Uma nova API para criação de videoconferências em dispositivos móveis Android

Giancarlo Rampanelli
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Bloco IV, Prédio 72, Sala 220
Porto Alegre, Brasil
gian419@gmail.com

Felipe Cecagno
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Bloco IV, Prédio 72, Sala 220
Porto Alegre, Brasil
fcecagno@gmail.com

André Schulz
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Bloco IV, Prédio 72, Sala 220
Porto Alegre, Brasil
afertschulz@gmail.com

Valter Roesler
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Bloco IV, Prédio 72, Sala 233
Porto Alegre, Brasil
roesler@inf.ufrgs.br

RESUMO

Este artigo tem o objetivo de apresentar uma nova estratégia para criação de videoconferências em dispositivos móveis com sistema operacional Android. Será 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 são comparadas, ressaltando prós e contras, e conduzindo a necessidade da utilização de uma nova estratégia, detalhada neste artigo, que é baseada principalmente em bibliotecas alternativas. A ideia proposta é então validada num sistema Android real, mostrando suas vantagens.

ABSTRACT

This article aims to present a strategy for implementation of real-time interactive multimedia systems on Android-powered mobile devices. We will present a study about the standard multimedia classes of Android's software development kit, 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, highlighting pros and cons of each one, concluding with the contribution of this paper, a solution based primarily on alternative libraries.

Categories and Subject Descriptors

[**Peer-to-peer multimedia systems and streaming**]; [**Software development using multimedia techniques**]

Keywords

Interação multimídia de tempo real, Vídeo, JNI

1. INTRODUÇÃO

VERIFICAR SE OS AUTORES FICARÃO DESSA FORMA OU AGRUPADOS

VERIFICAR CATEGORIAS OFICIAIS DA ACM

VERIFICAR TERMOS QUE SERÃO UTILIZADOS

FAZER INTRODUÇÃO

2. VÍDEO INTERATIVO EM TEMPO REAL

ESTE É UM CAPÍTULO QUE PROVAVELMENTE IREMOS REDUZIR OU REMOVER PARA CABER TUDO EM OITO PÁGINAS

Um sistema que permite que uma pessoa comunique-se, ao vivo, com uma ou mais pessoas, cada qual em sua máquina, através da transmissão da sua fala (capturada pelo microfone) e da sua imagem em forma de vídeo (capturada pela câmera), ao mesmo tempo em que escuta e visualiza os outros participantes, representa um sistema de transmissão de vídeo interativo em tempo real (doravante chamado STVITR) **MELHORAR ESSA ABREVIAÇÃO**. Tais sistemas são complexos de serem desenvolvidos, já que, além da necessidade do áudio e do vídeo serem transmitidos, recebidos e exibidos de forma contínua por todos os participantes, é preciso que essa transmissão ocorra com um baixíssimo atraso: se o intervalo de tempo entre o instante da captura do áudio e do vídeo e o instante da sua exibição nas

máquinas dos outros participantes for maior que 400 milissegundos, por exemplo, o resultado pode ser uma interação altamente frustrante (KUROSE; ROSS, 1999).

É um desafio desenvolver com qualidade tais sistemas pois as redes atuais, em geral, suportam apenas uma pequena taxa de bit/s. O desafio se torna ainda maior na área dos dispositivos móveis, que, além de terem hardware limitado, que torna a tarefa da decodificação e da exibição do vídeo menos eficiente, também têm limitações de rede, pois costumam utilizar 3G ou Internet sem fio, ambos suscetíveis a alta perda de pacotes e, na maioria das vezes, a baixas velocidades (HUYNH-THU; GHANBARI, 2008).

Um exemplo de utilização dessa técnica se dá em videoconferências, que correspondem a reuniões remotas nas quais os participantes podem, por vídeo, interagir e colaborar com o propósito da reunião. Mesmo estando, cada um, em lugares diferentes, a utilização do vídeo lhes sugere que todos estão presentes em uma mesma sala, aumentando a efetividade da colaboração em comparação a uma reunião remota por texto ou por voz apenas (TSCHÖKE, 2001).

Um STVITR pode ser dividido, basicamente, em dois módulos:

- Um módulo para gerenciar o lado da origem, dividido em 3 sub-módulos:
  - Um sub-módulo para capturar quadros de áudio e de vídeo não comprimidos do microfone e da câmera, respectivamente.
  - Um sub-módulo para codificar os quadros de áudio e de vídeo.
  - Um sub-módulo para enviar pela rede os dados codificados.
- Um módulo para gerenciar o lado do destino, dividido em 3 sub-módulos:
  - Um sub-módulo para receber pela rede dados de áudio e de vídeo codificados.
  - Um sub-módulo para decodificar os dados.
  - Um sub-módulo para renderizar os quadros de áudio e de vídeo decodificados.

3. LIMITAÇÕES DAS CLASSES PADRÃO DO ANDROID PARA DESENVOLVER UM STVITR

Android é um sistema operacional de código aberto (desenvolvido pela empresa Google) para dispositivos móveis construído sobre uma versão modificada do kernel Linux. Em comparação aos outros sistemas operacionais para smartphones existentes, o Android é caracterizado por aplicações de vídeo mais simples. Isso se deve às limitações das APIs fornecidas pelo Android para programação de vídeo e, alternativamente, à grande complexidade para criar tais aplicativos sem a utilização dessas APIs. A dificuldade de se encontrar bons aplicativos de vídeo para Android se torna ainda mais marcante na área de STVITR.

Devido ao crescente número de usuários do Android, a demanda por aplicações de vídeo diferenciadas (com maior número de funcionalidades) para esse sistema operacional da Google aumenta no mesmo ritmo. Para poder suprir essa demanda de forma rápida e qualificada, os desenvolvedores de aplicativos procuram por frameworks de que lhes deem mais flexibilidade e mais recursos na programação em comparação às APIs tradicionais. Neste trabalho, será proposta uma solução que visa servir como alternativa às classes padrão do Android. Essa ideia será implementada na forma de um framework **COMO O FRAMEWORK NAO ESTÁ COMPLETO PODEMOS INCLUÍ-LO COMO TRABALHO FUTURO E APRESENTAR APENAS A SOLUÇÃO, OU ENTAO PODEMOS SUPOR QUE O FRAMEWORK ESTA PRONTO, JA QUE COMPROVAMOS QUE A ESTRATEGIA FUNCIONA**, visando abstrair dos seus usuários (os programadores) as partes mais trabalhosas e menos intuitivas da alternativa, para que seja possível a criação de tais aplicativos de vídeo para Android de forma simples e rápida.

Aplicativos para Android são escritos com a linguagem de programação Java e rodam na 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).

Considerando os dois módulos de um STVITR descritos anteriormente, as classes fornecidas pelo Android que têm utilidade para cada um desses módulos são: *Para o módulo de origem: Classe MediaRecorder, Classe Camera, Classe AudioRecord. *Para o módulo de destino: Classe MediaPlayer, classe AudioTrack.

Existem outras classes relacionadas, mas não passam de pequenas extensões ou encapsulamentos das classes acima, sem importância para este trabalho. Também há classes relacionadas a envio e recebimento de dados genéricos pela rede que poderiam ser utilizadas em conjunto com as classes acima. A seguir serão explicadas as utilidades básicas de cada uma dessas cinco classes. Então, com base nessa descrição, serão levantadas possíveis estratégias de se desenvolver um STVITR com elas, e as limitações de cada estratégia.

3.1 CLASSE MEDIARECORDER

A Classe MediaRecorder realiza a captura de áudio e de vídeo (a partir do microfone e da câmera, respectivamente) e codifica-os utilizando o hardware do aparelho. Os dados codificados podem ser salvos encapsulados em um arquivo, ou podem ser transmitidos pela rede com a utilização de sockets. A classe também pode exibir um preview da captura para o usuário local. Basicamente, as etapas necessárias são as seguintes:

1. Declarar uma instância da classe: MediaRecorder mMediaRecorder = new MediaRecorder();
2. Escolher o destino (arquivo ou socket): mMediaRecorder.setOutp
3. Iniciar a captura: mMediaRecorder.prepare(); mMediaRecorder.start();

4. Exibir o preview da captura: mMediaRecorder.setPreviewDisplay(<superfície local>);

Portanto, a classe pode ser utilizada de duas formas diferentes: salvando em um arquivo ou utilizando um socket. É importante notar que não há como acessar diretamente os dados capturados nem os codificados, pois este não é o objetivo da classe (a classe mais adequada para essa tarefa é a Camera).

3.2 CLASSE CAMERA

A classe Camera realiza a captura de vídeo (a partir da câmera) e pode exibir um preview da captura para o usuário local. Ela pode ser utilizada de duas formas: para tirar fotos (ou seja, salvar um quadro da captura, codificando-o e encapsulado-o em um arquivo) ou para instalar um callback que é chamado a cada frame capturado e que recebe os dados não codificados do frame capturado.

Basicamente, as etapas necessárias são as seguintes:

1. Obter acesso à câmera: Camera mCamera = Camera.open();
2. Escolher a superfície de preview: mCamera.setPreviewDisplay(<superfície local>);
3. Instalar o callback (opcional): mCamera.setPreviewCallback(<contexto>);
4. Iniciar a captura: mCamera.startPreview();
5. Tirar foto (opcional): takePicture(<Camera.ShutterCallback>,<Camera.PictureCallback>,<Camera.PictureCallback>);
6. Obter os dados não codificados do frame capturado (opcional): @Override public void onPreviewFrame (byte[] data, Camera camera)

3.3 CLASSE AUDIORECORD

A classe AudioRecord realiza a captura de áudio (a partir do microfone) e, enquanto captura, escreve os dados não codificados em um buffer principal. Esses dados podem ser transferidos para um buffer secundário sempre que se desejar, para que possam ser utilizados.

Basicamente, as etapas necessárias são as seguintes:

1. Declarar uma instância da classe: AudioRecord mAudioRecord = new AudioRecord(MediaRecorder.AudioSource.MIC, <taxa de amostragem>,<numero de canais>,<bits por amostra>,<tamanho do buffer principal>);
2. Iniciar a captura: mAudioRecord.startRecording();
3. Transferir o conteúdo do buffer principal (quando desejado) para um buffer secundário: record.read(<buffer secundário>,<índice inicial do buffer secundário>,<número de bytes a transferir>);

3.4 CLASSE MEDIAPLAYER

A classe MediaPlayer fornece funções de alto nível para a exibição de vídeos. Basicamente, permite a escolha da origem do vídeo (arquivo local, servidor web ou streaming por RTSP), e faz o início da exibição (HASHIMI; KOMATINENI, 2009). Dada a origem do vídeo, a classe realiza a decodificação do áudio e do vídeo em hardware e os renderiza.

Basicamente, as etapas necessárias são as seguintes:

1. Declarar uma instância da classe: MediaPlayer mMediaPlayer = new MediaPlayer();
2. Escolher a origem do vídeo: mMediaPlayer.setDataSource(<origem do vídeo>);
3. Iniciar o processo de recebimento (caso a origem do vídeo seja remota), decodificação e renderização: mMediaPlayer.prepare(); mMediaPlayer.start();

3.5 CLASSE AUDIOTRACK

A classe AudioTrack serve para renderizar áudio. Sempre que se desejar renderizar áudio com ela, deve-se transferir os dados de áudio não codificados de um buffer secundário para o buffer principal da classe.

Basicamente, as etapas necessárias são as seguintes:

1. Declarar uma instância da classe: mAudioTrack = new AudioTrack(<tipo de stream>,<taxa de amostragem>,<numero de canais>,<bits por amostra>,<tamanho do buffer principal>,<modo (estático ou stream)>);
2. Caso o modo utilizado seja o stream, chamar o método play(). mAudioTrack.play(); Nada será renderizado pois o buffer principal está vazio.
3. Transferir o conteúdo do buffer secundário (quando desejado) para o buffer principal: mAudioTrack.write(<buffer secundario>,<índice inicial do buffer secundário>,<número de bytes a transferir>); Caso o modo utilizado seja o stream, cada vez que o buffer principal receber novos dados, eles serão reproduzidos sem a necessidade de chamar o método play().
4. Caso o modo utilizado seja o estático, chamar o método play() para renderizar os dados (quando desejado): mAudioTrack.play();

4. ESTRATÉGIAS PARA STVITR COM AS CLASSES PADRÃO

4.1 ESTRATÉGIA 1

É a técnica implementada em **TRAB ALEMAO**.

4.1.1 Do lado da origem

Esta estratégia consiste em iniciar uma captura de áudio e de vídeo com a classe MediaRecorder passando como parâmetro ao método setOutputFile um descritor de arquivo local (ex: /sdcard/temp0.3gp) e finalizar a captura após um período curto (ex: 2 segundos). Os dados codificados do arquivo que está sendo gerado vão sendo enviados desde o início da captura, utilizando uma classe do Android para transmissão

genérica de dados. Seria muito mais simples passar como parâmetro ao método setOutputFile um socket localizado no destino. Assim os dados seriam enviados diretamente sem a utilização de arquivo local. No entanto, estamos querendo criar um sistema de STVITR utilizando apenas as classes padrão do Android, e a classe MediaPlayer (única capaz de reproduzir vídeo) reproduz apenas dados encapsulados ou com o container 3GPP ou com o container MPEG-4 (exceto quando é utilizado RTSP, que será abordado na estratégia seguinte). Ambos containers são muito similares, e só podem ser incluídos ao final da gravação do arquivo. A classe MediaRecorder, ao final da gravação de um arquivo, insere, no início do arquivo, o header e os metadados do container escolhido. O método, então, ao finalizar o período curto de gravação, abre o início do arquivo que foi gerado localmente e envia os headers e os metadados para o destino. Enquanto os headers e os metadados são enviados, repete-se todo o processo com outro arquivo local (ex: /sdcard/temp1.3gp) de mesma duração do anterior, e assim sucessivamente.

4.1.2 Do lado do destino

O lado do destino vai recebendo os dados de áudio e vídeo que estão sendo enviados e vai armazenando-os em um arquivo. Quando, enfim, chegam o header e os metadados do container escolhido, eles são inseridos no início desse arquivo. Somente nesse instante a classe MediaPlayer pode reproduzí-los. O processo é repetido para cada arquivo curto capturado, mas os dados são armazenados em um arquivo diferente para não sobrescrever o que está sendo reproduzido pela classe MediaPlayer.

4.1.3 Análise da estratégia

Segundo **TRAB ALEMAO**, a classe MediaRecorder leva 1,3 segundos (em média) para ser inicializada. E não é possivel inicializar um novo MediaRecorder antes de finalizar o anterior. Logo, utilizando esta estratégia, ocorreriam períodos de 1,3 segundos em que não se poderia capturar nada. Do lado do destino, considerando apenas essa limitação, isso seria percebido como travamentos de 1,3 segundos no vídeo seguidos de saltos de 1,3 segundos na sua exibição. A classe MediaPlayer só pode ser inicializada quando o arquivo estiver pronto, e a demora na sua inicialização é, em média, 0,83 segundos. supondo:

- duração de cada captura = 2s.
- tempo para inicializar a classe MediaRecorder = 1.3s.
- tempo para inicializar a classe MediaPlayer = 0.9s.
- tempo para a classe MediaRecorder inserir os metadados no início do arquivo = 0.1s.
- tempo para o lado do destino inserir os metadados no início do arquivo = 0.1s.
- tempo para transmitir um pacote pela rede = 0.1s.
- t = 0s a 1.3s: lado da origem: Classe MediaRecorder é inicializada.
- t = 1.3s: lado da origem: a inicialização da Classe MediaRecorder está pronta e a captura inicia.
- t = 1.3s a 3.3s: lado da origem: o arquivo local "01origem" está sendo gerado e os dados úteis do vídeo estão sendo transmitidos.
- t = 1.4s: lado do destino: os primeiros pacotes começam a ser recebidos.
- t = 3.3s: lado da origem: a captura é finalizada.
- t = 3.4s: lado da origem: os metadados são inseridos pela classe MediaRecorder no início do arquivo local "01origem". Em seguida o header e os metadados são enviados ao destino.
- t = 3.4s: lado do destino: todos os pacotes de áudio e vídeo já foram recebidos. Faltam o header e os metadados.
- t = 3.4s: lado da origem: a classe MediaRecorder é inicializada novamente.
- t = 3.5s: lado do destino: o header e os metadados são recebidos.
- t = 3.6s: destino: o header e os metadados são inseridos no início do arquivo "01destino".
- t = 3.6s: destino: a classe MediaPlayer é inicializada.
- t = 4.5s: destino: a classe MediaPlayer começa a renderização do arquivo "01destino". (delay = 4.5 - 1.3 = 3.2s).
- t = 4.7s: origem: a inicialização da Classe MediaRecorder está pronta e a captura inicia.
- t = 4.7s a 6.7s: origem: o arquivo local "02origem" está sendo gerado e os dados úteis do vídeo estão sendo transmitidos.
- t = 4.8s: destino: os primeiros pacotes da nova captura começam a ser recebidos.
- t = 6.5s: destino: a classe MediaPlayer finaliza a renderização do arquivo "01destino".
- t = 6.7s: origem: a captura é finalizada.
- t = 6.8s: lado da origem: os metadados são inseridos pela classe MediaRecorder no início do arquivo local "02origem". Em seguida o header e os metadados são enviados ao destino.
- t = 6.8s: destino: todos os pacotes de áudio e vídeo já foram recebidos. Faltam o header e os metadados.
- t = 6.8s: origem: a classe MediaRecorder é inicializada novamente.
- t = 6.9s: destino: o header e os metadados são recebidos.
- t = 7.0s: destino: o header e os metadados são inseridos no início do arquivo "02destino".
- t = 7,0s: destino: a classe MediaPlayer é inicializada.
- t = 7.9s: destino: a classe MediaPlayer começa a renderização do arquivo "02destino". (delay = 7.9 - 4.7 = 3.2s. Tempo sem exibição = 7.9 - 6.5 = 1.4s).

- t = 8.1s: origem: a inicialização da Classe MediaRecorder está pronta e a captura inicia.
- t = 8.1s a 10.1s: origem: o arquivo local “03origem” está sendo gerado e os dados úteis do vídeo estão sendo transmitidos.
- t = 8.2s: destino: os primeiros pacotes da nova captura começam a ser recebidos.
- t = 9.9s: destino: a classe MediaPlayer finaliza a renderização do arquivo “02destino”.
- t = 10.1s: origem: a captura é finalizada.
- t = 10.2s: lado da origem: os metadados são inseridos pela classe MediaRecorder no início do arquivo local “03origem”. Em seguida o header e os metadados são enviados ao destino.
- t = 10.2s: destino: todos os pacotes de áudio e vídeo já foram recebidos. Faltam o header e os metadados.
- t = 10.2s: origem: a classe MediaRecorder é inicializada novamente.
- t = 10.3s: destino: o header e os metadados são recebidos.
- t = 10.4s: destino: o header e os metadados são inseridos no início do arquivo “03destino”.
- t = 10.4s: destino: a classe MediaPlayer é inicializada.
- t = 11.3s: destino: a classe MediaPlayer começa a renderização do arquivo “03destino”. (delay = 11.3 - 8.1 = 3.2s. Tempo sem exibição = 11.3 - 9.9 = 1.4s).

E assim por diante. **TALVEZ FOSSE MELHOR REPRESENTAR ESSA LINHA DE TEMPO COM UMA IMAGEM**

Portanto, nessa simulação, o delay é de 3.2 segundos. Além disso, ocorrem pausas de 1.4s a cada 2s de exibição do vídeo. Em um STVITR, o maior delay recomendado é de 400ms. Não existe uma norma para determinar a aceitabilidade pausas na exibição, mas é evidente que as pausas obtidas nesse método tornariam a comunicação altamente frustrante. O autor reconhece que a técnica não serve para STVITR, mas seu objetivo era identificar as limitações das APIs e propor a melhor aproximação possível de um STVITR. Segundo o autor, essa provavelmente é a técnica possível de ser implementada com as classes padrão do Android que mais se aproxima de um STVITR. No entanto, ele deixa claro que sempre podem surgir novas ideias.

4.2 ESTRATÉGIA 2

É a técnica implementada em **SIPDROID**.

Esta técnica utiliza a capacidade da classe MediaPlayer em reproduzir conteúdo ao vivo **DIFERENCIAR CONTEÚDO AO VIVO DE CONTEÚDO EM TEMPO REAL** utilizando o protocolo RTSP.

4.2.1 Lado da origem

A classe MediaRecorder é utilizada para capturar vídeo. O método setOutputFile recebe como parâmetro um descritor de arquivo obtido de um socket implementado pela classe LocalSocket do Android. Isso significa que os dados capturados não são salvos em um arquivo local, mas sim diretamente enviados para um socket. Os dados do socket são lidos e tratados para ficarem de acordo com o protocolo RTSP.

4.2.2 Lado do destino

A classe MediaPlayer é inicializada e o seu método setDataSource é setado para o endereço do stream RTSP.

4.2.3 Análise do método

O método acaba resultando em um sistema de vídeo ao vivo, e não em um STVITR, pois a implementação de RTSP do Android não serve para tempo real. Para provar isso, realizamos o seguinte teste :

1. Capturamos vídeo com uma webcam conectada a um desktop utilizando o software VideoLAN .
2. Direcionamos o fluxo para um servidor de streaming RTSP (Darwin Streaming Server) especificado por um arquivo SDP (Session Description Protocol).
3. Recebemos o stream em um Motorola Milestone rodando Android 2.0.1 utilizando a API MediaPlayer.

O delay obtido foi de 10 segundos. Recebendo o mesmo stream no próprio VLC (ao invés de recebê-lo no Android), o delay foi de 1 segundo.

Também fizemos uma comparação do RTSP do Android com o RTSP do VideoLAN utilizando diversos links RTSP encontrados na Internet. Para todos eles, o vídeo exibido no Android foi exibido com um atraso entre 2 e 20 segundos após o mesmo vídeo sendo exibido no VídeoLAN.

Comparado ao método anterior, neste método não ocorrem pausas na exibição. Porém, segundo os testes de desempenho realizados com a classe MediaPlayer, o delay é maior que o do método anterior, devido ao baixo desempenho do RTSP utilizado juntamente com a classe MediaPlayer do Android.

4.3 ESTRATÉGIA 3

Ao invés de utilizar a classe MediaRecorder para capturar, utilizar as classes Camera e AudioRecord. Com ambas as classes é possível se obter os dados não codificados dos quadros de vídeo e de áudio capturados. No entanto, não há nenhuma classe padrão do android que receba como parâmetro um quadro não codificado e que codifique-o. E quadros não codificados são grandes demais para serem transmitidos pela rede em tempo viável para STVITR. Poderia ser implementado um codificador/decodificador em Java. No entanto, a tarefa da codificação/decodificação é extremamente custosa computacionalmente, além de ser muito complexa de ser desenvolvida. Implementar uma tarefa custosa em uma linguagem que roda em máquina virtual que, por sua vez, roda em um dispositivo de baixo poder computacional, seria

pouco eficiente. Além disso, mesmo que fosse possível codificar os quadros em Java, enviá-los ao destino e decodificá-los em Java em tempo viável para STVITR, não haveria como reproduzí-los utilizando as classes padrão do Android, visto que nenhuma delas tem a capacidade de receber como parâmetros quadros decodificados. Logo, esta estratégia não é possível.

4.4 ESTRATÉGIA 4

4.4.1 Lado da origem

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

4.4.2 Lado do destino

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

4.4.3 Análise da estratégia

Codecs de vídeo são mais eficazes que o codec de imagens JPEG, visto que um vídeo é um conjunto de imagens relacionadas entre si, e a compressão é realizada com base na relação entre essas imagens. Um codec JPEG apenas comprime uma imagem sem relacioná-la com nenhuma outra, portanto tem menos poder de compressão. Além disso, a função foto do Android não foi projetada para este uso e é pouco eficiente. Um dos motivos é que quando a função foto é chamada, o preview do vídeo capturado é interrompido. Para tirar uma nova foto é preciso reiniciar o preview. O método em si não foi testado, mas certamente a taxa de frames e o delay obtidos não chegariam perto do recomendado para STVITR. Outro problema dessa estratégia é não ter um correspondente para o áudio. Somente o vídeo seria transmitido.

4.5 ESTRATÉGIA 5

Como o Android é de código aberto, essa estratégia seria copiar as libraries nativas (C/C++) internas (privadas) ao sistema operacional que interagem com as classes java MediaPlayer e MediaRecorder e com o hardware dos aparelhos e colá-las no em um novo projeto. Em seguida, compilaríamos essas libraries dentro do nosso projeto, com a utilização da NDK **EXPLICAR O QUE É A NDK**. Desse modo, teríamos acesso aos quadros codificados/decodificados pelo hardware e, desse modo, poderíamos utilizar alguma classe genérica de transmissão de dados do Android para realizar a transmissão/recepção desses quadros. Assim, a implementação ficaria desta forma:

Captura: câmera e microfone seriam acessados diretamente pelo C++, utilizando as libraries privadas do android correspondentes.

Codificação: em hardware, utilizando as libraries privadas do android correspondentes.

Transmissão: alguma classe padrão do Android para transmissão genérica de dados.

Recepção: alguma classe padrão do Android para transmissão genérica de dados.

Decodificação: em hardware, utilizando as libraries privadas do android correspondentes.

Renderização: display e alto falante seriam acessados diretamente pelo C++, utilizando as libraries privadas do android correspondentes.

Esta seria a solução mais eficiente possível, visto que todas as tarefas pesadas seriam realizadas em hardware, e os quadros trafegariam pela rede comprimidos. Contudo, como as bibliotecas nativas interagem com o resto do Android, e o sistema operacional muda muito a cada nova versão (e muda a cada modelo de aparelho também, mesmo dentro de uma mesma versão), haveria probabilidade de 50% **SEGUNDO DESENVOLVEDOR DO ANDROID, APRESENTAR FONTE** do aplicativo deixar de funcionar em diferentes versões do Android ou funcionar apenas no modelo de aparelho cujo código utilizamos, e a manutenção para fazê-lo voltar a funcionar ou para que funcione em múltiplos aparelhos com certeza seria inviável em termos de complexidade. Por esses motivos, tal uso das bibliotecas privadas do Android não deve ser feito, segundo os desenvolvedores do Android. Além disso, com essa solução estaríamos limitados a utilizar os codecs suportados por padrão pelos aparelhos android. Por todos esses motivos, descartamos essa alternativa.

5. SOLUÇÃO ADOTADA

Pela análise acima, pode-se perceber que não foi possível contruir um STVITR utilizando-se apenas as classes padrão do Android. É claro que novas idéias podem surgir. A solução adotada baseia-se em algumas das classes padrão do Android apresentadas, mas sua parte principal é focada em implementações independentes.

A solução consiste em:

5.0.1 Lado da origem

1. Utilizar a classe Camera para capturar os quadros de vídeo e instalar um callback que receba cada frame não codificado obtido da captura. Com o auxilio dessa API, nós setamos os parâmetros de captura desejados (resolução, quadros/s, etc) e enviamos cada frame capturado no formato YUV420SP (também conhecido como NV21, que é o único formato de captura suportado por todos os aparelhos Android) para o C++. A JNI **EXPLICAR O QUE É A JNI** é utilizada para passar os frames do Java para o C++. Um detalhe que convém citar é a utilização de Java reflection **EXPLICAR O QUE É JAVA REFLECTION** para que se possa utilizar a função setPreviewCallbackWithBuffer em versões do android anteriores a 2.2, quase dobrando a taxa de frames em comparação ao método setPreviewCallback, visto que essa função evita que o GC (garbage collector do android) interrompa o programa a cada frame.

2. Utilizar a classe AudioRecord para capturar os quadros de áudio, e obter os quadros não codificados da captura. Para isso desenvolvemos uma classe em Java que faz uso da classe AudioRecord, e captura o audio do microfone. Cada frame capturado é enviado ao C++ no formato RAW (PCM) com a utilização da JNI.

3. Como o Android roda sobre um kernel Linux (FU et al., 2010), foi possível compilar o ffmpeg **EXPLICAR O QUE É O FFMPEG** em C (com a NDK) adaptado para o Android. Desse modo, é possível codificar cada quadro de forma mais eficiente que em um codificador Java. No entanto, a implementação de diversos codecs de vídeo importantes fornecida pelo ffmpeg requer que o quadro decodificado de entrada esteja no formato YUV420P. Nesses casos, antes de codificar o quadro, é preciso convertê-lo de YUV420SP para YUV420P, numa função implementada por nós.

4. Enviar os frames codificados para o destino utilizando a implementação de sockets do C/C++. (Poderia-se passar os quadros codificados para o Java e enviá-los com alguma classe padrão do Android, mas a tarefa de passar os quadros com a JNI é custosa e complexa de ser implementada).

5.0.2 Lado do destino

1. Receber os quadros com a implementação de sockets do C/C++. (Igualmente poderia-se utilizar alguma classe padrão do Android, mas, pelo mesmo motivo citado no item Anterior, não é o mais indicado).

2. Decodificar os quadros de áudio e de vídeo com o ffmpeg em C/C++ para o formato RGB. Para os casos em que o ffmpeg apenas decodifica para YUV420P, criamos uma função que converte de YUV420P para RGB.

3. Não há como acessar o alto falante e para renderizar os quadros de áudio diretamente do C++. A solução encontrada é passar os quadros de áudio decodificados para a classe AudioTrack utilizando a JNI. Desse modo, eles são reproduzidos. A API AudioTrack toca frames de audio que são entregues a ela no formato RAW (PCM). Para enviarmos os frames de áudio do C++ para o Java, primeiramente tentamos chamar uma função Java a partir do C++ utilizando a JNI, passando como parâmetro o frame, que seria a solução mais simples. No entanto, esta solução ativa o Garbage collector a cada frame, o que causa uma enorme queda no desempenho, logo essa alternativa foi descartada. A alternativa que implementamos para evitar o GC foi utilizar uma técnica da JNI um pouco mais trabalhosa. Em resumo, essa técnica consiste na criação de um array em Java que representa um frame e de um array em C++ que também representa um frame. Em seguida, com chamadas a algumas funções da JNI, associamos os dois arrays para a mesma posição de memória para que eles sejam tratados como se fossem um só. Assim, cada vez que se atualiza o array do C++ com os dados do novo frame decodificado, essa atualização é automaticamente propagada ao array do java, sem que seja feita nenhuma cópia e nenhuma alocação de memoria adicional, resultando na solução mais eficiente que encontramos para renderizar o áudio.

4. Não há como acessar o display para renderizar os quadros de vídeo diretamente do C++. Também não há alguma classe Java que receba como parâmetros quadros decodificados e renderize-os. A solução encontrada foi renderizar os quadros de vídeo com OpenGL ES **EXPLICAR O QUE É OPENGL ES** em C++, desenhando um retângulo com as dimensões desejadas e aplicando o frame de vídeo decodificado como textura RGB ao retângulo. No entanto, para que a renderização seja exibida no display, é necessário que a função de renderização do C++ seja chamada de dentro do método onDrawFrame da classe GLSurfaceView.Renderer do Java e retorne para esse método a cada frame.

Outro detalhe importante, caso o frame decodificado esteja numa resolução, e se queira renderizá-lo em outra, verificamos que realizando esse redimensionamento com o ffmpeg é mais lento que com o OpenGL ES. Ainda assim, caso a versão do OpenGL ES do aparelho utilizado seja a 1.0, o redimensionamento é feito em software. Caso seja 1.1 em diante, o redimensionamento é feito em hardware. Outro detalhe: algumas versões do OpenGL ES suportam apenas texturas sobre retângulos de resolução cujas dimensões são potência de 2. Para poder desenhar o vídeo em qualquer resolução, criamos uma função que gera um retângulo com dimensões de potência de dois imediatamente maiores que as que irão ser utilizadas. Em seguida, a cada frame, aplica-se a textura com qualquer dimensão sobre parte desse retângulo.

Também realizamos cálculos considerando a resolução máxima da tela e a orientação do celular para que a exibição do vídeo esteja sempre centralizada na tela, independentemente da resolução e da orientação.

6. ANÁLISE TÉCNICA

1. Inicialmente foi testada a decodificação e a exibição de um arquivo de vídeo local com a estratégia implementada. Como resultado, para um vídeo de resolução 176x144, foram decodificados 227 quadros por segundo. A taxa de exibição obtida foi de 60 quadros por segundo (valor máximo suportado pelo aparelho).

2. **APRESENTAR INTEGRAÇÃO COM O IVA E O RESULTADO DA INTEGRAÇÃO, PRIMEIRO DO DESKTOP PARA O CELULAR, DEPOIS DE CELULAR PARA CELULAR. PARA ISSO BASTA COPIAR AS TABELAS DO OUTRO ARTIGO**.

3. **APRESENTAR A INTEGRAÇÃO AO MCONF**.

4. **VERIFICAR SE NÃO HOUVE OUTROS TESTES QUE PODEM SER INCLUÍDOS**.

7. CONCLUSÃO

ESCREVER CONCLUSÃO

8. TRABALHOS FUTUROS

Estamos desenvolvendo um framework que irá encapsular todas as partes complexas da implementação dessa técnica. O framework permitirá que seja possível implementar aplicativos de interação por vídeo com a estratégia acima de modo simples e rápido, mas ao mesmo tempo oferecendo flexibilidade ao programador com relação a aspectos como, por exemplo, escolha do codec de vídeo a ser utilizado e 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 será possível 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.

O framework deverá consistir, basicamente, de **COMO O FRAMEWORK NÃO ESTÁ PRONTO, ESCREVI COMO EU IMAGINO QUE SERÁ**:

- Uma classe Java para realizar a captura e o envio dos frames de vídeo capturados não codificados para o C++ e para renderizar o preview da captura.
- Uma classe Java para realizar a captura e o envio dos frames de áudio capturados não codificados para o C++.
- Uma classe Java que extenda a classe GLSurfaceView do Android para ser a superfície na qual os frames serão renderizados.
- Uma classe Java que implemente a classe GLSurfaceView.Renderer do Android. Essa classe implementará o método onDrawFrame e chamará a função do C++ responsável por renderizar o frame.
- Uma classe Java responsável por renderizar os quadros de áudio decodificados provenientes do C++.
- Um elemento de layout **EXPLICAR O QUE É UM ELEMENTO DE LAYOUT DO ANDROID** que permitirá a fácil inclusão de uma superfície para exibir o vídeo que está sendo decodificado.
- Um elemento de layout que permitirá a fácil inclusão de uma superfície para exibir o preview do vídeo que está sendo capturado pela câmera.
- Biblitecas C++ compiladas no formato .so (shared libraries). Essas bibliotecas terão a implementação do ffmpeg e a implementação da parte nativa do framework responsável por interagir com o ffmpeg e com o Java. A implementação da parte nativa do framework corresponderá a duas classes: uma responsável por gerenciar o lado da origem e uma para o lado do destino.

A instalação do framework consistirá na criação de um Android Project no Eclipse **EXPLICAR O QUE É O ECLIPSE** ao qual as bibliotecas Java do framework deverão ser incluídas. As bibliotecas .so deverão ser incluídas em um diretório libs/armeabi dentro da raiz do projeto. **CASO O PROGRAMADOR USUÁRIO DO FRAMEWORK QUEIRA TER ACESSO ÀS BIBLIOTECAS .SO A PARTIR DO C++, ENTÃO TALVEZ SEJA NECESSÁRIO TER OS HEADERS DAS BIBLIOTECAS, MAS NÃO TENHO CERTEZA DISSO. E NESSE CASO ELE TAMBÉM TERÁ QUE FAZER ALGUNS AJUSTES AO MAKEFILE DO SEU PROJETO. MAS SÓ SABEREI DISSO DEPOIS QUE TIVER TERMINANDO DE IMPLEMENTAR O FRAMEWORK**.

A utilização do framework deverá ser, basicamente, desta forma **COMO O FRAMEWORK NÃO ESTÁ PRONTO, ESCREVI COMO EU IMAGINO QUE SERÁ**:

a) Iniciar uma captura de áudio e/ou vídeo, exibir um preview, enviar os quadros capturados do Java para o C++, codificar os quadros e enviar ao(s) destino(s) utilizando o método de envio implementado pelo framework:

O programador usuário do framework poderá escolher em qual elemento de layout ele quer exibir a captura. Por exemplo, caso ele queira exibir a captura em um Dialog **EXPLICAR O QUE É UM DIALOG**, ele deverá criar uma classe que estenda a classe Dialog do Android, chamar o método do Android setContentView passando como parâmetro a esse método o elemento de layout correspondente ao preview da captura fornecido pelo framework. Em seguida, ele deverá criar uma nova instância da classe de captura (fornecida pelo framework) com o método findViewById passando como parâmetro a esse método a view de captura fornecida pelo framework. Por fim, com a instância criada, deverá chamar o método start da classe de captura fornecida pelo framework, passando os parâmetros de captura desejados (taxa de quadros por segundo, resolução, etc **SÓ SABEREI MELHOR TODOS OS PARÂMETROS QUE SERÃO NECESSÁRIOS QUANDO TIVER TERMINANDO DE IMPLEMENTAR O FRAMEWORK. ISSO VALE PARA TODOS OS MÉTODOS E CLASSES DO FRAMEWORK** e os ips aos quais ele quer transmitir o vídeo. O código será semelhante a este:

```
class VideoDialog extends Dialog {
    public CaptureDialog(Context context) {
        setContentView(R.layout.video_capture);
        VideoCapture videoWindow;
        videoWindow = (VideoCapture) findViewById(R
        videoWindow.start(<parametros de captura>,<
    }
}
```

Caso ele queira exibir o preview da captura em outro elemento que não um Dialog (em tela cheia em uma nova activity **EXPLICAR O QUE É UMA ACTIVITY**, por exemplo), o procedimento é praticamente idêntico, porém de acordo com as características do elemento desejado, seguindo as recomendações do Android.

Para o áudio, ele apenas precisa declarar uma nova instância da classe de captura de áudio fornecida pelo framework e chamar o método start dela passando os parâmetros de captura desejados (frequência, bits por amostra, etc) e os ips aos quais ele quer transmitir o áudio.

Ao chamar o método start (tanto o do vídeo como o do áudio), a captura será iniciada e os quadros serão enviados ao C++ automaticamente. Também automaticamentes os quadros vão sendo enviados aos ips passados como parâmetro.

b) Iniciar um recebimento de áudio e/ou vídeo utilizando o método de recebimento implementado pelo framework e decodificar e renderizar os quadros:

Para exibir o vídeo, o procedimento é o mesmo que para exibir o preview da captura. A diferença é que a classe e o elemento de layout utilizados devem ser os relacionados ao destino, e não à origem. Além disso, o método start não necessita de parâmetros.

Para reproduzir o áudio, ele precisa declarar uma nova instância da classe de renderização de áudio fornecida pelo framework e, com essa instância, chamar o método start da classe.

Ao seguir os itens a) e b), o áudio e o vídeo serão exibidos sincronizadamente no destino. Esse é o uso mais simples do framework. No entanto, para dar maior flexibilidade ao programador, exitirão outros usos possíveis (cada qual deles, no entanto, adicionará mais complexidade ao desenvolvimento do framework). Por exemplo:

c) Iniciar uma captura de áudio e/ou vídeo, exibir um preview, enviar os quadros capturados do Java ao C++ e dar acesso aos dados dos quadros não codificados ao programador em C++:

Para isso o programador deverá seguir os passos do item a), mas sem informar um ip. Em seguida, deverá criar um arquivo C/C++ dentro do qual ele instalará um callback que receberá os dados desejados. **SÓ SABEREI MELHOR COMO DEVERÁ SER ESSE ARQUIVO C/C++ QUANDO TIVER TERMINANDO DE IMPLEMENTAR O FRAMEWORK. O MESMO VALE PARA O MAKEFILE QUE O USUÁRIO DEVERÁ CRIAR**. Isso poderá ser útil caso o programador não queira transmitir os dados, mas apenas realizar um processamento sobre eles ou codificá-los utilizando sua própria implementação de codec.

d) Iniciar uma captura de áudio e/ou vídeo, exibir um preview, enviar os quadros capturados do Java ao C++, codificar os quadros e dar acesso aos dados dos quadros codificados ao programador em C++:

Idem ao item c). 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 **PROVAVELMENTE ELE TERÁ ACESSO A UM TIMESTAMP DOS DADOS**.

e) Reproduzir um arquivo local, e não um stream:

SÓ SABEREI MELHOR COMO SERÁ ISSO QUANDO TIVER TERMINANDO DE IMPLEMENTAR O FRAMEWORK, MAS IMAGINO QUE O PROCESSO SERÁ SEMELHANTE AO ITEM B), PORÉM O MÉTODO START DEVERÁ RECEBER COMO PARÂMETRO O CAMINHO DO ARQUIVO LOCAL. 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.

f) 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 item d). 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.

g) 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++:

SÓ SABEREI MELHOR COMO SERÁ ISSO QUANDO TIVER TERMINANDO DE IMPLEMENTAR O FRAMEWORK, MAS IMAGINO QUE O PROCESSO SERÁ SEMELHANTE AO ITEM B), PORÉM O PROGRAMADOR DEVERÁ CRIAR UM ARQUIVO C/C++ DENTRO DO QUAL ELE INSTALARÁ UM CALLBACK QUE RECEBERÁ OS DADOS DESEJADOS. SÓ SABEREI MELHOR COMO DEVERÁ SER ESSE ARQUIVO C/C++ QUANDO TIVER TERMINANDO DE IMPLEMENTAR O FRAMEWORK. O MESMO VALE PARA O MAKEFILE QUE O USUÁRIO DEVERÁ CRIAR. 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.

h) 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++:

Idem ao item g). Isso poderá ser útil caso o programador não queira renderizar os dados, mas apenas realizar um processamento sobre eles.

i) 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 fornece essa implementação complexa (por lidar com utilização não trivial da JNI do OpenGL ES em C++ e em Java) encapsulada, de fácil uso para o programador. **SÓ SABEREI MELHOR COMO SERÁ ISSO QUANDO TIVER TERMINANDO DE IMPLEMENTAR O FRAMEWORK, MAS IMAGINO QUE O PROGRAMADOR PODERÁ CHAMAR UM MÉTODO, A PARTIR DO C/C++ OU DO JAVA, PASSANDO COMO PARÂMETRO O QUADRO DE ÁUDIO OU DE VÍDEO DECODIFICADO. PROVAVELMENTE OS FORMATOS ACEITOS SERÃO, PARA O ÁUDIO, RAW PCM, E PARA O VÍDEO, ALGUM(S) TIPO(S) DE RGB E DE YUV, MAS TAMBÉM É POSSÍVEL UTILIZAR O FFMPEG PARA CONVERTER FORMATOS, LOGO TALVEZ SEJAM ACEITOS UM MAIOR NÚMERO DE FORMATOS**.

j) SÓ CONSEGUIREI IMAGINAR OUTROS USOS POSSÍVEIS AO LONGO DA IMPLEMENTAÇÃO DO FRAMEWORK.

9. REFERENCES