biblioteca de transmissão, escrita tanto em C/C++ quanto em Java, pode ser utilizada no sistema, e o mesmo não fica limitado aos *codecs* oferecidos pelo *hardware* do dispositivo. Além disso, ao contrário dos trabalhos mencionados, a estratégia proposta não apenas realiza codificação, decodificação e transmissão de áudio e vídeo, mas também faz essa transmissão em tempo real, permitindo uma comunicação interativa por vídeo.

6.1 Trabalhos Futuros

A estratégia apresentada neste trabalho permitiu a implementação da API apresentada no capítulo 5.4. Essa API (que está disponível em http://www.inf.ufrgs.br/prav/downloads_api_android.html) pode ser estendida na forma de um *framework* (ou uma API mais completa), oferecendo flexibilidade ao programador com relação a aspectos como, por exemplo, a escolha da forma de transmissão/recepção dos dados codificados. Por exemplo, o programador, se desejar, poderá implementar seu próprio protocolo para transmissão, utilizar um protocolo padrão já existente, ou utilizar a transmissão fornecida pelo *framework*. Também poderá utilizar o *framework* para arquivos de vídeo locais que estão codificados com *codecs* ou formatos não suportados em *hardware* pelo Android, visto que o FFmpeg suporta, em *software*, um número muito grande de *codecs*.

Desse modo, além das quatro utilizações possíveis da API apresentadas no capítulo anterior, o *framework* poderia acrescentar outras funcionalidades, como:

- Iniciar uma captura de áudio e/ou vídeo, exibir uma pré-visualização, enviar os quadros capturados do Java ao C++ e dar acesso aos dados dos quadros não codificados ao programador em C++ ou em Java. Isso poderá ser útil caso o

programador usuário não queira transmitir os dados, mas apenas realizar um processamento sobre eles ou codificá-los utilizando sua própria implementação de *codec*.

- Iniciar uma captura de áudio e/ou vídeo, exibir uma pré-visualização, enviar os quadros capturados do Java ao C++, codificar os quadros e dar acesso aos dados dos quadros codificados ao programador em C++ ou em Java. Isso pode ser útil caso o programador queira utilizar o seu próprio protocolo de transmissão, bastando encapsular os dados utilizando o protocolo que ele implementar. Nesse caso o programador deverá ser responsável pela sincronização do áudio com o vídeo, mas o *framework* deve fornecer um *timestamp* dos dados.

- Reproduzir um arquivo local, e não um *stream*. Isso poderá ser útil para reproduzir arquivos codificados com *codecs* não suportados pelo *hardware* do aparelho, visto que a decodificação do FFmpeg é realizada em *software*.

- Iniciar um recebimento de áudio e/ou vídeo utilizando um método ou protocolo de recebimento implementado pelo usuário programador. Isso será responsabilidade do programador, que provavelmente utilizará essa técnica em conjunto com a do segundo item acima. Após receber os dados codificados utilizando a sua implementação específica, o programador pode, se quiser, passá-los ao decodificador do *framework* (que posteriormente poderá chamar o renderizador), contanto que os dados estejam de acordo com o suportado pelo decodificador.

- Iniciar um recebimento de áudio e/ou vídeo utilizando o método de recebimento implementado pelo *framework* e dar acesso aos quadros codificados ao programador em C/C++ ou em Java. Isso poderá ser útil caso o programador não queira renderizar os dados, mas apenas realizar um processamento sobre eles ou decodificá-los utilizando sua própria implementação de *codec*.

- Iniciar um recebimento de áudio e/ou vídeo utilizando o método de recebimento implementado pelo *framework*, decodificar os dados e dar acesso aos quadros decodificados ao programador em C/C++ ou em Java. Isso poderá ser útil caso o programador não queira renderizar os dados, mas apenas realizar um processamento sobre eles.

- Apenas renderizar quadro(s) de áudio e/ou de vídeo decodificado(s), sem utilizar qualquer outra funcionalidade do *framework*. Como o Android não dá suporte a isso com suas classes padrão para o vídeo, essa utilidade pode ser muito importante. O *framework* forneceria essa implementação complexa (por lidar com utilização não trivial da JNI e do OpenGL ES em C++ e em Java) encapsulada, de fácil uso para o programador. Desse modo, o programador usuário poderia chamar um método, a partir do C/C++ ou do Java, passando como parâmetro o quadro de áudio ou de vídeo decodificado.

REFERÊNCIAS

ABLESON, W. Frank; COLLINS, Charlie; SEN, Robi. **Unlocking Android:** A Developers's Guide. [s. L.]: Manning, 2008. 372 p.

ADOBE. **Requisitos de sistema do Adobe AIR.** Disponível em: <http://www.adobe.com/br/products/air/systemreqs/>. Acesso em: nov. 2010.

ARCMEDIA. **ArcMedia - a media player for Android.** Disponível em: <http://www.arcmediasoft.com/arcMedia/index.aspx>. Acesso em: nov. 2010.

DARONCO, Leonardo Crauss. **Avaliação Subjetiva de Qualidade Aplicada à Codificação de Vídeo Escalável.** 2009. 146 f. Dissertação (Mestrado) – Programa de Pós-graduação em Computação, Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2009.

EMANI, Anudhar. **Multicast vs. P2P.** Piscataway Township: Rutgers University. Department Of Computer Science, 2008. 9 p.

FFMPEG. **FFmpeg.** Disponível em: <http://www.ffmpeg.org/>. Acesso em: nov. 2010.

FRING. **Fring- make FREE Mobile calls, video calls &live chat to other fringsters, GTalk, AIM, Facebook, Yahoo and others.** Disponível em: <http://www.fring.com/default.php>. Acesso em: nov. 2010.

FU, Xiangling et al. **Research on Audio/Video Codec Based on Android.** Beijing: School Of Software Engineering. Beijing University Of Posts And Communications, 2010. 4 p.

GARTNER. **Gartner Says Worldwide Mobile Phone Sales Grew 35 Percent in Third Quarter 2010; Smartphone Sales Increased 96 Percent.** Disponível em: <http://www.gartner.com/it/page.jsp?id=1466313>. Acesso em: nov. 2010.

HASHIMI, Sayed Y.; KOMATINENI, Satya. **Pro Android.** Berkeley: Apress, 2009. 466 p.

HEININK, Vincent E. P.. **Video-on-demand over the Internet.** Delft: Parallel And Distributed Systems Group. Faculty Of Electrical Engineering, Mathematics, And Computer Science, 2007. 32 p.

HU, Ailan. **Video-on-Demand Broadcasting Protocols: A Comprehensive Study.** Santa Clara: Intel Corporation. 2200 Mission College Blvd, 2001. 10 p.

HUANG, Cheng; LI, Jin; ROSS, Keith W.. **Can Internet Video-on-Demand be Profitable?** Kyoto: Sigcomm '07. Proceedings Of The 2007 Conference On Applications, Technologies, Architectures, And Protocols For Computer Communications, 2007. 10 p.

HUYNH-THU, Quan; GHANBARI, M.. **Temporal Aspect of Perceived Quality in Mobile Video Broadcasting.** Ipswich: Psytechnics Ltd., 2008. 11 p.

JANSSEN, Dominic. **Implementierung und Evaluierung von IP-Video-Streaming unter Android**. 2010. 140 f. Curso de Fakultät Für Informations-, Medien- Und Elektrotechnik, Departamento de Institut Für Nachrichtentechnik Labor Für Datennetze, Fachhochschule Köln University Of Applied Sciences Cologne, Cologne, 2010.

KUROSE, James F.; ROSS, Keith W.. **Computer Networking: A Top-Down Approach Featuring the Internet.** Massachusetts: Addison Wesley Longman, 1999. 679 p.

LIANG, Sheng. **The Java™ Native Interface:** Programmer's Guide and Specification. Palo Alto: Addison Wesley Longman, 1999. 318 p.

MEDIAPLAYER. **MediaPlayer:** Android Developers. Disponível em: <http://developer.android.com/reference/android/media/MediaPlayer.html>. Acesso em: dez. 2010.

MICROSOFT. **Streaming Methods: Web Server vs. Streaming Media Server.** [s. L.], [200-]. 4 p.

MOBILESOFT.KR. **Mobilesoft.** Disponível em: <http://www.mobilesoft.kr/>. Acesso em: nov. 2010.

MOVICHA. **Movicha.** Disponível em: <http://www.movicha.com/>. Acesso em: nov. 2010.

NDK, Android. **Android NDK:** Android Developers. Disponível em: <http://developer.android.com/sdk/ndk/index.html>. Acesso em: dez. 2010.

NUNES, Márcio Serafim. **Redes com Integração de Serviços:** Streaming sobre Redes IP. [s. L.]: Instituto Superior Técnico, [200-]. 21 p.

PLATFORMVERSIONS. **Platform Versions: Android Developers.** Disponível em: <http://developer.android.com/resources/dashboard/platform-versions.html>. Acesso em: nov. 2010.

REDIRECT INTELLIGENCE CORP. **Rock Player - Redirect Intelligence Corp.** Disponível em: <http://rockplayer.freecoder.org/index_en.html>. Acesso em: nov. 2010.

RHEE, Injong; JOSHI, Srinath R.. **FEC-based Loss Recovery for Interactive Video Transmission – Experimental Study.** Raleigh: Department Of Computer Science, North Carolina State University, 1998. 23 p.

ROESLER, Valter; HUSEMANN, Ronaldo; COSTA, Carlos H.. **A new multimedia synchronous distance learning system: the IVA study case.** In Proceedings of SAC'2009. pp.1765~1770

SIPDROID. **Free sip/void client for android.** Disponível em: <http://sipdroid.org/>. Acesso em: mai. 2011.

SONG, Maoqiang et al. **Design and Implementation of Media Player Based on Android.** Beijing: School Of Software Engineering. Beijing University Of Posts And Communications, 2010. 4 p.

TANGO. **Tango.** Disponível em: <http://tango.me/>. Acesso em: nov. 2010.

TSCHÖKE, Clodoaldo. **Criação de Streaming de Vídeo para Transmissão de Sinais de Vídeo em Tempo Real pela Internet.** Blumenau: Universidade Regional De Blumenau Centro De Ciências Exatas E Naturais Curso De Ciências Da Computação (bacharelado), 2001. 82 p.

WUZHENHUA PLAYER. **Media player for Android that can handle WMV, FLV, Xvid, etc.** Disponível em: <http://mobiputing.com/2010/03/media-player-for-android-that-can-handle-wmv-flv-xvid-etc/>. Acesso em: nov. 2010.