UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
ESCOLA POLITÉCNICA
DEPARTAMENTO DE ELETRÔNICA E DE COMPUTAÇÃO

UMA SOLUÇÃO DE STREAMING DE VÍDEO PARA CELULARES: CONCEITOS, PROTOCOLOS E APLICATIVO

Autor: _____
Ricardo Gomes Clemente

Orientador: _____
Prof. José Ferreira de Rezende, Dr.

Examinador: _____
Prof. Luís Henrique Maciel Kosmalski Costa, Dr.

Examinador: _____
Prof. Marcelo Luiz Drumond Lanza, M.Sc.

DEL
Julho de 2006

Dedicatória

Dedico este trabalho de conclusão ao meu pai.

Agradecimento

Agradeço primeiramente a Deus.

Ao meu pai, minha mãe, meu irmão, minha namorada e toda minha família, pelo carinho e suporte incondicional.

Ao meu orientador, pela dedicação e conselhos acertados. Agradeço também aos integrantes do GTA que me ajudaram neste projeto.

Aos professores e funcionários do DEL, pelo trabalho de ensino realizado.

Ao povo brasileiro que financiou meus estudos.

Aos meus amigos da faculdade, do trabalho, etc, em especial, ao Bruno e ao Leonardo que contribuíram diretamente para o projeto e à Dona Maria por ter cuidado de mim durante esses anos.

Resumo

A distribuição de seqüências de vídeos para celulares através da técnica de *streaming* é hoje uma realidade. Este trabalho tem como objetivo introduzir os conceitos, protocolos e mecanismos necessários a uma solução completa. Para isso, foi concebido um sistema de *streaming* de vídeo que envolve desde ferramentas para a preparação do vídeo, passando pela escolha e configuração do servidor, e terminando com o desenvolvimento de uma aplicação cliente usando a linguagem Java.

Palavras-chaves

Vídeo, *streaming*, celular, programação, Java.

Índice do texto

Índice de figuras

Índice de tabelas

Siglas

3GP	*3GPP File Format*
3GPP	*3rd Generation Partnership Project*
AAC	*Advanced Audio Coding*
AMR	*Adaptative Multi-Rate*
APN	*Access Point Name*
CDC	*Connected Device Configuration*
CDMA	*Code Division Multiple Access*
CLDC	*Connected Limited Device Configuration*
DSS	*Darwin Streaming Server*
EDGE	*Enhanced Data Rates for Global Evolution*
EV-DO	*Evolution Data Optimized*
GSM	*Global System for Mobile Communications*
GPRS	*General Packet Radio Service*
HTTP	*Hyper Text Transfer Protocol*
IDE	*Integrated Development Environment*
J2ME	*Java 2 Platform Micro Edition*
J2SE	*Java 2 Platform Standard Edition*
JSR	*Java Specification Request*
MIDP	*Mobile Information Device Profile*
MMAPI	*Mobile Media API*
RTP	*Real-Time Transport Protocol*
RTCP	*RTP Control Protocol*
RTSP	*Real-Time Streaming Protocol*
RTT	*Radio Transmission Tecnology*
UMTS	*Universal Mobile Telecommunications System*
URL	*Uniform Resource Locator*
XML	*Extensible Markup Language*

1 Introdução

O setor de telecomunicações, em particular as empresas de telefonia celular, está hoje atento às novas possibilidades de serviços que as tecnologias de transmissão de dados em sua rede podem proporcionar. Constata-se um aumento no uso de serviços de dados por parte de usuários, com o crescimento do número de *bytes* trafegados, que caminha junto com avanços nas tecnologias de transmissão de dados por parte das operadoras.

A navegação na Internet via celular, assim como serviços de *download* de vídeo, já representa um percentual expressivo na receita das operadoras. Segundo estimativas [1], a receita com serviços de dados no Brasil deverá corresponder a 14% do total até o final de 2008.

Por outro lado, as empresas produtoras de conteúdo também enxergam grandes oportunidades com o estabelecimento deste novo meio de distribuição de mídias, em particular vídeos. Desta forma, a venda de conteúdo por este meio parece ser uma prática lucrativa e com grande potencial de crescimento.

A confluência de interesses destes dois mercados leva ao aumento do consumo de vídeo via celular. Atualmente, o *download* de vídeos curtos já é um serviço disponível, embora ainda represente muito pouco em termos percentuais de receita. Nesse sentido, há também uma outra modalidade que seria o consumo de vídeos através de *streaming*. *Streaming* é uma técnica que permite a transmissão de informação multimídia através de uma rede de computadores concomitantemente com o consumo desta informação multimídia por parte do usuário. Em outras palavras, enquanto o usuário assiste a um vídeo, as próximas cenas estão sendo transmitidas. Deste modo, o usuário começa a assistir a um vídeo sem antes ter que baixá-lo integralmente.

Aplicações de *streaming* de vídeo possibilitam a transmissão de vídeo ao vivo e também de vídeos maiores sem que o usuário tenha que armazená-los ou esperar muito tempo para começar a assistí-los. Este tipo de serviço também já é encontrado no Brasil, no entanto, sua utilização ainda é muito baixa.

Como infra-estrutura de transmissão de dados em redes celulares as principais tecnologias se dividem entre as redes GSM (*Global System for Mobile Communication*) e TIA/EIA IS-95 (*Telecommunications Industry Association / Electronic Industries Association Interim Standard - 95*), conhecida como CDMA (*Code Division Multiple Access)* [2]. Nas redes celulares do tipo CDMA estão hoje em funcionamento no Brasil

duas tecnologias de transmissão de dados: a 1xRTT que possui uma taxa máxima teórica de 153,6 kbits/s e a 1xEV-DO com taxa máxima teórica de 2.400 kbits/s. Relacionada à tecnologia GSM, existem também duas tecnologias em operação: o GPRS (*General Packet Radio Service*) possui taxa máxima teórica e 171,2 kbits/s de o EDGE (*Enhanced Data Rates for Global Evolution*) que possui taxa máxima teórica de 473.6 kbits/s.

No entanto estas taxas máximas teóricas, que já não é alta no caso do GPRS, nunca são atingidas na prática. Na verdade, a taxa efetiva de transmissão é muito menor, pois os *slots* de tempo são compartilhados com o tráfego de voz e com outros usuários GPRS, ficando na média em torno de 30 kbits/s para o GPRS [2]. Esta banda limitada, tendo como base a tecnologia GPRS, dificulta a garantia de qualidade de um *streaming* de vídeo para celulares. Nesta área, há ainda outros problemas mais complexos como a questão da mobilidade e do tratamento de erros na camada de rádio que também prejudicam a qualidade de um *streaming* de vídeo.

Nas redes celulares de terceira geração sucessoras do GSM, como o UMTS (*Universal Mobile Telecommunications System*), existe uma especificação criada pelo órgão 3GPP (*3rd Generation Partnership Project*) que contém uma arquitetura para serviços de *streaming* por comutação de pacotes. Esta especificação, TS 26.234 [3], é nomeada de *Transparent End-to-End Packet Switched Streaming Service (PSS)*. Nesse serviço, a especificação define os protocolos a serem utilizados, um vocabulário para descrição das características do aparelho com relação a *streaming*, codificadores que devem ser utilizados, métricas de qualidade, entre outros.

Por parte do cliente do *streaming* de vídeo, no caso um aparelho celular, existem também outras questões importantes. A primeira questão é a restrição de processamento e de memória, quando comparado ao desenvolvimento de aplicações para computadores pessoais ou servidores. A segunda questão é o fato de cada modelo de celular definir um novo ambiente de execução o que dificulta o desenvolvimento de aplicações que possam ser executadas em um grande número de modelos. Pode-se adiantar que estas duas primeiras questões foram fundamentais para a escolha da linguagem Java com sua plataforma J2ME (*Java 2 Platform Micro Edition*), para o desenvolvimento deste projeto. Outra questão é a limitação na interface com usuário através de uma tela

pequena e de poucos botões, o que leva a uma maior preocupação com a usabilidade do aplicativo.

Todas estas questões e problemas motivaram este projeto no sentido de que fosse adquirido um maior conhecimento sobre o tópico: *streaming* de vídeo para celular. Neste sentido, o projeto foi planejado de modo que as principais questões que envolvem este tópico, desde a produção do vídeo, passando por sua transmissão, até a visualização no celular pudessem ser abordadas.

Sendo assim, o produto final deste projeto é um sistema completo de *streaming* de vídeo para celular, objetivando, sobretudo, adquirir um maior conhecimento no assunto. Desse modo, este projeto inclui tanto o estudo, configuração e uso de soluções disponíveis, no caso do servidor de *streaming* de vídeo, quanto o desenvolvimento de uma solução, no caso da aplicação cliente no celular.

Em resumo, a solução criada neste projeto compreende a utilização de um aparelho celular GSM, utilizando a tecnologia GPRS como meio de transmissão de dados, e de um computador como servidor de *streaming* de vídeo. Dessa forma, uma aplicação desenvolvida em Java é executada no aparelho celular para se comunicar com o servidor de *streaming* de vídeo. Enquanto os pacotes de vídeo são recebidos o aplicativo Java irá, em paralelo, mostrar a mídia na tela.

É importante ressaltar, como premissa do projeto, que a comunicação com o servidor de *streaming* de vídeo e o transporte do conteúdo, em si, deve utilizar protocolos padronizados e estabelecidos no mercado. Assim, como os vídeos utilizados também devem ser codificados através de codificadores padronizados.

Outra premissa do projeto é a orientação para o uso de software livre. Tudo o que foi desenvolvido, as ferramentas utilizadas e até mesmo a aplicação servidora utilizam ao máximo software livre. Somente quando não houve alternativas foi usado, no pior caso, um software apenas gratuito. Sendo assim, o custo da solução em termos de software é zero.

Para apresentar o que foi feito, a documentação do projeto foi organizada em capítulos da seguinte forma. Primeiramente, é dada uma visão geral da solução, mostrando os principais componentes do sistema, explicando sucintamente o papel de cada um, e descrevendo todo o trabalho feito durante o projeto. No capítulo seguinte são

explicados os conceitos de base para o entendimento de um servidor de *streaming* de vídeo e para o desenvolvimento de um aplicativo de vídeo em Java para celular. Em seguida são apresentadas com mais detalhes as soluções do servidor de *streaming* de vídeo e do cliente. Por fim, são apresentadas as conclusões deste trabalho.

2 Desenvolvimento

Neste capítulo será apresentado todo o trabalho realizado durante o projeto. Será apresentada a solução de *streaming* de vídeo projeto dividida em fases. Em cada uma das fases serão apresentadas as questões relevantes que envolveram pesquisa, decisão ou desenvolvimento durante o andamento do projeto.

Em uma solução completa de *streaming* podemos destacar três fases principais. A primeira é a produção do vídeo, a segunda é a sua distribuição e a terceira é seu consumo.

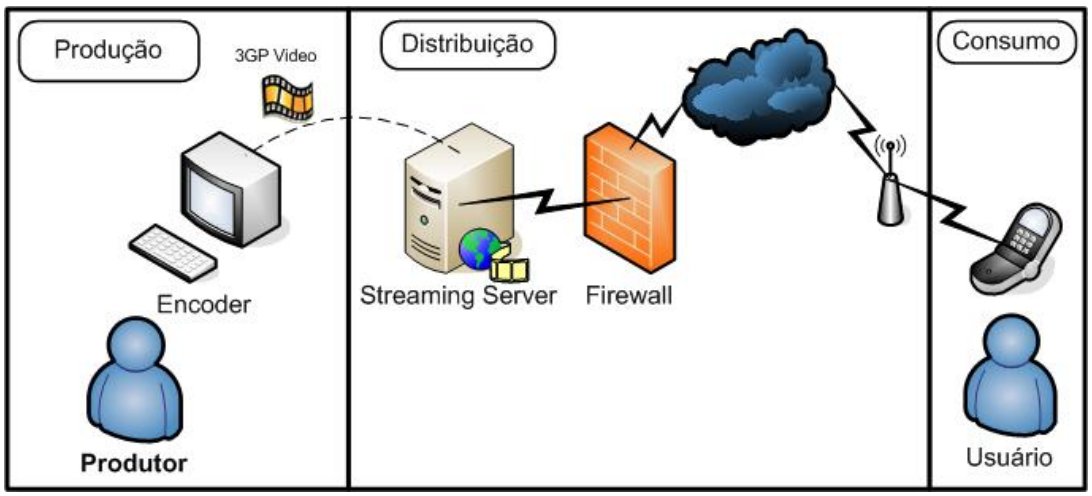

Figura 1 – Esquema geral do sistema.

Para que o vídeo possa ser tocado corretamente no celular este deve estar codificado em um formato que seja reconhecido pelo celular. O padrão mais reconhecido pelos celulares atualmente é o 3GP (*3GPP File Format*). Trata-se de um *container* multimídia, ou seja, um formato de arquivo que pode conter diferentes mídias codificadas através de codificadores padronizados. Definido pelo orgão 3GPP (*3rd Generation Partnership Project*), este *container* é uma versão simplificada do MPEG-4 Part 14, onde as seqüências de vídeo podem ser codificadas em MPEG-4 ou H.263 e as seqüências de áudio podem ser codificadas em AMR-NB (*Adaptive Multi-Rate - Narrow Band*) ou AAC-LC (*Advanced Audio Coding – Low Complexity*).

É importante ressaltar que não foi foco do projeto o estudo aprofundado destes codificadores. O escopo foi limitado em apenas produzir vídeos compatíveis com o formato reconhecido pelo aparelho celular. Sendo assim, o primeiro desafio do projeto foi encontrar um codificador livre capaz de gerar vídeos neste padrão. Durante a busca

foi encontrada uma ferramenta gratuita que é uma interface gráfica para o uso dos principais codificadores livres existentes, englobando ffmpeg [4], x264 [5], etc. Este aplicativo, nomeado de SUPER (*Simplified Universal Player Encoder & Renderer*) [6], é muito simples de usar e possui uma grande variedade de opções para a codificação.

Com os vídeos sendo criados no formato correto, ainda há mais uma etapa na produção de vídeo. Esta etapa adicional consiste em adicionar informações sobre as faixas de vídeo e áudio para que o servidor de *streaming* de vídeo saiba separar, de maneira otimizada, o conteúdo em pacotes para o *streaming* de vídeo.

Estas faixas de informação extra, conhecidas como *hint tracks*, são uma necessidade específica de alguns servidores de *streaming* de vídeo, como no caso do adotado no projeto, *Darwin Streaming Server* [7], e são usadas apenas para o caso de *streaming* via RTP. Sendo assim, outras formas de distribuição, como HTTP, ignoram esta informação. Cada faixa, seja de áudio ou de vídeo, necessita de uma *hint track* para que possa ser realizado o *streaming*. Uma *hint track* especifica, por exemplo, o escala de tempo do RTP e o tamanho máximo de um pacote (MTU), do inglês Maximum Transmission Unit [20]. É importante ressaltar, que as informações contidas em *hint track* podem influenciar diretamente o desempenho de serviço de *streaming* [20].

Houve uma dificuldade em encontrar uma ferramenta livre capaz de fazer a criação de *hint tracks* para os vídeos. Foi encontrado o projeto GPAC [8] que entre seus aplicativos possui o MP4Box, capaz de produzir tais *hint tracks*. Para utilizar a ferramenta é preciso baixar seu código fonte e compilá-lo. Neste projeto, a ferramenta foi compilada em uma máquina com sistema operacional Linux com a distribuição Fedora Core 4. Para realizar a compilação também é preciso baixar, além do código fonte, as bibliotecas extras utilizadas pelo aplicativo, disponíveis também no site do projeto GPAC.

A distribuição de vídeo via *streaming* necessita de um servidor de *streaming* de vídeo. Este servidor além de atender a premissa de ser livre também deveria suportar conteúdos no formato 3GP. Como já mencionado, o servidor de *streaming* de vídeo escolhido foi o *Darwin Streaming Server* (DSS) da empresa Apple Computers. O DSS foi o único servidor de *streaming* de vídeo livre encontrado que suporta conteúdo 3GP. Desenvolvido em C++, o DSS além de possuir código aberto, possui uma excelente

documentação para desenvolvedores. Desta forma, foi possível estudar a arquitetura interna do servidor, seus módulos e até mesmo funções específicas no código. Em um capítulo posterior será descrita esta estrutura e abordadas as características internas do servidor.

Para ter o servidor funcionando existem duas opções: baixar o arquivo binário e instalá-lo diretamente ou baixar o código fonte, compilá-lo e instalá-lo. No projeto foi escolhida a segunda opção, pois permite um maior conhecimento sobre o código fonte. Desse modo, foi baixado o código fonte do site de desenvolvimento do DSS. Este código fonte foi compilado e está sendo executado no sistema operacional Linux com a distribuição Fedora Core 4. Uma observação importante é sobre a versão do compilador a ser usada. Utilizando a versão 4.0 do compilador gcc eram gerados erros e não foi obtido sucesso. Com uma versão anterior, no caso a 3.2, o código pôde ser compilado com sucesso.

Depois de compilado, o servidor deve ser iniciado passando como parâmetro um arquivo de configuração do tipo XML. Uma das configurações a serem feitas neste arquivo é o diretório onde os arquivos de mídia estão localizados no servidor. Outra configuração importante é sobre onde serão gerados os arquivos de *log* do servidor.

O *Darwin Streaming Server* também disponibiliza uma aplicação *web* de gerenciamento. Através desta interface, é possível iniciar e parar o serviço de *streaming* de vídeo, e alterar as principais configurações. Esta aplicação é acessada através da porta 1209.

Depois de iniciado, o servidor passa a escutar na porta 554, referente ao protocolo de aplicação RTSP, e no caso do projeto, responde ao endereço rtsp://joa.gta.ufrj.br. O sistema de *firewall* do laboratório GTA (Grupo de Teleinformática e Automação) foi configurado de modo que as portas utilizadas pelos protocolos RSTP e RTP fossem liberadas para a máquina joa.

Com as configurações corretas no servidor de *streaming* e no *firewall*, o próximo passo, foi então testar em celulares. Este teste inicial foi feito utilizando o aplicativo de *streaming* de vídeo nativo do celular, no caso um Sony Ericsson K750, pois o objetivo era testar se era possível obter um fluxo RTSP / RTP no celular. Desta forma, através do *browser* do aparelho é possível digitar a URL de um serviço de *streaming* de vídeo para

assistí-lo. Assim, com o servidor de *streaming* de vídeo funcionando e utilizando um aplicativo já confiável como cliente, problemas que possam aparecer ficam isolados na comunicação entre o celular e a rede do GTA.

O primeiro teste realizado utilizou um aparelho na rede da operadora de telefonia Tim. O acesso foi feito através de sua APN (*Access Point Name*) de GPRS (*General Packet Radio Service*). Uma APN leva informação ao SGSN (*Serving GPRS Support Nodes*) e ao GGSN (*Gateway GPRS Support Nodes*) sobre como os pacotes devem ser roteados. As APNs GPRS geralmente são configuradas nas operadoras para ter acesso direto a *internet*, ou seja, sem passar por *gateways* especias como é o caso dos *gateways* WAP (Wireless Application Protocol).

Mesmo utilizando a APN de GPRS não foi possível estabelecer um fluxo de vídeo. Possivelmente as portas necessárias para o fluxo RTP e RTCP estão sendo bloqueadas na operadora em questão.

No segundo teste, utilizou-se um aparelho da operadora de telefonia Claro. Novamente o acesso foi feito através de sua APN de GPRS. Desta vez, todas as conexões foram estabelecidas com sucesso. Com isso, a próxima fase do projeto consistiu em desenvolver o aplicativo cliente de *streaming* de vídeo Java no celular.

Primeiramente, como ambiente de desenvolvimento foi escolhida a ferramenta Eclipse que já vem com a distribuição *Fedora Core 4*. Trata-se de uma IDE (*Integrated Development Environment* - Ambiente Integrado de Desenvolvimento) de código aberto e de maior popularidade para o desenvolvimento de aplicações Java.

Além do Eclipse, é preciso ter uma ferramenta que dê suporte às especificidades do J2ME, contendo as APIs específicas e também emuladores de celular para os testes preliminares. Esta ferramenta é o *Wireless Toolkit* [9] fornecido gratuitamente pela Sun.

Sendo assim, toda a parte de criação, edição e organização do código foi feita utilizando o Eclipse. Já a compilação, “empacotamento” e a execução dos testes preliminares foram feitos utilizando o *Wireless Toolkit*.

Contudo, estes testes preliminares são muito limitados, pois o emulador não reproduz exatamente o ambiente real de execução, pois a implementação da Sun é diferente da Sony Ericsson, contendo codificadores diferentes, por exemplo. Além disso, não há suporte para o teste de aplicações que envolvam *streaming* de vídeo, o que

limitou os testes preliminares a apenas um teste da interface gráfica e da navegação dentro do aplicativo.

Dentre as centenas de modelos de celulares existentes hoje no mercado brasileiro, muito poucos suportam *streaming* de vídeo na sua implementação da Máquina Virtual Java. Sendo assim, restaram poucas opções e o ambiente de execução escolhido foi o celular do fabricante Sony Ericsson modelo K750 (Figura 2).

Figura 2 – Sony Ericsson K750.

Trata-se de um aparelho relativamente popular entre os modelos de alta gama que além de possuir a implementação de muitas APIs opcionais do J2ME, possui grande qualidade nas implementações, ou seja, um pequeno número de erros de implementação (*bugs*). Isto o faz uma excelente escolha para o desenvolvimento de aplicações J2ME.

O trabalho realizado no cliente consiste, então, no desenvolvimento de uma aplicação J2ME que se comunica com o *Darwin Streaming Server* utilizando-se da API de multimídia (MMAPI) para realizar o *streaming* de vídeos e sua visualização no aparelho celular. Além disso, o aplicativo cliente também possuirá uma tela de menu onde os vídeos poderão ser selecionados e disponibilizará controles sobre o vídeo quando este estiver em execução.

Neste capítulo foi apresentado, como um todo, o desenvolvimento realizado no projeto. É preciso, então, explicar a teoria envolvida no projeto. Assim, no próximo capítulo, na primeira seção, serão apresentados os principais protocolos envolvidos em um *streaming* de vídeo e na segunda seção serão apresentadas as tecnologias para o desenvolvimento de aplicações multimídia para celular.

3 Fundamentação Teórica

Neste capítulo serão descritos os principais conceitos do projeto. Estes conceitos dividem-se em duas áreas de conhecimento, sendo a primeira relativa aos protocolos e a segunda ao aplicativo cliente de *streaming* de vídeo.

Desse modo, a primeira seção tratará dos protocolos envolvidos em um *streaming* de vídeo. Sendo assim, foram detalhados os protocolos RTSP [10] e RTP [12]. O RTSP é responsável pelo comando de um *streaming* de vídeo, sendo conhecido como o "controle-remoto" da *Internet*. Já o RTP é responsável pelo transporte do conteúdo multimídia. A intenção é prover conhecimento geral sobre o funcionamento de cada protocolo para que se tenha um entendimento dos mecanismos de base de um *streaming* de vídeo. Para uma especificação mais detalhada se recomenda consultar diretamente as especificações que definem cada protocolo.

Na segunda seção, os conceitos para programação de aplicativos são introduzidos. Primeiro, é feita a exposição sobre o J2ME [13], suas configurações e perfis e, em seguida, é abordada a API específica para aplicativos multimídia, a JSR-135 [15].

3.1 Protocolos

3.1.1 RTSP

O RTSP (*Real Time Streaming Protocol*) é um protocolo de aplicação utilizado para controle da entrega de dados em tempo real [10]. Nesta seção, serão expostos os objetivos do protocolo e, em seguida, seu funcionamento geral será explicado. Depois, será explicitado seu relacionamento com outros protocolos, posicionando-o, assim, dentro de uma solução completa de *streaming* de vídeo. Em seguida, será realizado um aprofundamento técnico com ênfase nos principais métodos do protocolo e um exemplo de funcionamento.

Como propósito, o RTSP tem o estabelecimento e controle de um, ou vários, fluxos de áudio e vídeo sincronizados [10]. É importante esclarecer que o RTSP não é responsável pelo transporte do conteúdo da mídia em si. Para realizar o transporte, o RTSP se relaciona com outro protocolo, aspecto que será tratado mais à frente ainda nesta seção. Cabe ressaltar neste momento, que o transporte é feito pelo RTP, protocolo que será apresentado a seguir.

Para entender melhor o RTSP se pode pensar nele como se fosse um “controle de vídeo cassete” [11] para servidores multimídia na *Internet*. É utilizando o RTSP que um usuário irá controlar o que deseja assistir, com comandos equivalentes a “play”, “pause”, “stop”, etc. Como casos de uso, podem ser imaginados quaisquer situações em que uma aplicação faça um *streaming* de um vídeo, como um aluno vendo um vídeo de aula em seu computador pessoal, um segurança monitorando câmeras, um torcedor assistindo um jogo no celular, etc.

O RTSP funciona através de trocas de mensagens, requisições e respostas, entre o cliente e o servidor. Seu funcionamento é independente do protocolo que fará o transporte da mídia. Não existe o conceito de conexão RTSP, ao invés disso, o servidor identifica cada sessão através de um identificador único. Isto porque o servidor precisa guardar o estado dos clientes. Existem três estados possíveis, e o recebimento de mensagens implica na navegação entre eles. A Figura 3 é uma adaptação de [11] que mostra de maneira simplificada este conceito:

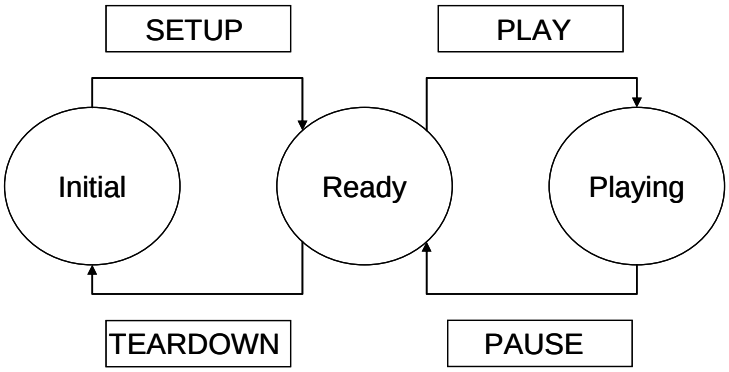

Figura 3 – Estados do cliente RTSP.

É preciso descrever um processo comum de interação cliente e servidor para um melhor entendimento global do protocolo. O processo é bem simples do ponto de vista do usuário. Um usuário digita uma URL em seu aplicativo de vídeo, do tipo rtsp://servidor.com/video, e o servidor irá passar a transmitir o vídeo para o cliente, ficando atento para novos comandos. Sustentando este processo simples, há várias trocas de mensagens de diferentes protocolos. Essas trocas serão mostradas no anexo do

projeto através de uma tabela que analisa um arquivo de *dump* feito na rede do GTA durante o *streaming* de vídeo para um celular.

O RTSP tem relacionamento com outros protocolos, ou seja, utiliza-se deles na execução de uma função, ou possui características sintáticas semelhantes. O primeiro a ser descrito é com o protocolo de transporte TCP. Para que as mensagens RTSP sejam trocadas entre cliente e servidor, uma conexão *full-duplex* TCP é estabelecida. Em seguida o RTSP precisa se relacionar com outro protocolo de transporte de dados, para que a mídia seja transportada.

Na grande maioria das vezes, o transporte dos dados de mídia é feito através do protocolo RTP. Este protocolo será abordado com detalhes em outra seção, mas cabe adiantar um pouco desta relação. O RTP irá criar uma conexão UDP unidirecional para o transporte da mídia e o RTCP cria duas conexões UDP em sentidos opostos para controle de sincronismo no cliente e informação de perda de pacotes para o servidor [11].

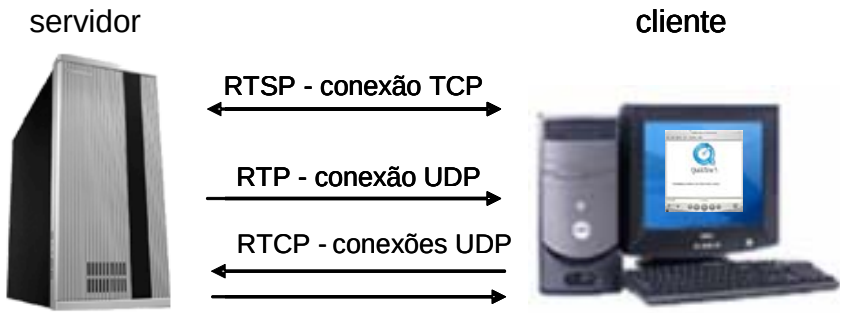

Figura 4 - Conexões cliente/servidor.

É possível também que o transporte possa ser realizado por outros protocolos. Um exemplo é o protocolo proprietário da RealNetworks, o RDT (*Real Data Transport*) [19]. Em termos de conexões estabelecidas, a diferença ocorre nas conexões UDP onde as duas são unidirecionais uma no sentido do servidor e outra no sentido do cliente. A conexão no sentido do servidor serve para o cliente requisitar o re-envio de pacotes perdidos [11].

Outro protocolo ligado ao RTSP é o HTTP. Este relacionamento se dá porque o acesso a uma seqüência de mídia na maioria das vezes vem através de uma página HTTP. Em seguida, a descrição do conteúdo pode vir via HTTP ou RTSP.

Além disso, existe uma similaridade sintática e operacional entre o RTSP e o HTTP/1.1, que foi feita intencionalmente [10]. Isto foi criado para que mecanismos de extensão do HTTP possam ser adicionados ao RTSP. Contudo, a especificação [10] destaca algumas diferenças, as quais são:

- O RTSP introduz novos métodos e tem um identificador de protocolo diferente;
- Um servidor RTSP precisa armazenar o estado do cliente, na grande maioria dos casos, em oposição à natureza sem estados do HTTP;
- Ambos, o cliente e o servidor, podem submeter requisições RTSP;
- Dados são transmitidos por um protocolo diferente;

O último protocolo com o qual o RTSP mantém relacionamento estrito é o SDP (*Session Description Protocol*). Este protocolo é responsável apenas por descrever as sessões multimídia para o RTSP.

Existem muitos parâmetros e especificações detalhados na RFC 2326 que define o protocolo RTSP. Neste trabalho, o foco será apenas sobre o formato e tipos de requisições e respostas. Toda requisição, que pode ser enviada do cliente para o servidor ou vice-versa, tem que conter o método a ser aplicado na fonte, um identificador da fonte e a versão do protocolo em uso.

Nesse sentido, os métodos do protocolo possuem um sentido definido do cliente para o servidor, ou vice-versa, ou ambos. Além disso, os métodos são classificados como obrigatório, recomendado ou opcional. A Tabela 1, adaptada de [10] relaciona os métodos e suas características.

Método	Direção	Requisito
DESCRIBE	C->S	recomendado
ANNOUNCE	C->S, S->C	opcional
GET_PARAMETER	C->S, S->C	opcional
OPTIONS (1)	C->S	obrigatório
OPTIONS (2)	S->C	opcional
PAUSE	C->S	recomendado
PLAY	C->S	obrigatório
RECORD	C->S	opcional
REDIRECT	S->C	opcional
SETUP	C->S	obrigatório
SET_PARAMETER	C->S, S->C	opcional
TEARDOWN	C->S	obrigatório

Tabela 1 – Métodos e características.

Uma breve descrição dos métodos se faz necessária para entendimento das funcionalidades do RTSP. Desse modo, o primeiro método a ser descrito é o OPTIONS. Este método é utilizado para perguntar, geralmente ao servidor, quais métodos são suportados. O uso deste método é recomendado quando, por exemplo, o cliente deseja utilizar um método no servidor que não é obrigatório.

O método DESCRIBE requisita a descrição de uma apresentação ou mídia no servidor identificada por uma URL. Este método é utilizado geralmente no estabelecimento de uma sessão. O método opcional ANNOUNCE pode se ter dois propósitos. Se enviado do cliente para o servidor, configura a descrição de uma apresentação ou mídia identificada por uma URL. Se enviado do servidor para o cliente, atualiza a descrição de uma sessão em tempo real.

Depois de saber das capacidades do servidor, através do método OPTIONS, e dos parâmetros da apresentação que se deseja ver, através do método DESCRIBE, o método SETUP pede ao servidor que especifique o mecanismo de transporte a ser utilizado. Pode-se então utilizar o método PLAY no cliente para requisitar o começo da

transmissão da mídia. É importante ressaltar que o PLAY só pode ser requerido depois de uma resposta do SETUP para que o mecanismo de transporte esteja acordado.

Para interromper temporariamente o fluxo de mídia sendo transportado, se deve utilizar o método PAUSE. Contudo, para interromper definitivamente o fluxo de mídia o método utilizado deve ser o TEARDOWN.

Os métodos GET_PARAMETER e SET_PARAMETER servem, respectivamente, para obter o valor de um parâmetro e definir o valor de um parâmetro. Ambos são bidirecionais, ou seja, do cliente para o servidor ou do servidor para o cliente.

Os últimos métodos descritos são o REDIRECT e o RECORD. Como o nome sugere, o método REDIRECT informa ao cliente outro servidor para que ele se conecte. O método RECORD é utilizado pelo cliente para requerer ao servidor a gravação de uma sessão de mídia em uma URI.

Depois de receber uma requisição, com qualquer um dos métodos citados, uma mensagem de resposta deve ser enviada. Esta mensagem tem como informação principal a primeira linha, chamada de *status-line*. A *status-line* é composta de um código de resposta e uma frase explicativa. Entre os códigos de respostas a maioria é adotada do HTTP/1.1. Estes códigos são números inteiros e são agrupados em centenas de modo que o primeiro algarismo traga alguma significância sobre o código. Isto faz com que muitos códigos com valor de dezena alta não sejam usados. Sendo assim, novos códigos exclusivos do RTSP são declarados em cada centena somente a partir da quinta dezena para evitar conflito. Cada código possui uma frase explicativa associada.

São exemplos de códigos oriundos do HTTP, o código 200 "*OK*" e o código 404 "*Not Found*". Como exemplos de códigos exclusivos do RTSP, podem ser citados o código 454 "*Session Not Found*" e o código 551 "*Option not supported*".

3.1.2 RTP

O protocolo RTP (*Real-time Transport Protocol*) foi definido pela IETF em 2003 através da RFC 3550 [12]. O RTP é um protocolo que provê serviços completos de entrega de dados em tempo real [12]. Estes serviços incluem identificação do *payload*, numeração de seqüência, marcação temporal e monitoramento da entrega.

Embora possa ser utilizado com outros protocolos de transporte de mais baixo nível, na grande maioria das implementações o RTP utiliza-se do protocolo UDP. Desta forma, entende-se que o UDP complementa a funcionalidade de transporte do RTP.

A RFC 3550 divide o RTP em duas partes, o *Real-Time Transport Protocol* (RTP) e o *RTP Control Protocol* (RTCP). A primeira parte é responsável por transportar os dados e a segunda parte deve monitorar a qualidade do serviço e trazer informações sobre os participantes de uma sessão.

Quanto à qualidade do serviço (QoS), é importante ressaltar que o RTP não possui nenhum mecanismo que garanta a qualidade do serviço. Contudo, através do RTCP é possível obter as informações necessárias para uma implementação de mecanismos de suporte à QoS.

Um cenário interessante presente no documento de definição [12] é a realização de uma áudio-conferência. Através da descrição do comportamento do protocolo durante uma áudio-conferência é possível identificar seus serviços em uso e, assim, ter um melhor entendimento.

No início da sessão exemplo, são alocados um endereço *multicast* e um par de portas, no qual uma das portas é usada para transmissão do dado de áudio e a outra é usada para o controle. Cada participante envia então pequenos pedaços de seu áudio e cada pedaço é precedido de um cabeçalho RTP. Este conjunto, cabeçalho RTP mais dado forma um pacote UDP.

No cabeçalho RTP existem informações importantes como a codificação de áudio utilizada, permitindo uma flexibilização na sua escolha até mesmo durante uma sessão. Outras informações necessárias são o tempo e o número seqüencial de cada pacote. O número de seqüência garante a reconstrução correta do que foi enviado e a informação de tempo permite tanto a reprodução no tempo correto, quanto a sincronização com outras sessões, por exemplo, uma sessão de vídeo que esteja acontecendo em paralelo.

A numeração seqüencial dos pacotes também serve para que o receptor tenha idéia de quantos pacotes perdeu e, a partir disso, como está a qualidade da recepção. A informação sobre a qualidade da recepção é enviada periodicamente através de relatórios na porta destinada para o controle da sessão.

Agora que já foi apresentada uma noção geral do funcionamento, maiores detalhes do protocolo serão mostrados. A maneira mais adequada de fazer isso é apresentar o pacote RTP e descrever a função dos principias campos que o compõe. A Figura 5, retirada de [12], representa um pacote RTP.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronization source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
:          contributing source (CSRC) identifiers               |
|=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                                                               |
|                                                               |
|                     RTP Packet Payload                        |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figura 5 – Pacote RTP.

- Versão (V) – Os primeiros dois bits indicam a versão do protocolo utilizada. Quando compatível com a RFC3550 seu valor é dois.
- *Payload type* (PT) – São usados sete bits para identificar o tipo do *payload*, ou seja, a codificação utilizada na mídia.
- *Sequence number* – São usados dezesseis bits para seqüencialmente identificar cada pacote.
- *Timestamp* – São usados trinta e dois bits para informar o instante em que foi amostrado o primeiro byte dos dados da mídia.
- *Syncronization Source* (SSRC) - São usados trinta e dois bits para identificar a fonte da seqüência de pacotes RTP. Esta identificação é importante para que a identificação da fonte fique independente dos protocolos inferiores de rede.
- *RTP Packet Payload* – Dados da mídia que está sendo transmitida.

Além do transporte da mídia há também um mecanismo para fazer o controle da transmissão. Nesse sentido, o RTCP possui quatro funções. A primeira é prover relatórios sobre a qualidade do serviço da transmissão. A segunda é manter um identificador persistente para uma fonte RTP, conhecido como CNAME. Este identificador persistente é necessário, pois o SSRC pode mudar em caso de conflito ou reinício do programa. Também é necessário para que os receptores possam acompanhar sessões relacionadas, por exemplo, áudio e vídeo.

A terceira função é controlar a taxa com que os participantes enviam seus relatórios RTCP. Como última e opcional, a quarta função é transmitir informação para controle de sessão.

Para desempenhar estas quatro funções o protocolo RTCP possui cinco tipos de pacotes.

- SR – Sender report, provê estatísticas de transmissão e recepção de participantes que enviam mídia.
- RR – Receiver report, provê estatísticas de recepção de participantes que recebem mídia.
- SDES – Source Description, possui informação sobre a fonte incluindo o CNAME.
- BYE – Indica o fim de uma participação.
- APP – Funções específicas de aplicações.

3.2 Aplicação Móvel

3.2.1 J2ME

A *Java 2 Platform Micro Edition* (J2ME) é justamente uma plataforma Java desenvolvida para pequenos dispositivos. Estão contidos no grupo de dispositivos alvo do J2ME desde aparelhos celulares, passando por PDAs e *Smart Phones*, até *set-top-boxes* e outros dispositivos embarcados.

Para atender a toda essa gama de dispositivos que possuem características distintas, o J2ME é subdividido em configurações e perfis. Como primeira subdivisão, existem duas configurações que segmentam os dispositivos sobre três características [13]: tipo e disponibilidade de memória, tipo e velocidade do processador e tipo da conexão disponível com a rede.

A configuração CLDC (*Connected Limited Device Configuration*) é direcionada para dispositivos com restrições nas três características citadas, ou seja, pouca memória, baixo poder de processamento e conexão limitada com a rede. Como dispositivos típicos, destacam-se justamente os celulares e PDAs. Já a configuração CDC (*Connected Device Configuration)* atende aos dispositivos com maiores capacidades de memória e processamento, situados entre os computadores pessoais e os dispositivos CLDC. Como exemplos de equipamentos CDC, pode-se citar *set-top-boxes*, vídeo games e alguns PDAs mais potentes.

Figura 6 – Configurações e perfis J2ME.

Neste trabalho o foco está todo voltado para a configuração CLDC. Sendo assim, a divisão em perfis da CDC não será abordada. Dentro da CDLC existe apenas um perfil, conhecido como MIDP (*Mobile Information Device Profile*).

O CLDC possui duas versões a 1.0 e 1.1 definidas pela JSR (*Java Specification Request*) 030 e JSR 139, respectivamente. São definidas por essas especificações as classes básicas do Java, em geral com grande semelhança ao J2SE, plataforma Java destinada a computadores pessoais. Em união ao CDLC, o MIDP que também possui duas versões 1.0 (JSR 037) e 2.0 (JSR 118), acrescenta os recursos de interface, armazenamento, rede e o ambiente MIDlet.

Juntos, o CDLC e o MIDP, formam o ambiente de execução J2ME. Isto é, os recursos computacionais mínimos que uma máquina virtual deve fornecer para que se possa rodar o aplicativo Java simples.

Porém, hoje existe uma demanda por parte do mercado de aplicativos que utilizem cada vez mais recursos, como apresentações multimídia e conexão *bluetooth*.

Ao identificar uma necessidade por determinadas funcionalidades extras, fabricantes e instituições podem se juntar para criar novas especificações (JSR) para disponibilizar mais recursos aos aplicativos Java.

É importante ressaltar que diferentemente do CLDC e do MIDP a implementação de JSR extras é opcional. Sendo assim, estas são conhecidas como pacotes opcionais do J2ME. Os exemplos de JSR comumente implementadas em celulares são a JSR 135, *Mobile Media API*, e a JSR 82, *Java APIs for Bluetooth*. Para este projeto, foi utilizada a JSR 135 no desenvolvimento da aplicação, e, portanto, esta será descrita com detalhes na seção seguinte.

3.2.2 JSR 135

A *Mobile Media API*, também conhecida como MMAPI, é um pacote opcional do J2ME destinado a fornecer funcionalidades de multimídia aos aplicativos. Assim como o J2ME, que pode ser visto como uma versão simplificada do J2SE, a MMAPI também tem seu correspondente de mais alto nível no J2SE que é o *Java Media Framework* (JMF).

Como uma forma de encorajar a adoção por parte dos fabricantes, a MMAPI foi projetada para ser independente de protocolos ou formatos [14]. Sendo assim, cada fabricante pode escolher os protocolos que deseja implementar e também os formatos que irá suportar.

Esta API é fundamentada sobre quatro classes principais que correspondem a uma abstração de alto nível das possíveis apresentações multimídia em um dispositivo J2ME. A Figura 7 traz uma visão geral sobre os papéis e relacionamentos das mesmas.

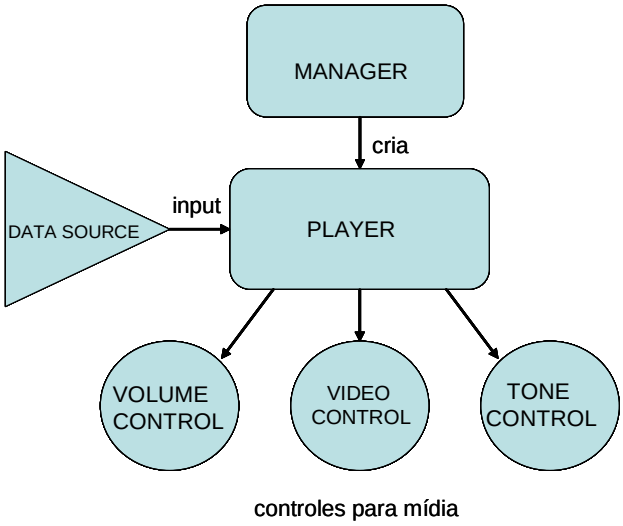

Figura 7 – Estrutura de classes da MMAPI.

O processamento de mídias pode ser dividido em duas partes principais: a aquisição e entrega das mídias e no processamento das mídias para a apresentação [15]. Correspondendo a estas partes, existem as duas classes principais da MMAPI que são a *DataSource* e o *Player*, respectivamente.

Sendo assim, a classe *DataSource* é responsável por capturar os dados da mídia, seja através de um sistema de arquivos, de um protocolo padrão, ou de algum mecanismo proprietário. Desta forma, toda a complexidade de acesso à mídia fica encapsulada nesta classe. Através de métodos específicos, o *DataSource* permite que o *Player* possa pedir o acesso a mídia.

O *DataSource* é uma classe abstrata, ou seja, precisa ser estendida e ter certos métodos obrigatoriamente implementados. Isto possibilita ao desenvolvedor criar seus próprios mecanismos de acesso à mídia. Contudo, na prática, o desenvolvimento de *DataSources*, por ser complexo, é realizado apenas pelos fabricantes de celular.

Com a mídia acessível, cabe ao *Player* processá-la, decodificá-la e colocá-la em uma saída. Esta classe ainda atua sobre as classes de controle de mídia, como vídeo e volume, e ainda disponibiliza métodos para o controle direto da mídia, como *play* e *stop*.

Os métodos de controle do *Player* o fazem navegar entre seus estados. Na Figura 8 tem-se esta representação e a Tabela 2 possui os estados e suas características.

Estado	Característica
UNREALIZED	O objeto *Player* foi instanciado na memória. Ainda não foi alocado nenhum recurso.
REALIZED	O *Player* fez a requisição do recurso de mídia. Já possui a mídia acessível, através de um protocolo ou do sistema de arquivo.
PREFETCHED	O *Player* fez requisição de outros recursos, como controle de vídeo e áudio do dispositivo, e a mídia já está na memória, pronta para ser consumida.
STARTED	O *Player* já começou a tocar a mídia.
CLOSED	O *Player* se liberou os recursos e não pode ser mais tocado.

Tabela 2 – Estados e características

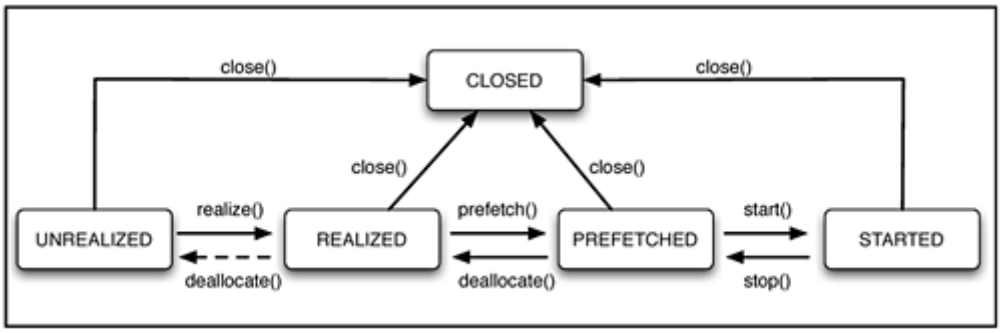

Figura 8 – Estados do Player.

Falta ainda descrever a classe *Manager* e as classes de controle que aparecem no esquema (Figura 7). A classe *Manager* é responsável por criar um *Player* através de um *DataSource*. A classe *Manager* é uma *factory* estática, ou seja, uma classe que cria outras classes e é única durante a execução. Para simplificar o desenvolvimento, a classe *Manager* também possibilita criar *Players* através de URLs, caminho local, ou *InputStreams*, ao invés de passar um *DataSource*.

Por fim, as classes de controle são classes que podem ser requeridas pelo *Player* para executar funções. Neste projeto, as classes de controle requeridas são a *VolumeControl* e a *VideoControl*, respectivamente para controle sobre o volume do som que sai para os auto-falantes e para controle sobre o vídeo que está na tela.

Neste capítulo foram apresentados os fundamentos teóricos do projeto em duas seções. A documentação da base teórica além de fazer parte do processo de aprendizado permite que sejam feitos aprofundamentos técnicos. Nesse sentido, a seguir serão apresentados detalhes da solução tanto na parte do servidor de *streaming* de vídeo quanto no aplicativo cliente.

4 Detalhamento da Solução

Em uma solução completa de *streaming* de vídeo, como já citado anteriormente, estão envolvidos principalmente dois tipos de aplicação, uma cliente e outra servidora. Nas seções seguintes serão mostrados, respectivamente, os detalhes de cada uma destas aplicações.

4.1 Cliente

Para a aplicação cliente foi desenvolvido um software em J2ME utilizando a API multimídia definida pela JSR-135. O código completo do aplicativo está em anexo.

4.1.1 Especificação do cliente

A aplicação cliente foi desenvolvida para atender aos requisitos básicos de um aplicativo de visualização de vídeo. Dentre estes requisitos, os requisitos funcionais são derivados do diagrama de casos de uso da aplicação que é mostrado a seguir.

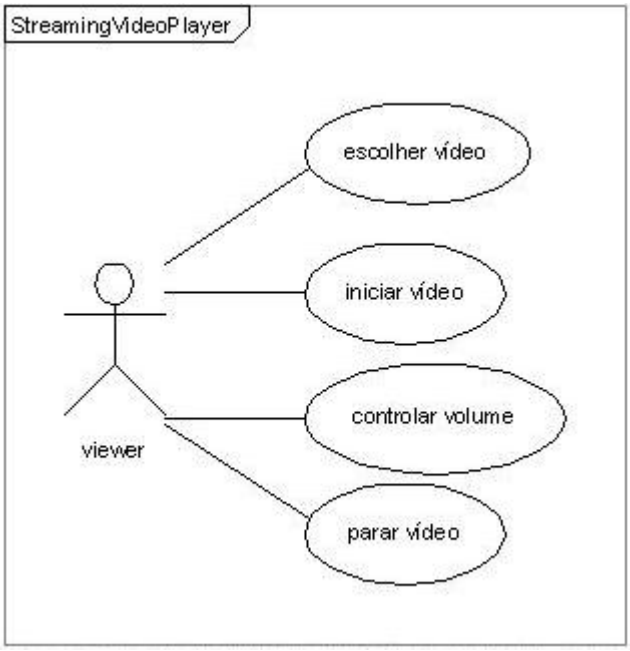

Figura 9 – Diagrama de Casos de Uso.

Além dos requisitos funcionais, representados pelo diagrama acima, uma boa usabilidade foi também considerada como um requisito do produto. Como referência para o estudo de usabilidade, foi utilizado um documento de boas práticas da Nokia [16], empresa considerada referência em usabilidade de celular.

Este documento foi escolhido, pois a série de aparelhos Nokia a qual se destina, Serie 60, possui uma interface de comando semelhante ao modelo K750. Sendo assim, em ambos a navegação é feita principalmente utilizando-se as duas teclas chamadas de *soft-keys* e o *joystick*, mostradas na Figura 10.

Figura 10 – Disposição das teclas principais do K750

A primeira orientação dada sobre a navegação é de sempre localizar o usuário na aplicação, ou seja, o usuário deve perceber quais são os comandos possíveis e deduzir o resultado esperado ao se executar cada comando. Além disso, as *soft-keys* devem sempre possuir o rótulo do comando que executam. Outra orientação é adequar as funcionalidades à *soft-key* correta. Sendo assim, funcionalidades do tipo "voltar", "cancelar" e "sair", devem ser colocadas para a *soft-key* direita, enquanto que as funcionalidades do tipo "executar", "escolher" e "avançar", devem ser colocadas na *soft-key* esquerda.

Considerando todas estas orientações, foi projetado um menu inicial da aplicação, onde o usuário tem a opção de escolher entre as opções de vídeo para assistir ou sair da aplicação. O vídeo a ser tocado é escolhido através de um cursor utilizando-se o *joystick* e é iniciado através da *soft-key* esquerda, rotulada como "Play". Para sair da aplicação deve ser selecionada a *soft-key* direita, rotulada como "Exit".

Escolhida a opção "Play", o vídeo selecionado é tocado em outra tela. Esta tela possui como opções a *soft-key* esquerda, rotulada como "On/Off", e a *soft-key* direita, rotulada como "Stop". A opção "On/Off" troca o estado do volume do vídeo, ou seja, se estiver ligado, ele é desligado, e vice-versa. A opção "Stop" pára a execução do vídeo e retorna para a tela de menu inicial. A Figura 11 ilustra o desenho das telas citadas.

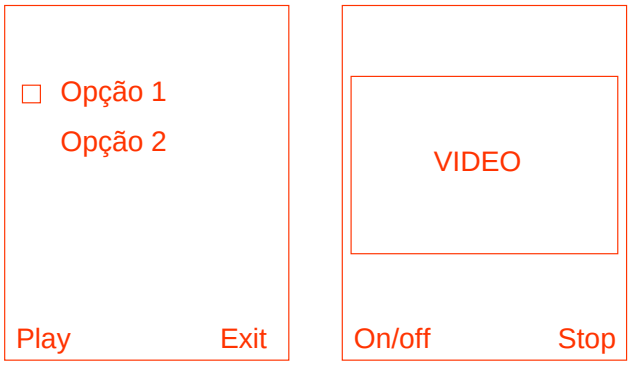

Figura 11 – Esquema de telas do aplicativo.

Outro ponto importante também citado no documento [16] é manter o usuário informado do resultado de suas ações, principalmente quando estas tomam tempo de execução. Sendo assim, entre a ação do usuário de apertar “Play” e o vídeo começar a ser tocado, é mostrada a mensagem de conforto “Carregando...”.

4.1.2 Arquitetura do cliente

A aplicação cliente é composta de três classes, VideoMIDlet, VideoCanvas e Resources. Estas se relacionam entre si e com as outras classes e interfaces J2ME conforme o diagrama de classes representado na Figura 12.

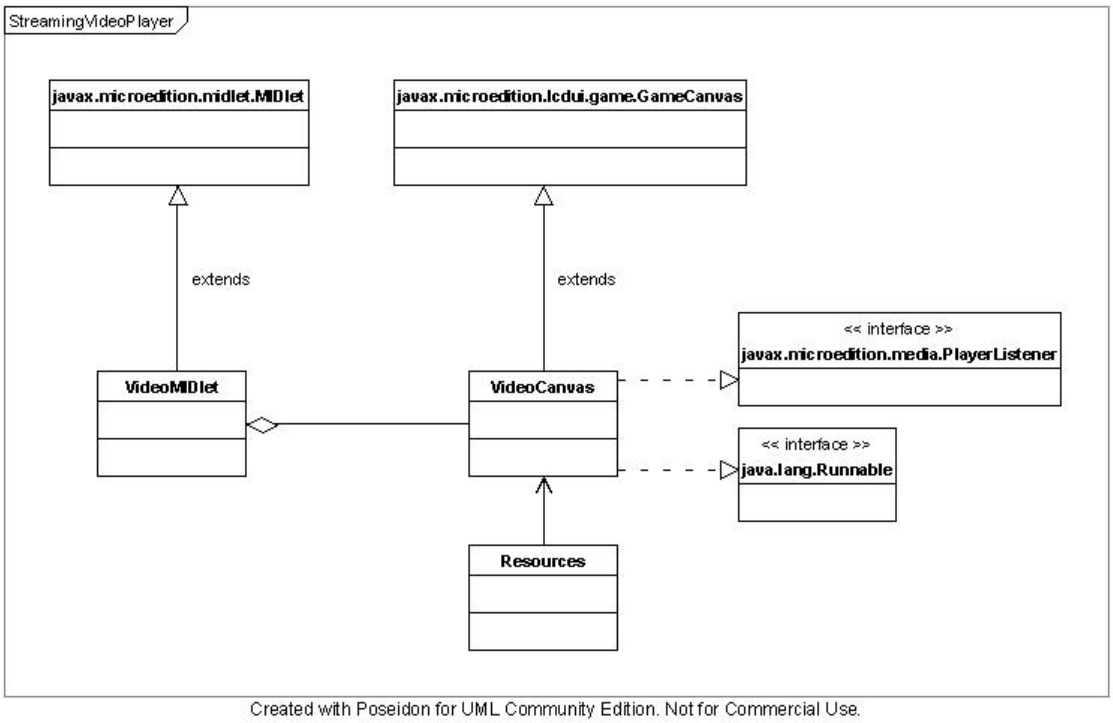

Figura 12 – Diagrama de classes

A classe VideoMidlet é a classe principal do aplicativo J2ME. Ela estende a classe MIDlet e implementa os métodos abstratos necessários para iniciar, parar e finalizar uma aplicação J2ME. Quando uma classe MIDlet é iniciada, primeiramente é chamado seu construtor. O construtor da classe VideoMidlet cria uma nova instância da classe VideoCanvas, passando como parâmetro o objeto *Display*, que dá acesso à tela do celular. O trecho de código a seguir é o método construtor da classe VideoMidlet.

```
public VideoMidlet() {
        videoCanvas = new VideoCanvas(Display.getDisplay(this));
        videoCanvas.setFullScreenMode(true);
}
```

Depois do construtor toda a classe MIDlet tem o seu método *startApp()* chamado. A implementação deste método na classe VideoMidlet chama o método que inicia a classe VideoCanvas (*start()*) e atribui à variável de instância o objeto VideoMidlet em questão. Esta atribuição é necessária, pois o aplicativo faz uso do

padrão de desenvolvimento *Singleton* [17]. Neste padrão, apenas uma instância da classe é feita durante toda a execução e para acessar esta instância deve ser criado também um método que a retorna, no caso o método *getInstance()*. O trecho a seguir traz a implementação do método *startApp()*.

```
public void startApp() throws MIDletStateChangeException {
        instance = this;
        videoCanvas.start();
}
```

A classe VideoCanvas é a mais importante, pois ela é a responsável pela parte gráfica, pela interação com o usuário e pelo tratamento do vídeo. A lógica desta classe está baseada em estados e a navegação entre os estados. Sendo assim, esta classe possui três estados, o MENU_STATE, o PLAYING_STATE e o ERROR_STATE. Conforme o estado atual, a classe VideoCanvas pinta a tela da forma apropriada. Da mesma forma, conforme o estado, a tecla pressionada é tratada do modo adequado. Neste sentido, através de comandos de teclas, o usuário pode mover entre os estados e executar o que deseja.

Quando é iniciada pela classe VideoMidlet, a classe VideoCanvas se coloca como a classe que deve ser mostrada na tela e chama o método *repaint()* responsável por pintar a tela. O método *repaint()* chama internamente o método *paint*() que deve necessariamente estar implementado em uma classe que estende a classe *GameCanvas*.

Este método *paint()* possui um trecho comum a ambos os estados e uma parte dependente do estado do aplicativo. A parte comum executa funções compartilhadas, como limpar a tela e a parte dependente é necessária, pois conforme o estado a tela é desenhada de maneira diferente. O trecho a seguir, retirado do código, mostra esta estrutura.

```
public void paint(Graphics g) {
        …
        // clear the screen (paint it white):
        g.setColor(0xffffff);
```

```
                g.fillRect(x, y, w, h);
                …
                if (this.state == Resources.PLAYING_STATE) {
                        ….
                }
                else if (this.state == Resources.MENU_STATE) {
                        …
                }
        }
```

Outro método que deve ser implementado da classe GameCanvas é o *keyPressed()*. Este método recebe o código de uma tecla quando esta é acionada. Dependendo do estado do aplicativo cada tecla possui uma ação, esta ação pode fazer com que o aplicativo navegue entre os diferentes estados ou execute suas funcionalidades, como iniciar um vídeo, ou pará-lo. O trecho de código a seguir mostra a estrutura do método *keyPressed()*.

```
Public void keyPressed(int keyCode) {
                if (state == Resources.MENU_STATE) {
                        if ((keyCode == Resources.RIGHT_SOFT_KEY) || (keyCode ==
Resources.END_KEY)) {
                                …
                        } else if (keyCode == Resources.LEFT_SOFT_KEY) {
                                …
                        } else if (keyCode == Resources.UP_KEY) {
                                …
                        } else if (keyCode == Resources.DOWN_KEY) {
                                …
                        }
                } else if (state == Resources.PLAYING_STATE) {
                        if ((keyCode == Resources.RIGHT_SOFT_KEY) || (keyCode ==
Resources.END_KEY)) {
```

```
                    …
                } else if ((keyCode == Resources.LEFT_SOFT_KEY)) {
                 …
                }else{
                …
                }
                …
            } …
        }
```

Um ponto de destaque é que quando o usuário opta por iniciar um dos vídeos disponíveis, é iniciada uma nova *thread*. Isto é importante, pois a conexão e início do vídeo tomam muito tempo e, portanto, não podemos travar a interação do usuário com o aplicativo. Esta é a razão para esta classe implementar a interface *Runnable.* Desta forma, sempre que a tecla "Play" é acionada é criada e iniciada uma nova *thread*. Para criar um novo objeto do tipo *Thread* é preciso passar para o construtor um objeto que implemente a interface *Runnable*. Sendo assim, a classe VideoCanvas que é passada para o construtor da *Thread* implementa a interface *Runnable* através da implementação do método *run()*. Dessa forma, quando a tecla "Play" é acionada método *run()* é chamado em uma nova *thread*. A implementação do método *run()* apenas chama o método *play()*. No método *play()* é feito todo o tratamento da parte multimídia do aplicativo.

Primeiramente é criado um objeto *Player* através do vídeo escolhido, que na verdade é representado por uma URL. Depois disso, o objeto *player* navega em seus estados, conforme já mostrado na seção 3.2.2, que explica a JSR 135. Em seguida, os controles de vídeo e volume são requeridos. Por fim, o vídeo é iniciado e colocado na tela. O trecho a seguir, retirado do código do aplicativo destaca o que foi citado.

```
Public void play() {
            if (Resources.VIDEO_URL1 == null || Resources.VIDEO_URL2 == null) {
                …
```

```
            } else {
                try {
                    player = Manager.createPlayer(Resources.VIDEO_URL1);
                    …
                    try {
                        player.realize();
                    …
                        player.prefetch();

                    } catch (InterruptedException e) {
                    …
                    }
            // get the volume control
            volumeControl = (VolumeControl) player.getControl("VolumeControl");
            …
            // get the video control
            videoControl = (VideoControl) (player.getControl("VideoControl"));
            …
            videoControl.setVisible(true);
            player.start();
                        …
        }
```

Outro ponto importante para a aplicação é a implementação da interface *PlayerListener*. Implementando esta interface através do método *playerUpdate*(), todos os eventos ocorridos com o objeto *Player* podem ser tratados. Desta forma, no caso do aplicativo, eventos como o fim do vídeo ou o fechamento do *Player* são tratados da maneira apropriada.

A classe *Resources*, ainda não descrita, é apenas um repositório de informações. Estas informações são referentes a tamanhos em *pixels*, a códigos de teclas, e a URLs a serem acessadas. Com esta classe, a intenção é concentrar a definição das constantes em um único lugar. Todas as variáveis desta classe são estáticas e podem ser acessadas diretamente pela classe VideoCanvas. O trecho a seguir mostra algumas das variáveis

usadas pela classe VideoCanvas, como o tamanho da tela, os valores das teclas, e a URL de *streaming* do primeiro vídeo.

```
Public class Resources {
        …
        /**
         * Screen Constants
         */
        public static int SCREEN_WIDTH = 176;
        public static int SCREEN_HEIGHT = 220;
        …
        /**
         * Key constants
         */
        public static int LEFT_SOFT_KEY = -6;
        public static int RIGHT_SOFT_KEY = -7;
        public static int END_KEY = -11;
        public static int UP_KEY = -1;
        public static int DOWN_KEY = -2;
        …
        public static String VIDEO_URL1 = "rtsp://joa.gta.ufrj.br/MOV00001.3gp";
        ..
}
```

4.2 Servidor

Existe uma documentação completa sobre a estrutura do Darwin Streaming Server. O documento [18] é na verdade um guia para programação de novos módulos para o servidor. Nele são cobertos desde os conceitos sobre o servidor até um passo-a-passo para compilação e inserção de um novo módulo desenvolvido.

Destaca-se neste servidor sua arquitetura modular que facilita a criação de novas funcionalidades, através de módulos independentes. Neste sentido, o servidor possui um

processo principal, chamado de *Core Server*, que é na verdade uma interface entre as requisições dos clientes e os módulos.

Os módulos devem possuir uma estrutura específica, obrigatoriamente possuindo dois métodos, um chamado pelo servidor para iniciá-lo e outro chamado pelo servidor para executar uma determinada tarefa. Para saber que módulos devem ser chamados para um determinado evento cada módulo deve explicitar seu papel. Sendo assim, cada módulo tem que possuir uma lista de papéis, denominados “Roles” em inglês. Os papéis disponíveis e as tarefas a serem desempenhadas por cada um estão detalhados no documento citado.

O escopo deste trabalho limitou-se no estudo da arquitetura do servidor, não englobando o desenvolvimento de novos módulos ou melhoria dos existentes. Contudo, no processo de aprendizado foi produzido, compilado e colocado em execução, um módulo sem funcionalidades. Este módulo possui o papel “RTCP Process role”, ou seja, seria responsável pelo processamento dos relatórios de qualidade dos clientes. Por uma restrição de tempo, o escopo deste trabalho, foi limitado ao estudo do servidor e por isso a implementação deste módulo fica como uma sugestão de trabalho futuro.

5 Conclusão

Analisando criticamente o trabalho realizado, conclui-se que o mesmo atingiu os objetivos determinados inicialmente. Isto porque propiciou ao aluno um conhecimento geral sobre os diversos componentes de uma solução de *streaming* de vídeo para um dispositivo móvel.

Além do aprendizado, o produto final, considerando toda a solução, que envolve toda configuração na parte do servidor e o desenvolvimento no cliente, também foi satisfatório. Primeiramente, porque a solução funcionou e em segundo lugar, pois é uma solução totalmente gratuita que poderia ser implantada sem qualquer custo de software.

É importante ressaltar que o uso da linguagem Java possibilita que com apenas poucas alterações o código do cliente possa ser aproveitado para outros dispositivos. Esta independência do *hardware* é certamente uma grande vantagem da linguagem Java e foi decisiva na sua escolha para a implementação do cliente. Além disso, se pode concluir que o uso da MMAPI facilita o desenvolvimento de aplicativos de *streaming* de vídeo, pois já possui métodos de alto nível que encapsulam toda a complexidade dos protocolos RTSP e RTP. Contudo esta facilidade também é uma desvantagem, pois limita a ação do programador. Neste projeto, por exemplo, seria interessante tratar os relatórios RTCP enviados para o cliente, mas isto via MMAPI não é possível.

Diversas foram as dificuldades encontradas no projeto. Uma delas foi encontrar um servidor livre capaz de fazer *streaming* do formato 3GP, específico para celulares. Também foi difícil transpor os *firewalls* no caminho entre o cliente e servidor, configurando-os onde era possível ou escolhendo APNs de operadoras que permitam o tráfego RTSP e RTP. Outra dificuldade foi achar um aparelho celular que tivesse em sua implementação da MMAPI o suporte para *streaming* de vídeo, assim como o aprendizado da MMAPI em si e da linguagem Java.

Apesar das dificuldades, o projeto pôde ser concluido com êxito. Sendo assim, a produção dos vídeos no formato padrão, a configuração do servidor *Darwin Streaming Server* e o desenvolvimento de uma aplicação cliente, possibilitaram a criação de uma solução completa de *streaming* de vídeo.

Como trabalhos futuros, diversas possibilidades podem ser enxergadas. Na esfera da implementação de novos módulos no servidor, uma sugestão seria a implementação de um módulo que fosse capaz de armazenar informações de perdas de pacotes dos clientes celulares para uma futura análise sobre a qualidade do serviço de

streaming de vídeo em uma rede celular brasileira. Na esfera do cliente, uma sugestão seria o aperfeiçoamento do cliente e sua implementação em outros dispositivos móveis como PDAs ou outros modelos de celulares.

6 Referência Bibliográfica

[1] ITELOGY PARTNERS. Mercado Brasileiro de Dados para Telefonia Celular: 2004. Disponível em: < http://www.tele-semana.com/archivo/Download.php?c=0649910006021-283> Acesso em: 03 de jul. 2006.

[2] TELECO. Evolução Tecnológica de Sistemas Celulares. Disponível em: <http://www.teleco.com.br/ tecnocel.asp > Acesso em: 04 de jul. 2006.

[3] 3GPP. Transparent end-to-end Packet-switched Streaming Service. V 6.8.0. [S.L.] 3GPP, 2006. Especificação Técnica.

[4] FFMPEG. Disponível em: <http://ffmpeg.mplayerhq.hu> Acesso em: 04 de jul. 2006.

[5] X264. Disponível em: <http://developers.videolan.org/x264.html> Acesso em: 04 jul. 2006.

[6] ERIGHTSOFT. Super. Disponível em: <http://www.erightsoft.com/SUPER.html> Acesso em: 03 de jul. 2006.

[7] APPLE COMPUTER. Darwin Streaming Server. Disponível em: <http://developer.apple.com/opensource/server/streaming/index.html> Acesso em: 03 de jul. 2006.

[8] FEUVRE, J. GPAC. Disponível em: <http://gpac.sourceforge.net/index.php> Acesso em: 03 de jul. 2006.

[9] SUN MICROSYSTEMS. Sun Java Wireless Toolkit for CLDC. Disponível em: <http://java.sun.com/products/sjwtoolkit/> Acesso em: 05 jul. 2006.

[10] IETF. RFC 2326: Real Time Streaming Protocol (RTSP). [S.L.], 1998.

[11] TOPIC, M.Streaming Media Demystified. New York: McGraw Hill, 2002

[12] IETF. RFC 3550: RTP: A Transport Protocol for Real-Time Applications. [S.L.], 2003.

[13] TOPLEY, K. J2ME in a nutshell. [S.L.]: O'Reilly Media, 2002.

[14] GOYAL, V. Pro Java ME MMAPI: Mobile Media API for Java Micro Edition. [S.L.]: Apress, 2006.

[15] SUN MICROSYSTEMS; NOKIA CORPORATION. Mobile Media API (JSR-135) Specification. Disponível em: < http://jcp.org/aboutJava/communityprocess/mrel/jsr135/index.html> Acesso em: 03 de jul. 2006.

[16] NOKIA CORPORATION. Series 60 Developer Platform 2.0: Usability Guidelines For Enterprise Applications. Nokia Corporation. Disponível em: < http://sw.nokia.com/id/4ac4491e-d232-459a-81a5-888f770cd719/Series_60_DP_2_0_Usability_Guidelines_For_Enterprise_Applications_v1_0_en.pdf> Acesso em: 03 de jul. 2006.

[17] WIKIPEDIA. Singleton Pattern. Disponível em: <http://en.wikipedia.org/wiki/Singleton_pattern> Acesso em 10 de jul. 2006.

[18] APPLE COMPUTERS. QuickTime Streaming Server Modules - Programming Guide. Disponível em: < http://developer.apple.com/documentation/QuickTime/QTSS/QTSS.pdf> Acesso em: 03 jul. 2006.

[19] WIKIPEDIA. RDT stream transport. Disponível em: <http://en.wikipedia.org/wiki/RDT_stream_transport> Acesso em: 12 jul. 2006.

[20] CRANLEY, N; DAVIS, M. Performance analysis of network-level QoS with encoding configurations for unicast video streaming over IEEE 802.11 WLAN networks. Wireless Networks, Communications and Mobile Computing, 2005 International Conference on, v.1, p.510-515, jun. 2005.

7 Apêndice

7.1 Análise de protocolos

Uma parte interessante do projeto foi poder analisar o funcionamento dos protocolos através da ferramenta Etherreal. Sendo assim, para ilustrar o funcionamento do protocolo RTSP, foi destacado o trecho em que o celular através do aplicativo, interage com o servidor usando os diferentes comandos RTSP.

Origem	Mensagem	Observação
cliente	DESCRIBE rtsp://146.164.69.35/MOV00001.3gp RTSP/1.0 CSeq: 1 Accept: application/sdp User-Agent: Sony Ericsson/K500	Cliente pede servidor descrever a URL forncecida.
servidor	RTSP/1.0 200 OK Server: DSS/5.5.1 (Build/489.8; Platform/Linux; Release/Darwin;) Cseq: 1 Last-Modified: Tue, 25 Apr 2006 21:26:46 GMT Cache-Control: must-revalidate Content-length: 488 Date: Wed, 21 Jun 2006 14:06:01 GMT Expires: Wed, 21 Jun 2006 14:06:01 GMT Content-Type: application/sdp x-Accept-Retransmit: our-retransmit x-Accept-Dynamic-Rate: 1 Content-Base: rtsp://146.164.69.35/MOV00001.3gp/ v=0 o=StreamingServer 3359887560 1146000406000 IN IP4 146.164.69.35 s=/MOV00001.3gp u=http:/// e=admin@ c=IN IP4 0.0.0.0 b=AS:72	Servidor retorna com uma descrição completa que envolve o nome do servidor, a plataforma, e também o conteúdo. Sobre o conteúdo são informados os codecs de vídeo e áudio , ale´m de outras informações.

	t=0 0 a=control:* a=x-copyright: MP4/3GP File hinted with GPAC 0.4.0 (C)2000-2005 - http://gpac.sourceforge.net a=range:npt=0- 9.40000 m=video 0 RTP/AVP 96 b=AS:59 a=rtpmap:96 H263-1998/90000 a=control:trackID=65536 a=cliprect:0,0,96,128 m=audio 0 RTP/AVP 97 b=AS:13 a=rtpmap:97 AMR/8000/1 a=control:trackID=65537 a=fmtp:97 octet-align	
cliente	SETUP rtsp://146.164.69.35/MOV00001.3gp/trackID=65537 RTSP/1.0 CSeq: 2 User-Agent: Sony Ericsson/K500 Transport: RTP/AVP/UDP;unicast;client_port=24912-24913	Especifica o transporte como RTP, alocando as portas 24912 e 24913 para serem usadas no cliente.
servidor	RTSP/1.0 200 OK Server: DSS/5.5.1 (Build/489.8; Platform/Linux; Release/Darwin;) Cseq: 2 Last-Modified: Tue, 25 Apr 2006 21:26:46 GMT Cache-Control: must-revalidate Session: 8598385139658089008 Date: Wed, 21 Jun 2006 14:06:03 GMT Expires: Wed, 21 Jun 2006 14:06:03 GMT Transport: RTP/AVP/UDP;unicast;source=146.164.69.35;client_	Servidor responde o cliente com confirmação do protocolo de transporte RTP e especifica as portas para acesso (6970-6971).

	port=24912-24913;server_port=6970-6971;ssrc=2F49A9CB	
cliente	PLAY rtsp://146.164.69.35/MOV00001.3gp RTSP/1.0 CSeq: 4 User-Agent: Sony Ericsson/K500 Range: npt=0- Session: 8598385139658089008	Cliente ordena que a transmissão do vídeo comece.
servidor	RTSP/1.0 200 OK Server: DSS/5.5.1 (Build/489.8; Platform/Linux; Release/Darwin;) Cseq: 4 Session: 8598385139658089008 Range: npt=0.00000-9.40000 RTP-Info: url=rtsp://146.164.69.35/MOV00001.3gp/trackID=65537;seq=50720;rtptime=1856799883,url=rtsp://146.164.69.35/MOV00001.3gp/trackID=65536;seq=25545;rtptime=120437254	Servidor confirma o início da transmissão.
cliente	TEARDOWN rtsp://146.164.69.35/MOV00001.3gp RTSP/1.0 User-Agent: Sony Ericsson/K500 CSeq: 5 Session: 8598385139658089008	Cliente solicita a parada da transmissão e fim da sessão.
servidor	RTSP/1.0 200 OK Server: DSS/5.5.1 (Build/489.8; Platform/Linux; Release/Darwin;) Cseq: 5 Session: 8598385139658089008 Connection: Close	Servidor responde com confirmação do fechamento da conexão.

Outro conjunto de pacotes interessantes a serem analisados são os pacotes RTCP. Primeiramente é apresentado um *Receiver Report* gerado pelo aparelho celular durante o *streaming*. É possível destacar algumas informações mais interessantes como o tipo do pacote (*Packet type: Receiver Report (201)*), a fração de pacotes perdidos naquele intervalo (*Fraction lost: 160 / 256*) e o número cumulativo de pacotes perdidos até o momento (*Cumulative number of packets lost: 295*).

```
Real-time Transport Control Protocol (Receiver Report)
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 0001 = Reception report count: 1
    Packet type: Receiver Report (201)
    Length: 7
    Sender SSRC: 113989717
    Source 1
        Identifier: 456700746
        SSRC contents
            Fraction lost: 160 / 256
            Cumulative number of packets lost: 295
        Extended highest sequence number received: 25599
            Sequence number cycles count: 0
            Highest sequence number received: 25599
        Interarrival jitter: 16829
        Last SR timestamp: 1622786244
        Delay since last SR timestamp: 2083521
```

O servidor de *streaming* de vídeo também gera relatórios para o cliente. O pacote mostrado a seguir é um *Sender Report* capturado durante o *streaming* de vídeo para o celular. Neste pacote, se pode destacar as seguintes informações, como a identificação da fonte através do SSCR (*Sender SSRC: 976329591*) o *timestamp* (*RTP timestamp: 250668297*) e o número de pacotes enviados (*Sender's packet count: 2108*),

```
Real-time Transport Control Protocol (Sender Report)
```

```
    Stream setup by RTSP (frame 1955)
        Setup frame: 1955
        Setup Method: RTSP
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 0000 = Reception report count: 0
    Packet type: Sender Report (200)
    Length: 6
    Sender SSRC: 976329591
    Timestamp, MSW: 3360579811
    Timestamp, LSW: 3835405794
    MSW and LSW as NTP timestamp: Not representable
    RTP timestamp: 250668297
    Sender's packet count: 2108
    Sender's octet count: 69564
```

7.2 Código fonte do cliente

7.2.1 VideoMIDlet

```
import javax.microedition.lcdui.Display;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

import javax.microedition.lcdui.*;

/**
 * This is the main class of VideoStreaming application
 *
 * @author Ricardo Clemente
 */

public class VideoMidlet extends MIDlet {
```

```
private final VideoCanvas videoCanvas;

/** Singleton do MIDletController */
private static VideoMidlet instance;

public static VideoMidlet getInstance() {
	return instance;
}

/**
 * Initialize the canvas
 */
public VideoMidlet() {
	videoCanvas = new VideoCanvas(Display.getDisplay(this));
	videoCanvas.setFullScreenMode(true);
}

public void startApp() throws MIDletStateChangeException {
	instance = this;
	videoCanvas.start();

}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}

public void exitRequested() {
	destroyApp(false);
	notifyDestroyed();
}
```

```
	Display getDisplay() {
		return Display.getDisplay(this);
	}

	void videoExit() {
		exitRequested();
	}

	void alertError(String message) {
		Alert alert = new Alert("Error", message, null, AlertType.ERROR);
		Display display = Display.getDisplay(this);
		Displayable current = display.getCurrent();
		if (!(current instanceof Alert)) {
			display.setCurrent(alert, current);
		}
	}
}
```

7.2.2 VideoCanvas

```
import java.io.IOException;

import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.game.GameCanvas;
import javax.microedition.media.Manager;
import javax.microedition.media.MediaException;
import javax.microedition.media.Player;
import javax.microedition.media.PlayerListener;
import javax.microedition.media.control.VideoControl;
import javax.microedition.media.control.VolumeControl;
```

```
class VideoCanvas extends GameCanvas implements PlayerListener, Runnable {

    /**
     * A handle to the screen of the device.
     */
    Display myDisplay;

    Thread videoThread;

    int step = 0;

    int pos = 0;

    /**
     * A handle for video and volume control. These are mmapi controls
     */
    VideoControl videoControl;

    VolumeControl volumeControl;

    boolean isMute = false;
    boolean exitRequested = false;

    /**
     * Keeps the actual canvas state. This variable will be used by the paint()
     * method
     */

    private int state = Resources.MENU_STATE;

    /*
     * The main class in MMAPI is the player. This is responsable for playing
     * the video.
```

```
 */
private Player player;




private Image header;

/*
 * new constructor
 */

public VideoCanvas(Display d) {
	super(false);
	myDisplay = d;
}

public void start() {
	try {
		header = Image.createImage(Resources.HEADER_FILE);
	} catch (IOException e) {
		e.printStackTrace();
	}
	exitRequested = false;
	myDisplay.setCurrent(this);

	repaint();
}

public void run() {
	try {
		play();
	} catch (Throwable t) {
		this.stop();
	}
```

```
	}

	public void paint(Graphics g) {
		// x and y are the coordinates of the top corner
		int x = g.getClipX();
		int y = g.getClipY();

		// w and h are the width and height of the display area:
		int w = Resources.SCREEN_WIDTH;
		int h = Resources.SCREEN_HEIGHT;

		// clear the screen (paint it white):
		g.setColor(0xffffff);
		g.fillRect(x, y, w, h);

		// Handles writing on Screen
		Font font = g.getFont();
		int fontWidth;
		int fontHeight = font.getHeight();

		g.setColor(0x00ff0000);
		g.setFont(font);

		if (this.state == Resources.PLAYING_STATE) {

			fontWidth = font.stringWidth("Carregando...");
			// write the string in the center of the screen
			g.drawString("Carregando...", (w - fontWidth) / 2,(h - fontHeight) / 2,
Graphics.TOP | Graphics.LEFT);

			// paitns a rectangle on borders
			g.drawRect(x + 1, y + 1, w - 3, h - 3);

			fontWidth = font.stringWidth("on/off");
```

```
                    g.drawString("on/off", (x + 5), (h - fontHeight - 3), Graphics.TOP|
Graphics.LEFT);

                    fontWidth = font.stringWidth("Stop");
                    g.drawString("Stop", (w - fontWidth - 5), (h - fontHeight - 3),
Graphics.TOP | Graphics.LEFT);
            }

            else if (this.state == Resources.MENU_STATE) {

                    // paitns a rectangle on borders
                    g.drawRect(x + 1, y + 1, w - 3, h - 3);
                    // paints "Play" command
                    fontWidth = font.stringWidth("Play");
                    g.drawString("Play", (x + 5), (h - fontHeight - 3), Graphics.TOP |
Graphics.LEFT);
                    // paints "Exit" command
                    fontWidth = font.stringWidth("Exit");
                    g.drawString("Exit", (w - fontWidth - 5), (h - fontHeight - 3),
Graphics.TOP | Graphics.LEFT);

                    //paints Header

        g.drawImage(header,2*Resources.ATOM_STEP,2*Resources.ATOM_STEP,
Graphics.TOP | Graphics.LEFT);

                    // paints videos Options
                    fontWidth = font.stringWidth("Streaming 1");
                    g.drawString("Streaming 1", (x + Resources.COLUMN_STEP),
4*Resources.LINE_STEP, Graphics.TOP | Graphics.LEFT);

                    fontWidth = font.stringWidth("Streaming 2");
                    g.drawString("Streaming 2", (x + Resources.COLUMN_STEP),
5*Resources.LINE_STEP, Graphics.TOP | Graphics.LEFT);
```

```
                // draws cursor
                g.drawRect((x + 2*Resources.ATOM_STEP),
4*(Resources.LINE_STEP)+ 2*Resources.ATOM_STEP + pos, Resources.ATOM_STEP,
Resources.ATOM_STEP);

        } else if (this.state == Resources.ERROR_STATE) {
                state = Resources.MENU_STATE;
                repaint();
        }
    }

    public void keyPressed(int keyCode) {
        if (state == Resources.MENU_STATE) {
                if ((keyCode == Resources.RIGHT_SOFT_KEY) || (keyCode ==
Resources.END_KEY)) {
                        VideoMidlet.getInstance().exitRequested();
                } else if (keyCode == Resources.LEFT_SOFT_KEY) {
                        state = Resources.PLAYING_STATE;
                        repaint();
                        this.videoThread = new Thread(this);
                        this.videoThread.start();
                } else if (keyCode == Resources.UP_KEY) {
                        if(pos == Resources.LINE_STEP){
                                pos = 0;
                        }
                        repaint();
                } else if (keyCode == Resources.DOWN_KEY) {
                        if(pos == 0){
                                pos = Resources.LINE_STEP;
                        }
                        repaint();
                }
        } else if (state == Resources.PLAYING_STATE) {
```

```
                if ((keyCode == Resources.RIGHT_SOFT_KEY) || (keyCode == Resources.END_KEY)) {
                        // stop
                        stop();
                        this.videoThread = null;
                        state = Resources.MENU_STATE;
                        repaint();
                } else if ((keyCode == Resources.LEFT_SOFT_KEY)) {
                        mute();
                }else{
                        repaint();
                }

        } else if (state == Resources.ERROR_STATE) {
                if ((keyCode == Resources.RIGHT_SOFT_KEY)
                                || (keyCode == Resources.END_KEY)) {
                        VideoMidlet.getInstance().exitRequested();
                } else if (keyCode == Resources.LEFT_SOFT_KEY) {
                        repaint();
                }
        }
    }

    void stop() {
        if (player != null) {
                player.close();
        }
    }

    void mute() {
        if (player != null) {
                isMute = !isMute;
                volumeControl.setMute(isMute);
        }
```

```
        }

        public void play() {
                if (Resources.VIDEO_URL1 == null || Resources.VIDEO_URL2 == null) {
                        VideoMidlet.getInstance().alertError("No video url specified");
                } else {
                        try {
                                if(pos == 0){
                                        player                                          =
Manager.createPlayer(Resources.VIDEO_URL1);
                                }else{
                                        player                                          =
Manager.createPlayer(Resources.VIDEO_URL2);
                                }

                                player.addPlayerListener(this);
                                try {
                                        player.realize();
                                        while (player.getState() != Player.REALIZED) {
                                                Thread.sleep(100);
                                        }

                                        player.prefetch();
                                        while (player.getState() != Player.PREFETCHED) {
                                                Thread.sleep(100);
                                        }

                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                                // get the volume control
                                volumeControl                   =                   (VolumeControl)
player.getControl("VolumeControl");
                                if (volumeControl == null) {
```

```
                                        state = Resources.ERROR_STATE;
                                        VideoMidlet.getInstance().alertError("VolumeControl
not supported");
                                } else {
                                        volumeControl.setMute(isMute);
                                }

                                // get the video control
                                videoControl                    =                    (VideoControl)
(player.getControl("VideoControl"));

                                if (videoControl == null) {
                                        state = Resources.ERROR_STATE;
                                        VideoMidlet.getInstance().alertError("VideoControl
not supported");
                                } else {

        videoControl.initDisplayMode(VideoControl.USE_DIRECT_VIDEO,this);



        videoControl.setDisplaySize(Resources.QCIF_WIDHT                                -
3*Resources.ATOM_STEP,Resources.QCIF_HEIGHT);

        videoControl.setDisplayLocation(Resources.ATOM_STEP, 50);
                                        //videoControl.setDisplayFullScreen(true);
                                        videoControl.setVisible(true);
                                        player.start();
                                        while (player.getState() != Player.STARTED) {
                                                try {
                                                        Thread.sleep(100);
                                                } catch (InterruptedException e) {
                                                        e.printStackTrace();
                                                }
                                        }
```

```
                        }

                } catch (IOException ioe) {
                        discardPlayer();
                        VideoMidlet.getInstance().alertError("IOException:"          +
ioe.getMessage());
                        state = Resources.ERROR_STATE;
                } catch (MediaException me) {
                        discardPlayer();
                        VideoMidlet.getInstance().alertError("MediaException:"       +
me.getMessage());
                        state = Resources.ERROR_STATE;
                } catch (SecurityException se) {
                        discardPlayer();
                        VideoMidlet.getInstance().alertError("SecurityException:"    +
se.getMessage());
                        state = Resources.ERROR_STATE;
                } catch (OutOfMemoryError om) {
                        discardPlayer();
                        VideoMidlet.getInstance().alertError("MemoryException:"      +
om.getMessage());

                        state = Resources.ERROR_STATE;
                }
        }
    }

    // Called in case of exception to make sure invalid players are closed
    private void discardPlayer() {
        if (player != null) {
            player.close();
            player = null;
        }
    }
```

```
        public void playerUpdate(final Player p, final String event, final Object eventData) {
                // queue all to updateEvent
                Display display = VideoMidlet.getInstance().getDisplay();
                display.callSerially(new Runnable() {
                        public void run() {
                                VideoCanvas.this.updateEvent(p, event, eventData);
                        }
                });
        }

        private void updateEvent(Player p, String event, Object eventData) {
                if (event == END_OF_MEDIA) {
                        state = Resources.MENU_STATE;
                        p.close();
                        repaint();
                } else if (event == CLOSED) {
                        player = null;
                        state = Resources.MENU_STATE;
                }
        }
}
```

7.2.3 Resources

```
public class Resources {
 /**
  * This class is the repository for all constants configurations
  */

        /**
         *  Painting constants
         */
```

```
public static int LINE_STEP = 15;
public static int COLUMN_STEP = 15;
public static int ATOM_STEP = 3;

/**
 * Screen Constants
 */
public static int SCREEN_WIDTH = 176;
public static int SCREEN_HEIGHT = 220;


/**
 * States constants
 */
public static int MENU_STATE = 1;
public static int PLAYING_STATE = 2;
public static int ERROR_STATE = 3;


/**
 * Key constants
 */
public static int LEFT_SOFT_KEY = -6;
public static int RIGHT_SOFT_KEY = -7;
public static int END_KEY = -11;
public static int UP_KEY = -1;
public static int DOWN_KEY = -2;

/**
 * Video constants
 */

public static String SUPPORTED_FORMAT = "video/3gp";
```

```
        public static String VIDEO_URL1 = "rtsp://joa.gta.ufrj.br/MOV00001.3gp";
        public static String VIDEO_URL2 = "rtsp://joa.gta.ufrj.br/pepsi.3GP";

        public static int QCIF_HEIGHT = 144;
        public static int QCIF_WIDHT = 176;

        /**
         * Image constants
         */
        public static String HEADER_FILE = "/logo-GTA_header.png";

    }
```