UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Alexandre Carapiá Ferraz
Arthur Cunha Granado

Análise e Implementação de Arquiteturas para Conferências Multimídia

Niterói
2006

ALEXANDRE CARAPIÁ FERRAZ
ARTHUR CUNHA GRANADO

Análise e Implementação de Arquiteturas para Conferências Multimídia

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação

Orientador: Célio Vinicius Neves de Albuquerque

Niterói
2006

ALEXANDRE CARAPIÁ FERRAZ

ARTHUR CUNHA GRANADO

Análise e Implementação de Arquiteturas para Conferências Multimídia

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação

Aprovados em agosto de 2006.

BANCA EXAMINADORA

_____

Prof. CÉLIO VINICIUS NEVES DE ALBUQUERQUE
Orientador
UFF

_____

Profa. SIMONE DE LIMA MARTINS
UFF

_____

Prof. EUGENE FRANCIS VINOD REBELLO
UFF

_____

TIAGO SILVA PROENÇA
UFF

Niterói
2006

RESUMO

Com o aumento das aplicações multimídia na Internet surge o desafio de como prover suporte às aplicações de conferência multimídia multi-destinatárias, tanto no nível de rede como no nível de aplicação, para o envio de pacotes de mídia em ambientes multiponto-multiponto (MPMP).

Este trabalho descreve e analisa três diferentes arquiteturas, no nível de aplicação, para o envio de dados multimídia através da Internet, com o objetivo de comparar e contrastar suas características, demonstrando como se podem minimizar os problemas enfrentados utilizando-se protocolos e arquiteturas de comunicação existentes.

Para efeito de demonstração e de testes realísticos, duas arquiteturas MPMP foram implementadas em Java utilizando-se o *framework* JMF (*Java Media Framework*) para a captura e transmissão multi-destinatária de vídeo em tempo real, verificação do comportamento dos pacotes de mídia em uma aplicação de videoconferência.

Palavras Chave:

Internet, Multiponto, Redes de Computadores, Videoconferência.

ABSTRACT

With the ascension of multimedia applications on the Internet comes the challenge to provide support to multicast multimedia conference applications, both in the network and in the application level, in order to send media packets over multipoint-to-multipoint (MPMP) environments.

This work describes and analyses three different architectures in the application level, for sending multimedia data over the Internet, with the objective of comparing and highlighting their features, showing how occurring problems can be minimized by using already existing communication protocols and architectures.

For demonstration and real test purposes, two MPMP architectures were developed in Java using the Java Media Framework (JMF) to capture and transmit multicast video in real-time and to verify the behavior of media packets in a videoconference application.

Keywords:

Internet, Multicast, Computer Networks, Videoconference.

LISTA DE ACRÔNIMOS

AIFF:	Audio Interchange File Format
AU:	Sun Audio
AVI:	Audio/Video Interleave
DVMRP:	Distance Vector Multicast Routing Protocol
JMF:	Java Media Framework
IGMP:	Internet Group Management Protocol
IP:	Internet Protocol
MOV:	QuickTime Movie
MP3:	MPEG Layer 3 Audio
MPEG:	Motion Picture Experts Group
SPL:	Future Splash
SWF:	Macromedia Flash 2 Movies
RFC:	Request For Comments
RTCP:	Real Time Control Protocol
RTP:	Real Time Protocol
TCP:	Transmission Control Protocol
UDP:	User Datagrama Protocol
XML:	Extensible Markup Language
WAV:	Audio Wave

SUMÁRIO

LISTA DE FIGURAS

TABELAS

CAPÍTULO 1 - INTRODUÇÃO

1.1 Motivação

O surgimento de várias aplicações multimídia na Internet ocasionou uma grande necessidade de se aperfeiçoar o envio dos pacotes de mídia através da rede, para que se obtenha otimizado consumo de banda passante e uma boa qualidade em sua reprodução.

A transmissão *multicast* [4] na Internet é foco de vários debates entre especialistas da área, principalmente tratando-se de *multicast* de pacotes de mídia. Isso porque existem problemas para se realizar a transmissão *multicast*. Pode ocorrer grande replicação dos pacotes que serão transmitidos, porque o transmissor deveria enviar um pacote para cada receptor, aumentando o fluxo na rede a níveis que podem levar o acarretamento de congestionamentos nos enlaces e nos roteadores. O que se refletiria em aumento dos atrasos fim-a-fim para todos os pacotes de aplicações que estejam utilizando aqueles enlaces e roteadores. Tal conseqüência é muito ruim para aplicações multimídia.

Alguns destes especialistas [6] acreditam que toda a estrutura, no nível de rede, deveria ser mudada para que houvesse suporte para o *multicast*. Essas mudanças envolveriam a substituição de roteadores e a inclusão de novos protocolos de gerenciamento *multicast* nos mesmos, para que fossem capazes de realizar a transmissão para grupos *multicast.*

Já outros acreditam [6] que não é necessário mudar nada, apenas adaptar-se a essa nova realidade, já que continuamente há aumento da largura de banda entre os enlaces de comunicação dos roteadores. Esses problemas de transmissão *multicast* seriam resolvidos com adoção de novas estratégias de *multcast* no nível da aplicação.

Diante desses desafios, foram analisados alguns modelos de arquiteturas para comunicação multimídia multiponto-multiponto (MPMP) [17], no nível de aplicação. Algumas destas arquiteturas, apresentadas no Capítulo 3, são usadas em diversas outras aplicações. Este projeto implementou duas arquiteturas multiponto-multiponto: a arquitetura com um servidor refletor, apresentada na Seção 3.2, e a arquitetura sem o servidor refletor, apresentada na Seção 3.3.

Na implementação das arquiteturas, decidiu-se construir uma aplicação de videoconferência para a transmissão dos pacotes de mídia. Desenvolvida em cima do *Java Media Framework* (JMF) [5], um *framework* desenvolvido pela *Sun Microsystem* [15], o qual é apresentado com mais detalhe na Seção 4.2. Através deste poderoso framework com vários recursos úteis paras aplicações multimídia, foi possível construir a aplicação de videoconferência que utilizou as arquiteturas apresentadas neste trabalho.

A construção desta aplicação permitiria o aprofundamento dos conceitos envolvidos na programação em rede, e no estudo dos protocolos utilizados por esse tipo de aplicação. O conhecimento do JMF é de grande valia para que se possam desenvolver outras aplicações multimídia. Tal conhecimento proporcionou nova visão sobre o assunto o que gerou maior motivação e empenho na obtenção dos resultados.

1.2 Objetivos

O objetivo deste projeto é analisar as arquiteturas propostas no Capítulo 3 para aplicações multimídia. Este tipo de aplicação tem características interessantes, como um alto nível de tolerância para perdas de pacotes de mídia, mas possuem um baixo nível de tolerância para o atraso na entrega destes mesmos pacotes. Por isso, em sua grande maioria, esse tipo de aplicação utiliza-se do *User Datagrama Protocol* (UDP) [13] para o envio da mídia ao invés do *Transmission Control Protocol* (TCP) [10] pelo fato do último ter retransmissão de pacote, que aumenta o tráfico de pacotes e o atraso fim-a-fim, e controle de congestionamento, que reduz a taxa de transmissão dos pacotes.

Como o UDP não possui entrega confiável de pacotes, oferecida pelo TCP, haverá perdas de pacotes de mídia, mas que é bem suportada pelas aplicações de multimídia na Internet.

Espera-se que com este material consiga-se demonstrada a importância que as aplicações multimídia têm, cada vez mais, desempenhado na Internet, e como é importante achar soluções para o envio de pacotes que onerem o mínimo possível a rede e que possibilitem uma grande qualidade na reprodução. Caso isso não ocorra, é provável que em um futuro próximo, o tráfico dos pacotes de mídia, que estas aplicações irão realizar, cause um crescimento excessivo de pacotes na rede levando a grande “engarrafamentos” e atrasos fim-a-fim impraticáveis.

1.3 Organização

Este trabalho foi organizado da seguinte forma. No Capítulo 2 são explicados os protocolos no nível de transporte mais importantes para aplicações de transmissão de mídia. O Capítulo 3 descreve quatro arquiteturas para transporte de mídia na Internet, sendo as últimas três arquiteturas multiponto-multiponto. No Capítulo 4 são apresentados dois *frameworks* em Java que auxiliaram o desenvolvimento das implementações, o JMF e o Log4J[8] que é um *framework* para criação de *logs*. O Capítulo 5 contem toda a explicação da implementação da arquitetura com servidor refletor e da arquitetura sem servidor. Finalizamos com o Capítulo 6, onde se tiram conclusões após todo a análise do material aqui contido.

CAPÍTULO 2 – PROTOCOLOS

2.1 Introdução

Os protocolos constituem parte muito importante das aplicações de rede, pois são eles que sincronizam as aplicações indicando como elas devem se comportar no momento que recebem uma mensagem de outra aplicação, ou seja, quais devem ser os procedimentos tomados para que o fluxo de processamento continue como esperado.

O TCP [10] é o protocolo da camada de transporte da Internet mais usado, por garantir a entrega dos pacotes enviados por ele. O TCP possui um protocolo de apresentação (3-way handshake), no qual há alocação de recursos, como criação de *buffers*, que permitirá ao TCP oferecer os serviços de entrega confiável de pacotes e o controle de congestionamento, o que o torna muito eficiente para o envio de pacotes na Internet. Contudo o TCP possui um desempenho ruim, para aplicações multimídia na Internet, porque elas têm baixa tolerância a atrasos de pacotes e razoável tolerância a pequenas perdas dos mesmos.

O serviço de entrega confiável de pacotes, oferecido pelo TCP, provoca retransmissão dos pacotes perdidos, com o intuito de recuperá-los, aumentando o atraso e a latência da conexão. Porém, as aplicações multimídia suportam bem a perda de pacotes, não sendo necessária a retransmissão dos mesmos. O maior problema é o atraso que os pacotes podem sofrer, pois estas aplicações não se comportam bem com estes atrasos, havendo saltos na reprodução da mídia, o que pode torná-la incompreensível. Por este motivo, o serviço de entrega confiável do TCP faz com que este protocolo não seja o melhor para aplicações multimídia na Internet.

O outro serviço oferecido pelo TCP, o controle de congestionamento, reduz a transmissão de pacotes quando os enlaces encontram-se congestionados, o que também irá aumentar o atraso fim-a-fim. Apesar deste serviço do TCP ser ruim para as aplicações multimídia, ele deverá ser analisado com mais atenção, pois se todas as aplicações na Internet enviarem pacotes ocupando toda a banda, chegará um momento em que o congestionamento será tão grande que nenhuma aplicação conseguirá enviar e nem receber mais nada. Por isso, mesmo os protocolos que não possuem o serviço de controle de congestionamento, deverão passar esta responsabilidade para alguém, como a aplicação.

Por todos esses motivos, apesar do TCP ser um protocolo eficiente no envio de pacotes que não podem ser perdidos, ele não é tão eficiente quando esses pacotes são de mídia em tempo real de uma aplicação com pouca tolerância ao atraso de pacotes. O protocolo mais usado para esse tipo de aplicação é o UDP [13].

2.2 UDP (User Datagrama Protocol)

O UDP [13] é um protocolo de transporte simplificado, leve, com um modelo de serviço minimalista. Ele é um serviço não orientado a conexão; portanto, não há apresentação antes que os dois processos comecem a se comunicar. Ele fornece um serviço de transferência de dados não confiável, isto é, não oferece nenhuma garantia de que a mensagem enviada alcançará a porta receptora. Além disso, as mensagens que, de fato, chegam à porta receptora podem chegar fora de ordem.

Por outro lado, o UDP não inclui um mecanismo de controle de congestionamento; portanto, o processo origem pode transmitir dados para dentro de uma porta de UDP na taxa que quiser. Embora nem todos os dados consigam alcançar a porta receptora devido a dificuldades enfrentadas em sua viagem (por exemplo, tráfego na rede), uma grande fração deles consegue chegar. Da mesma maneira que o TCP, o UDP não oferece garantias em relação aos atrasos.

Como exemplo de aplicações, temos a telefonia por Internet que em geral roda em UDP. Cada lado de uma aplicação de telefonia por Internet precisa enviar os dados pela rede a uma taxa mínima. E, também, as aplicações de telefonia por Internet são tolerantes às perdas.

O UDP faz atividades mínimas considerando o que um protocolo de transporte pode fazer. Fora sua função de multiplexação/demultiplexação e de alguma verificação de erros, ele nada adiciona ao Internet Protocol (IP) [14]. Considerando o apresentado, se o criador da aplicação escolher o UDP em vez do TCP, a aplicação estará se comunicando quase que diretamente com o IP.

Segue abaixo o fluxo de Transmissão do Protocolo UDP:

1. O UDP obtém as mensagens do processo de aplicação, anexa os campos de número de porta da fonte e do destino, adicionam dois outros pequenos campos e passa o segmento resultante à camada de rede.

2. A camada de rede encapsula o segmento dentro de um datagrama IP e, em seguida, faz uma tentativa de melhor esforço para entregar o segmento ao hospedeiro receptor.

3. Se o segmento chega ao hospedeiro receptor, o UDP usa os números de porta para entregar os dados do segmento ao processo de aplicação correto.

Note que, com o UDP, não há sessão de apresentação entre as entidades remetente e destinatário da camada de transporte antes de enviar um segmento. Por essa razão, dizemos que o UDP é não orientado à conexão.

O protocolo UDP tem as seguintes características:

- Não há estabelecimento de conexão. Diferente do protocolo TCP o UDP não utiliza o protocolo de apresentação (3-way handshake) antes de começar a transferir dados. O UDP simplesmente envia mensagens sem nenhuma preliminar formal. Assim, ele não introduz nenhum atraso para estabelecer uma conexão.
- Não há estado de conexão. O TCP mantém o estado de conexão nos sistemas finais. Esse estado inclui *buffers* de envio e recebimento, parâmetros de controle de congestionamento e parâmetros numéricos de seqüência e de reconhecimento. O UDP, por sua vez, não mantém o estado de conexão, por essa razão, um servidor devotado a uma aplicação específica pode suportar mais clientes ativos quando a aplicação roda sobre UDP do que quando roda sobre TCP.
- Pequeno *overhead* no cabeçalho do pacote. O segmento UDP tem somente 8 *bytes* de acréscimo.
- Taxa de envio não regulada. A taxa com a qual o UDP envia dados é limitada somente pela taxa com que a aplicação gera dados, pelas capacidades da fonte (CPU e freqüência de relógio) e pela largura de banda de acesso à Internet. Devemos ter em mente, no entanto, que o hospedeiro destinatário não recebe necessariamente todos os dados. Assim, a taxa de recebimento pode ser limitada pelo congestionamento da rede, mesmo que não haja limitação na taxa de envio. Esta característica é o principal motivo pelo qual aplicações em tempo real utilizam o UDP,

pois estas aplicações podem tolerar alguma perda de pacotes, mas exigem uma taxa mínima de envio.

O UDP é consideravelmente usado em aplicações de multimídia, como telefone por Internet, videoconferência em tempo real e recepção de áudio e vídeo armazenados. Essas aplicações podem tolerar uma pequena fração de perda de pacotes, de modo que a transferência confiável de dados não é absolutamente fundamental para o sucesso da aplicação. Além disso, aplicações em tempo real, como telefone por Internet e videoconferência, reagem muito mal ao controle de congestionamento do TCP.

Embora atualmente seja comum rodar aplicações de multimídia sobre UDP, devemos lembrar que, como já foi mencionado, o UDP não tem controle de congestionamento. Mas esse controle é necessário para evitar que a rede seja sobrecarregada. Se todos começassem a enviar vídeo com alta taxa de bits sem usar nenhum controle de congestionamento, haveria tamanha perda de pacotes nos roteadores que ninguém conseguiria ver nada. Assim, a falta de controle de congestionamento no UDP é um problema bastante sério. Por isso existem muitas propostas de novos mecanismos para forçar todas as fontes, inclusive as fontes UDP, a realizar um controle de congestionamento adaptativo.

Mesmo com a simplicidade do protocolo UDP, devemos mencionar que algumas de suas limitações podem ser contornadas no nível de aplicação, claro que de acordo com as necessidades da aplicação. Por isso, é possível que uma aplicação tenha transferência confiável de dados usando UDP (por exemplo, adicionando mecanismos de reconhecimento e de retransmissão). Assim os processos de aplicação podem se comunicar de maneira confiável sem ter de se sujeitar às limitações da taxa de transmissão impostas pelo mecanismo de controle de congestionamento do TCP, mas dispõem de reconhecimentos e retransmissões embutidos na aplicação para reduzir a perda de pacotes.

2.3 RTP (Real Time Protocol)

Nesta sessão será descrito o RTP [11], o protocolo de tempo real. RTP disponibiliza funções de transporte fim-a-fim para aplicações que transmitem dados em tempo real, como áudio e vídeo, sobre serviços de comunicação *multicast* ou *unicast*. RTP não faz reserva de recursos e não garante a qualidade do serviço para aplicações em tempo real. O transporte de dados é suportado por um protocolo de controle, RTCP [11] (*Real Time Control Protocol*), para permitir o monitoramento da entrega dos dados de uma maneira escalável às grandes

redes de *multicast*, e para prover a funcionalidade mínima de controle e identificação. RTP foi desenvolvido para ser independente da camada de transporte.

O RTP comumente roda sobre UDP. Porções de dados de áudio e vídeo gerados pelo lado remetente de uma aplicação de multimídia são encapsuladas em pacotes RTP. Cada pacote RTP, por sua vez, é encapsulado em um segmento UDP. Como o RTP fornece serviços (como marcas de tempo e números de seqüência) às aplicações de multimídia, ele pode ser visto como uma subcamada de transporte.

Cabe ao desenvolvedor integrar o RTP à aplicação, uma vez que o RTP não pertence a camada de transporte, mas sim da camada de aplicação. Para o lado remetente da aplicação, o desenvolvedor deve implementar a criação dos pacotes de encapsulamento RTP. A aplicação então envia os pacotes RTP para dentro da porta de interface UDP. De maneira semelhante, no lado receptor da aplicação, os pacotes RTP entram na aplicação através de uma porta de interface UDP. A extração das porções de mídia dos pacotes RTP também deve ser implementada no lado receptor.

Se uma aplicação incorporar o RTP, em vez de um esquema proprietário para fornecer tipo de carga útil (veja tabela 1), número de seqüência ou marcas de tempo, ela inter-operará mais facilmente com as outras aplicações de rede multimídia.

Tipos de Mídia
AIFF (.aiff)
AVI (.avi)
GSM (.gsm)
HotMedia (.mvr)
MIDI (.mid)
MPEG-1 Video (.mpg)
MPEG Layer II Audio (.mp2)
QuickTime (.mov)
Sun Audio (.au)
Wave (.wav)

Tabela 1: Tipos de formatos de mídia como carga útil

Devemos enfatizar que o RTP, em si, não fornece nenhum mecanismo que assegure a entrega de dados a tempo nem fornece outras garantias de qualidade de serviço; ele não garante nem mesmo a entrega dos pacotes nem evita a entrega de pacotes fora de ordem. Na verdade, o encapsulamento realizado pelo RTP somente é visto nos sistemas finais. Os

roteadores não distinguem os datagramas IP que carregam pacotes RTP dos datagramas IP que não os carregam.

O RTP permite que seja atribuída a cada fonte (por exemplo, uma câmera ou um microfone) sua própria e independente corrente de pacotes RTP. Por exemplo, para uma videoconferência entre dois participantes, quatro correntes RTP podem ser abertas, duas correntes para transmitir o áudio (uma em cada direção) e duas correntes para vídeo (novamente, uma em cada direção), como mostrado na figura 1. Contudo, muitas técnicas de codificação populares, incluindo MPEG1 (*Motion Picture Experts Group-1)* e o MPEG2 (*Motion Picture Experts Group-2*), conjugam áudio e vídeo em uma única corrente durante o processo de codificação. Quando áudio e vídeo são conjugados pelo codificador, somente uma corrente RTP é gerada em cada direção.

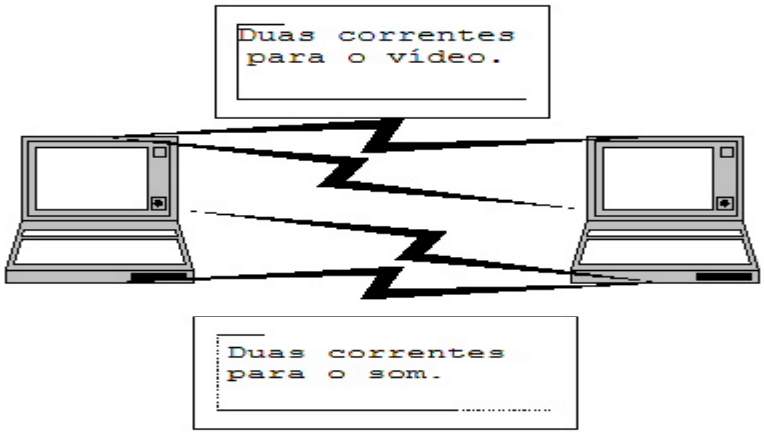

Figura 1: Correntes de transmissão de áudio e vídeo

Os pacotes RTP não são limitados às aplicações *unicast*. Eles podem também ser enviados sobre árvores *multicast*, um-para-muitos e muitos-para-muitos. Para uma sessão multicast muitos-para-muitos, todos os remetentes e fontes da sessão em geral usam o mesmo grupo *multicast* para enviar suas correntes RTP. As correntes *multicast* RTP que existem em conjunto, como áudio e vídeo que emanam de múltiplos remetentes em uma aplicação de videoconferência, pertencem a uma sessão RTP.

2.5 RTCP (Real Time Control Protocol)

O protocolo RTCP [11] é baseado em transmissões periódicas de pacotes para todos os participantes da sessão, usando o mesmo mecanismo de distribuição de pacotes de dados. O protocolo subjacente providencia a multiplexação de dados e pacotes de controle, como por exemplo, usando números de porta separadas com UDP.

RTCP é um protocolo que uma aplicação de rede multimídia pode usar juntamente com o RTP. Os pacotes RTCP são transmitidos por cada participante de uma sessão RTP para todos os outros participantes da sessão usando IP *multicast*. Para uma sessão RTP há, tipicamente, um único endereço *multicast* e todos os pacotes RTP e RTCP pertencentes à sessão usam o endereço *multicast*. Os pacotes RTP e RTCP se distinguem uns dos outros pelo uso de números de porta distintos.

Os pacotes RTCP não encapsulam porções de áudio ou de vídeo. Em vez disso, eles são enviados periodicamente e contêm relatórios de remetente e/ou receptor com dados estatísticos que podem ser úteis para a aplicação. Esses dados estatísticos contêm número de pacotes enviados, número de pacotes perdidos e variação de atraso entre chegadas. A especificação RTP [11] não determina qual aplicação deve realizar essa realimentação de informação; isso depende do desenvolvedor da aplicação. Os remetentes podem usar as informações de realimentação, por exemplo, para modificar suas taxas de transmissão. A realimentação de informações também pode ser utilizada para finalidades de diagnósticos; por exemplo, receptores podem determinar se os problemas são locais, regionais ou globais.

2.6 Protocolos no Nível de Aplicação

Os protocolos no nível de aplicação são muito importantes, pois são eles que realizam quase todo o trabalho de comunicação entre as aplicações multimídia. Já que são eles que sincronizam as aplicações permitindo que os eventos aconteçam de forma esperada.

A construção de um protocolo envolve o conhecimento exato do que se deve fazer em cada momento para que a aplicação haja como esperado. Como exemplo de protocolo de aplicação, pode-se tomar a comunicação entre transmissor, servidor e receptores da arquitetura apresentada no próximo Capítulo, a arquitetura com servidor refletor, ilustrada pela figura 2.

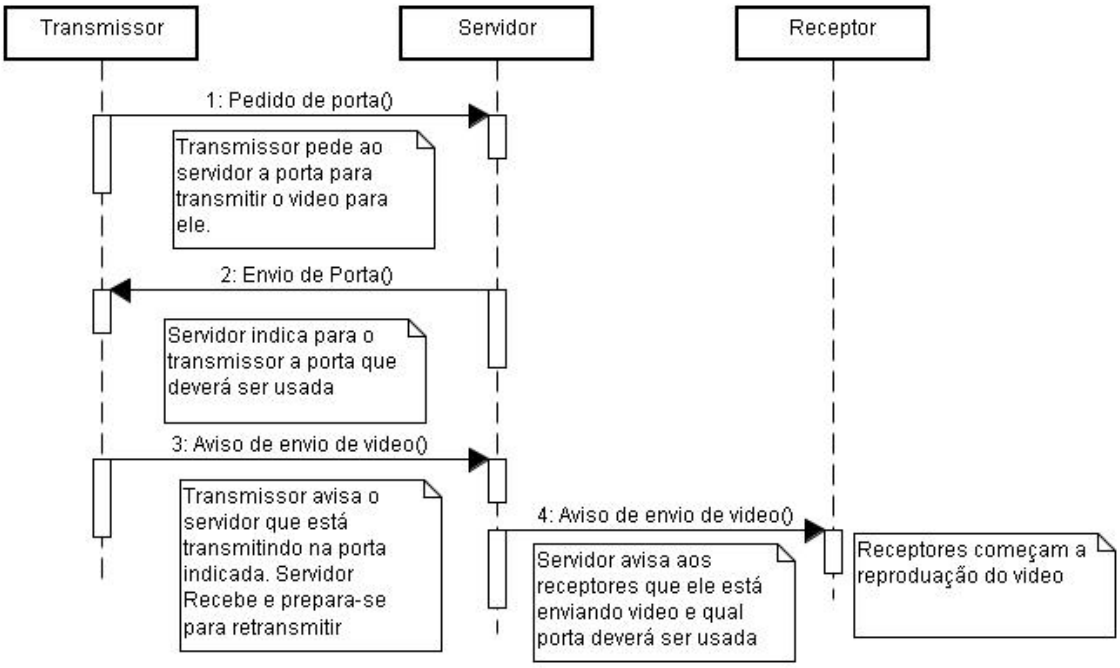

Figura 2: Diagrama de seqüência do protocolo de transmissão de vídeo da arquitetura com servidor refletor

O Transmissor, ante de poder enviar o vídeo capturado, precisa saber por qual porta enviá-lo, por isso envia uma mensagem de pedido para o servidor, que gerencia as portas para a transmissão de vídeo. O servidor verifica qual porta está disponível e envia uma mensagem para o transmissor avisando qual porta foi escolhida. O transmissor, então, começa a transmitir por esta porta para o servidor, e envia uma mensagem para o mesmo avisando sobre o início da transmissão. O servidor, recebendo a mensagem de aviso, inicia o recebimento do vídeo pela porta escolhida, e em seguida o retransmite para os respectivos receptores. O servidor deve ainda enviar uma mensagem para os receptores avisando-os de que eles devem reproduzir o vídeo enviado pelo servidor nas portas escolhidas pelo próprio servidor. Os receptores, ao receberem a mensagem de aviso do servidor, iniciam reprodução do vídeo através da porta indicado pelo servidor.

Este pequeno protocolo demonstra a importância dos protocolos no nível de aplicação para as aplicações de rede. Somente protocolos bem definidos garantem o bom funcionamento das aplicações.

CAPÍTULO 3 – ARQUITETURAS

3.1 Introdução

As arquiteturas de comunicação, utilizadas pelas aplicações possuem papel fundamental na viabilização dos serviços de trocas de pacotes, principalmente quando se trata de pacotes de mídia. A escolha da arquitetura correta em determinada aplicação irá otimizar a troca de pacotes permitindo que aplicações, como uma ferramenta de videoconferência em tempo real, reduza o atraso fim-a-fim possibilitando o funcionamento com uma razoável qualidade de recepção.

As implementações dessas arquiteturas poderão se dar tanto no nível de aplicação, quanto no nível de rede da Internet. Por exemplo, um roteamento *multicast* poderia se dar, no nível de rede da Internet, através dos protocolos *Internet Group Management Protocol* (IGMP) [12] e *Distance Vector Multicast Routing Protocol* (DVMRP) [7], onde o transmissor não teria a necessidade de replicar o pacote transmitido para todos os receptores. Bastaria apenas enviar um pacote e os roteadores da rede se encarregariam, através dos grupos multicast utilizado pelo IGMP e pelo DVMRP, de enviar um pacote para cada receptor, aproveitando melhor a largura da banda.

Várias arquiteturas já foram propostas e quanto melhor a eficiência na troca de pacotes, maior a complexidade de implementação das mesmas. Por isso deve-se ter cuidado ao se escolher uma determinada arquitetura para uma aplicação.

Neste Capítulo quatro tipos de arquiteturas serão apresentados, no nível de aplicação, para distribuição de pacotes de mídia em uma rede, como a Internet. Começa-se dos modelos de arquitetura mais simples, o ponto a ponto. Em seguida descreve-se um modelo que centraliza o roteamento dos pacotes refletindo-os para os receptores. Analisa-se também um modelo que otimiza a arquitetura anterior, minimizando o caminho percorrido pelos pacotes e por fim mostra-se uma versão dessa arquitetura com uma maior capacidade escalar.

3.2 Arquitetura ponto a ponto

É a mais simples das arquiteturas que se apresentará. Nesta arquitetura a transmissão se dá apenas entre dois *hosts*, ou seja, a infra-estrutura da rede é utilizada apenas para a comunicação e troca de pacotes por dois *host* diferentes. Como se um túnel imaginário os conectasse diretamente, como mostra a figura 3.

Com essa arquitetura tem-se o melhor modelo de troca de pacotes que se pode ter entre apenas dois *hosts*.

O transmissor, *host* 1, irá enviar os pacotes de media para o receptor, *host* 2, através da rede se utilizando o protocolo RTP [11]. Como mencionado anteriormente, para o transmissor seria como se houvesse um *link* ligando-o diretamente ao receptor. Contudo os pacotes irão viajar pela infra-estrutura real da rede, cabendo aos roteadores decidirem por quais diferentes enlaces estes pacotes deverão passar para que cheguem ao seu destino final.

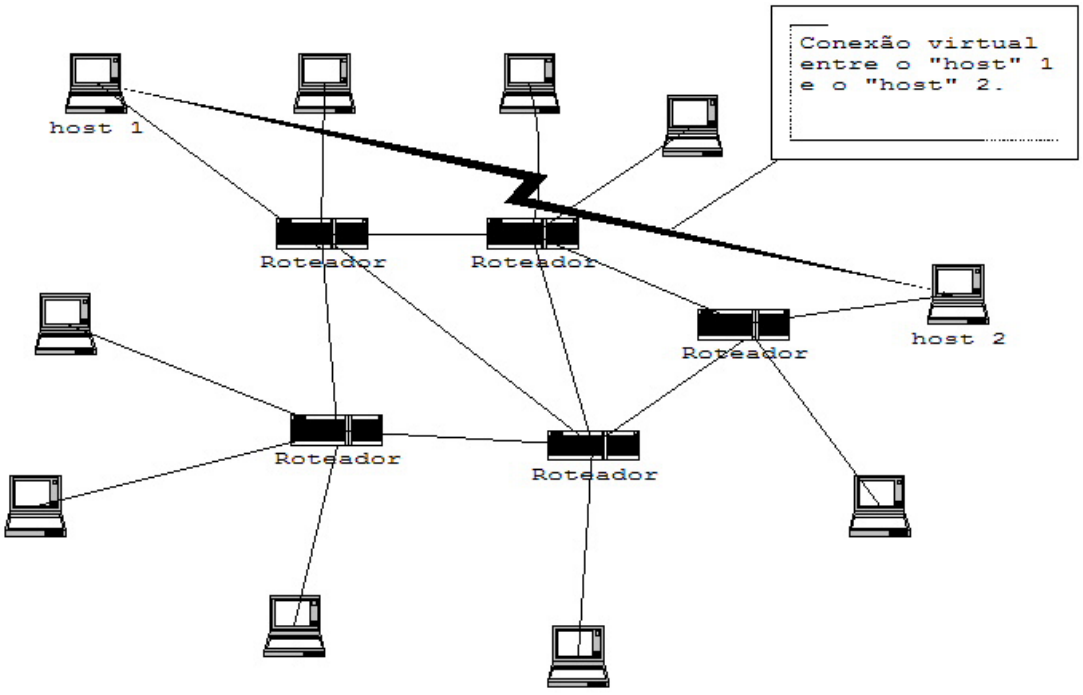

Figura 3: Conexão ponto-a-ponto sobre a rede

Como a rede está sendo utilizada por várias outras aplicações, vários pacotes utilizados por elas também estão em trânsito. Alguns enlaces podem ficar temporariamente congestionados e outros livres, fazendo com que pacotes de uma mesma aplicação tomem caminhos diferentes através da rede ou no pior caso, sejam perdidos.

Estes problemas são transparentes para o transmissor e para o receptor, que apenas sabem da existência um do outro, já que estão implementados no nível de aplicação da Internet, e não tem conhecimento sobre o difícil trabalho dos roteadores no envio dos pacotes do destino à fonte, como mostrado na figura 4.

Deste modo pode-se observar que o menor atraso fim-a-fim total, entre o transmissor e o receptor, é crucial para o bom desempenho da aplicação multimídia. Até nessa simples arquitetura ponto-a-ponto, se o atraso fim-a-fim total for grande, a aplicação não irá corresponder às expectativas dos usuários. Havendo pulos de quadros ou som picado por atraso ou falta dos pacotes de mídia que estão viajando pela rede. Como irá se ver logo em seguida, manter este atraso em um patamar baixo, para aplicações multipontos, pode ser ainda mais difícil, dependendo da arquitetura utilizada.

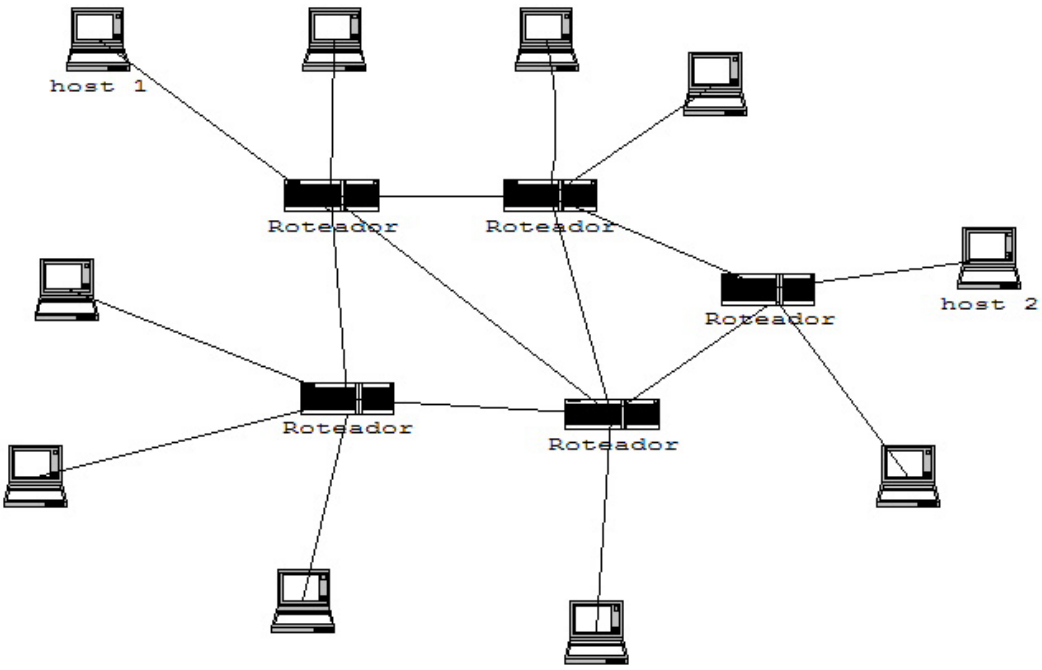

Figura 4: Organização dos *hosts* e roteadores na Internet

3.3 Arquitetura com servidor refletor

Está é a primeira arquitetura multiponto que irá ser apresentada [1], pode-se encontrar a utilização de uma arquitetura semelhante em [18]. A idéia inicial é a de se ter várias arquiteturas ponto a ponto, mas todos os *hosts* estariam se conectando a um mesmo *host* central, que chamaremos de servidor, como mostra a figura 5.

Com essa arquitetura tenta-se criar grupos *multicast,* no nível da aplicação, para a realização de *multicast* dos pacotes de mídia. Na intenção de oferecer um melhor aproveitamento da largura de banda oferecida pela rede, pois apenas um pacote de mídia seria enviado, pelo transmissor. O servidor iria se encarregar de enviar para cada receptor os pacotes de media, na tentativa de diminuir o tráfico total da rede.

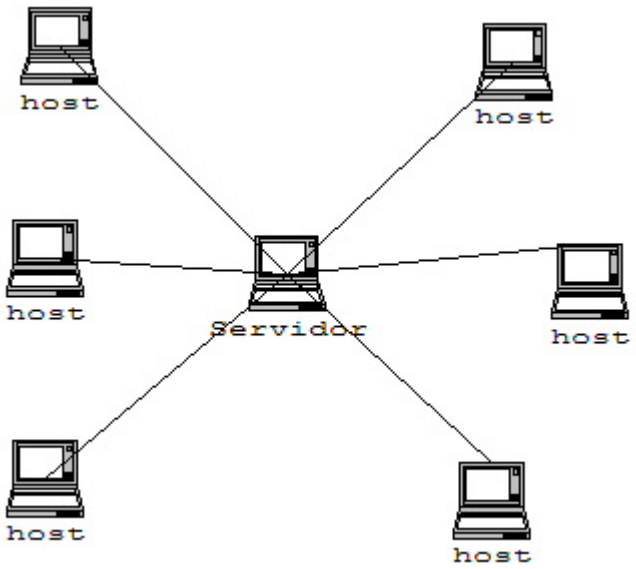

Figura 5: Diagrama de conexão da arquitetura com servidor refletor

Como todos os *hosts* estão conectados ao servidor, este centraliza todas as informações relevantes dos *hosts* e gerencia as conexões que serão realizadas.

Contudo, o principal papel do servidor, nesta arquitetura, é o de redistribuir os pacotes de mídia, enviados pelo transmissor, para todos os receptores. Se comportando como um refletor de pacotes ele replica o pacote enviado, deste modo a largura de banda é melhor aproveitada, já que o transmissor não tem que enviar um pacote para cada receptor, deixando está tarefa para o servidor refletor.

Contudo, podemos observar que o servidor deverá possuir várias conexões, uma para cada *host*, sendo, deste modo, onerado por ter que controlar tanto as conexões de gerência, como as conexões de envio de pacote de mídia.

Outro problema que pode surgir é do caminho percorrido pelos pacotes de mídia, desde o transmissor até o receptor. Como todos os pacotes que o transmissor gera são enviados para o servidor refletor, e só depois eles são enviados para os receptores, pode ocorrer de que, fisicamente, o transmissor e o servidor estejam longe um do outro, e que o transmissor e o receptor estejam perto, como mostra a figura 6. Isso pode ocorrer porque as

conexões entre os *hosts* e o servidor são conexões realizadas no nível de aplicação da rede, que abstraem o modo como a estrutura de enlaces e roteadores da rede realiza essas conexões entre os *hosts*.

Desde modo o pacote de mídia terá que viajar até o servidor, para que este o envie ao receptor, sendo que se o pacote fosse enviado diretamente do transmissor para o receptor, o caminho percorrido teria sido menor. Quando isso ocorre, o atraso fim-a-fim pode ser maior, já que o pacote tem que passar por vários roteadores e enlaces.

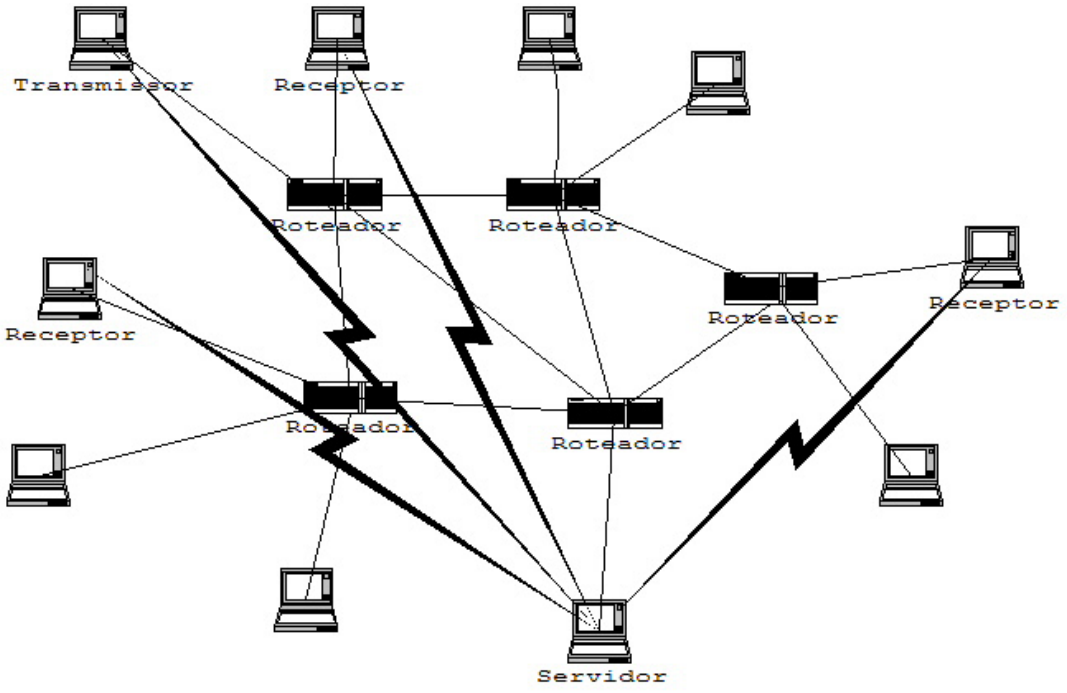

Figura 6: de conexões da arquitetura com servidor refletor sobre a Internet

A arquitetura com um servidor refletor diminui o tráfico desnecessário de pacotes de mídia pela rede, aproveitando melhor a largura de banda oferecida, até certo ponto. Contudo, o fato de centralizar a distribuição dos pacotes, pode fazer com que o caminho percorrido por eles seja maior do que realmente seria se o transmissor enviasse diretamente o pacote de mídia para o receptor.

A seguir mostra-se como uma transmissão de pacotes de mídia se comportaria em uma arquitetura que não centraliza a transmissão, suas vantagens e desvantagens, e como melhorar esta abordagem.

3.4 Arquitetura sem servidor refletor

Nesta arquitetura, tenta-se resolver o problema do caminho percorrido pelos pacotes de mídia fazendo com que cada *host* transmissor, possua uma conexão direta com cada *host* receptor. Deixando de lado a função do servidor de refletir os pacotes enviados pelo transmissor.

Então, cada transmissor possuirá, além de uma conexão com o servidor, uma conexão com cada um dos outros *hosts* que sejam receptores, como mostra a figura 7. A conexão com o servidor servirá para facilitar as atividades de gerência da aplicação, enquanto, como já havia sido dito, as conexões diretas entre os *hosts* servirá para o envio e recebimento dos pacotes de mídia.

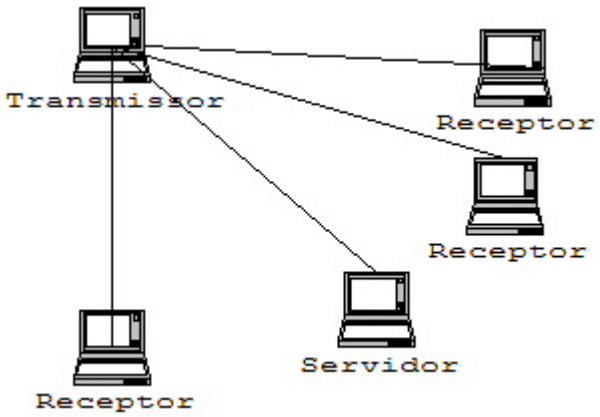

Figura 7: Diagrama de conexões da arquitetura sem servidor refletor

Deste modo tenta-se evitar com que os pacotes de mídia percorram um caminho maior do que realmente seria necessário. Como se explicou anteriormente pode ser que fisicamente o transmissor e os receptores estejam perto uns dos outros, mas todos longe do servidor (que funciona como um refletor de pacotes na arquitetura anterior) como mostra a figura 8. Neste cenário, os pacotes de mídia teriam que ir do transmissor até o servidor, este por sua vez verificaria quais os *hosts* que estão esperando a recepção deste pacote e o retransmitiria, replicando o pacote, um para cada receptor.

Esse caminho completo, transmissor - servidor – receptor, será sempre maior que o caminho transmissor – receptor, fazendo com que o pacote viaje mais, aumentando o atraso fim-a-fim e também, a probabilidade de perda do mesmo. Já no cenário onde o transmissor possui uma ligação direta com o receptor, o caminho percorrido pelos pacotes na maioria das vezes será menor, diminuído o atraso da entrega dos pacotes, como mostra a figura 9. E no

pior caso, o caminho percorrido será equivalente ao da arquitetura com refletor, caso o transmissor e o receptor estejam distantes.

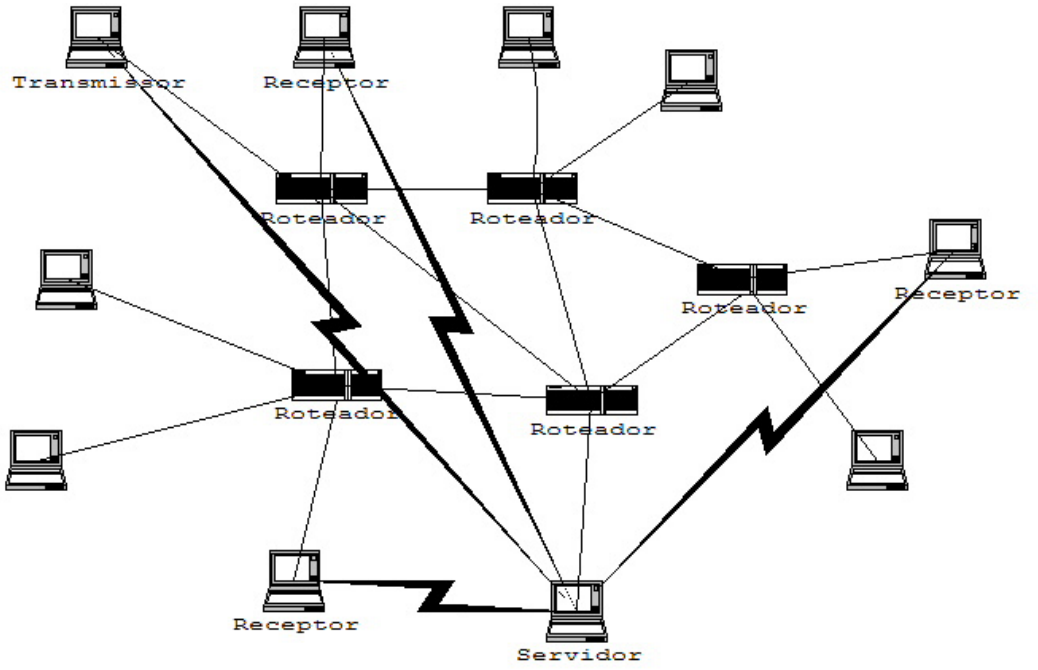

Figura 8: Digrama de conexões da arquitetura com servidor refletor sobre a Internet

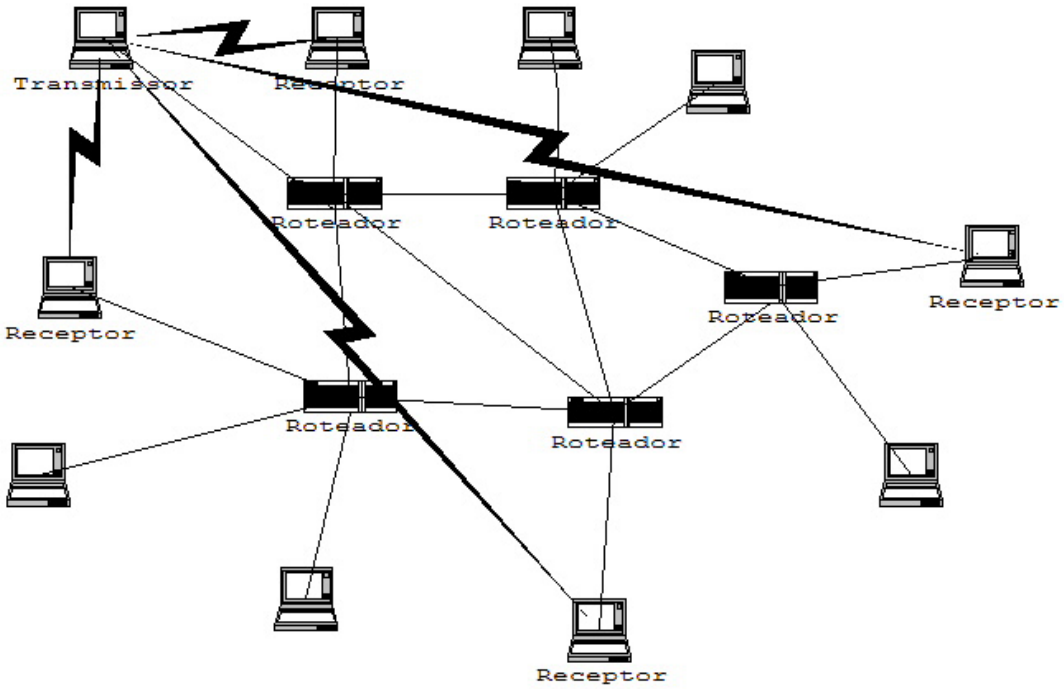

Figura 9: Diagrama de conexões da arquitetura sem servidor refletor sobre a Internet

Um problema, que se pode observar nesta arquitetura, é o fato do transmissor ter que manter uma conexão com cada receptor, além da conexão com o servidor. Há na verdade dois problemas com essas conexões. O primeiro é o fato de que o *host* transmissor pode não suportar um número muito grande de conexões com os receptores para o envio dos pacotes. O segundo seria o fato de que o transmissor enviaria várias cópias do mesmo pacote pela rede, um para cada receptor ocupando a banda passante, aumentando o tráfico e provavelmente o atraso fim-a-fim.

A arquitetura apresentada resolve o problema de viagem dos pacotes de mídia, o que faz desta arquitetura a que nos oferece o menor atraso fim-a-fim, mas nos traz um problema de escala, que havia em menor proporção na arquitetura anterior já que o servidor pode ser mais robusto que os *hosts* transmissores. Por esse motivo, devemos fazer algumas modificações nesta arquitetura para resolver esse problema de escala e torná-la melhor.

3.5 Arquitetura com receptores refletores

A idéia desta arquitetura é semelhante a anterior, mas com algumas melhorias. Ao invés do transmissor manter uma conexão para cada receptor, ele manteria apenas algumas conexões com alguns receptores, e alguns desses receptores se comportariam como um refletor para outros receptores, melhorando a capacidade escalar da arquitetura anterior, como ilustrado na figura 10. Uma aplicação popular na Internet que utiliza uma arquitetura semelhante é o BitTorrent [2].

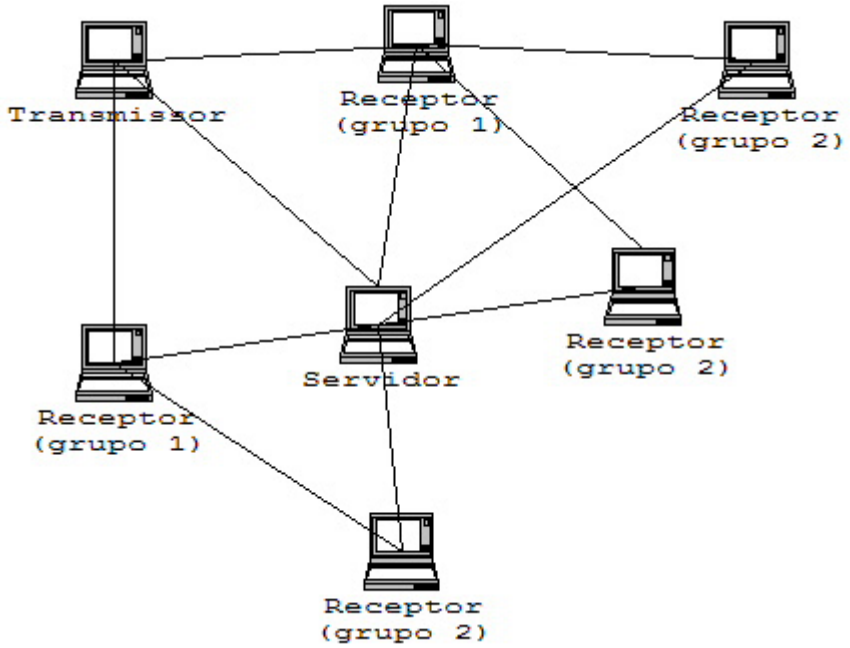

Figura 10: de conexões da arquitetura com receptores refletores

Como se pode observar, todos os *hosts* continuam mantendo uma conexão com o servidor, que tem que continuar a gerenciar a comunicação entre os *hosts*. Porém, o transmissor irá apenas manter algumas conexões com alguns dos receptores que esperam os pacotes enviados por ele. Esses receptores mantêm conexões com outros receptores que também esperam pacotes de mídia do transmissor, mas não possuem uma conexão direta com ele. O primeiro grupo de receptores, que mantém conexões diretas com o transmissor, irá replicar os pacotes de mídia para o segundo grupo de receptores, os que não possuem conexões diretas com o transmissor, mas possuem conexões com o primeiro grupo de receptores.

Deste modo continuamos resolvendo o problema do caminho percorrido pelos pacotes de mídia, que irão diretamente do transmissor para o primeiro grupo de receptores, que estão fisicamente mais perto, e em seguida irão para o segundo grupo de receptores, que estão fisicamente mais longe do transmissor, como mostra a figura 11. E melhoramos o problema de escala do transmissor, que passará a replicar menos pacotes para a rede e poderá possuir um número indefinido de receptores esperando seus pacotes de mídia. Sem que com isso o tráfico na rede seja muito prejudicado por causa dos pacotes de mídia enviados pelo transmissor.

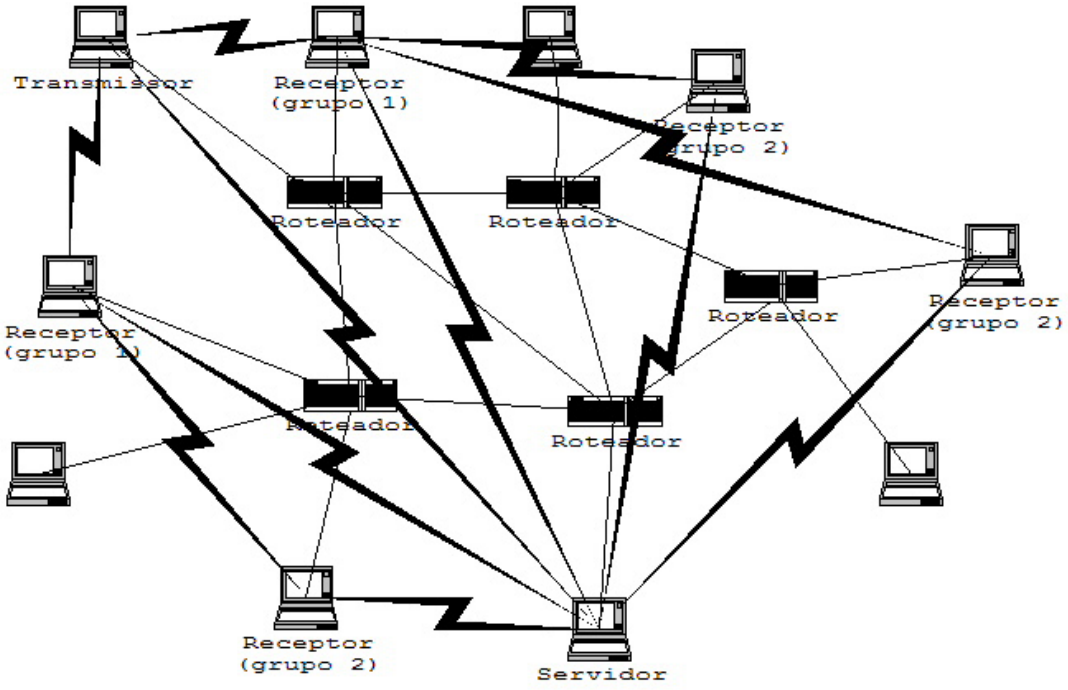

Figura 11: Diagrama de conexões da arquitetura com receptores refletores sobre a Internet

Com essa análise, pode-se observar que esta arquitetura seria a melhor opção, dentre as arquiteturas multipontos apresentadas, para envio de pacotes de mídia de um transmissor para múltiplos receptores. Mas com certeza haverá uma maior complexidade na implementação desta arquitetura, já que o servidor deverá escolher quais dos receptores farão parte do primeiro grupo de receptores, quais farão parte do segundo grupo e quais do terceiro grupo, caso haja a necessidade de se dividir em mais subgrupos. E toda essa análise deverá ser repassada para o transmissor, para que este saiba para quem começar a enviar os pacotes, e para os receptores saberem de quem esperar os pacotes e para quem replicar, caso seja necessário.

CAPÍTULO 4 – FERRAMENTAS DO PROJETO

4.1 Introdução

Com a grande popularidade que a linguagem Java vem ganhando nos últimos anos, vários grupos de desenvolvedores criaram diversas ferramentas e *frameworks* que facilitassem a outros desenvolverem aplicações de modo mais rápido, fácil e com maior robustez.

Neste cenário, inúmeros *frameworks* surgiram. Nesse capítulo, dois deles serão analisados, o JMF e o Log4J, que ajudaram a desenvolver a aplicação de análise.

4.2 Java Media Framework (JMF)

A grande demanda de recursos multimídia na Internet e a necessidade dos aplicativos Java de incorporarem esses recursos, a *Sun Microsystems*, a *Intel* e a *Silicon Graphics* desenvolveram juntas uma *Application Programming Interface* (API) para multimídia que facilitasse aos desenvolvedores Java, reproduzir, transmitir, editar e capturar os formatos mais populares de mídia. Neste contexto foi criada a *Java Media Framework* (JMF) [5] que até o momento de finalização deste capítulo se encontrava na versão 2.1.1.

As facilidades proporcionadas pela JMF são a reprodução, transmissão, captura e edição dos formatos de mídia mais comuns, como *Microsoft Audio/Video Interleave* (**.avi**), *Macromedia Flash 2 movies* (**.swf**), *Future Splash* (**.spl**), *MPEG Layer 3 Audio* (**.mp3**), *Musical Instrument Digital Interface* (MIDI; **.mid**), *vídeos MPEG-1* (**.mpeg**, **.mpg**), *QuickTime* (**.mov**), *Sun Audio* (.**au**), *áudio Wave* (**.wav**), *AIFF* (**.aiff**) e *GSM* (**.gsm**).

Na aplicação desenvolvida neste trabalho, foram usadas algumas importantes classes deste poderoso *framework*, dentre elas estão a **Player**, **Controller**, e a **Manager**. Através da classe **Player** conseguimos reproduzir, facilmente, qualquer tipo de formato suportado pelo JMF. A classe **Player** estende a classe **Controller**, que possui diversos recursos de configuração da reprodução da mídia. Contudo a classe **Player** simplifica a criação do *player* para a reprodução da mídia. A criação de um *player* para a reprodução de qualquer tipo de

formato de mídia suportado pelo JMF, passa ser uma tarefa muito menos árdua, como será demonstrado abaixo.

A classe **JPlayer** da aplicação cliente implementada neste trabalho, utiliza muitos recurso da JMF. Para a criação do *player* pela classe **JPlayer** iremos usar a classe **Player** da JMF, como mostrado na figura 12.

```
...
// endereço da origem da midia
MediaLocator mediaLocator = new MediaLocator( address );

if ( mediaLocator == null ) {
    showErrorMessage( "Error opening file" );
    return;
}

// cria um player a partir de MediaLocator
try {
    player = Manager.createPlayer( mediaLocator );

   // registra ControllerListener para tratar de eventos do Player
  player.addControllerListener( new PlayerEventHandler() );
...
```

Figura 12: Criação da classe *Player* da JMF

Pode-se observar no exemplo, que declaramos um *player* da classe **Player**, o qual será usado para reproduzir uma mídia contida em um arquivo. Primeiramente recuperamos o arquivo que será usado para reprodução, podendo ser qualquer um dos formatos mencionados acima, através da obtenção de um *media locador* do arquivo pela instanciação a classe **MediaLocator**. Essa classe gera um *Uniform Resource Locator* (URL) que o *framework* precisa para localizar o arquivo no sistema de arquivos ou na rede.

Em segundo lugar, através do método **createPlayer** da classe **Manager**, instancia-se a classe **Player** e registra-se o **ControllerListener** para tratar os eventos da classe **Player.**

Através dos eventos da classe **Player**, carrega-se a mídia, com o método **realize()**. E prepara-se o *player* para a execução da mídia, com o método **prefetch().** Neste momento, obtém-se os componentes visuais e de controle do *player* e inicia-se a reprodução da mídia através do método **start()**.

Quando o arquivo de mídia chegar ao fim, o evento *endOfMedia* será chamado pelo *player* e o método **stop()** será executado interrompendo a reprodução do arquivo de mídia, como mostra a figura 13.

```
// tratador para os eventos ControllerEvents do player
private class PlayerEventHandler extends ControllerAdapter {

    // carrega antecipadamente a mídia assim que o player é realizado
   public void realizeComplete( RealizeCompleteEvent realizeDoneEvent ){
   player.prefetch();
   }

  // player pode começar a mostrar a mídia após a carga antecipada
  public void prefetchComplete( PrefetchCompleteEvent prefetchDoneEvent ){
       getMediaComponents();

       // assegura um leiaute válido para a frame
       validate();

       / / começa a reproduzir a mídia
      player.start();

  } // fim do método prefetchComplete

  // se fim da mídia, restaura para o início e pára de reproduzir
  public void endOfMedia( EndOfMediaEvent mediaEndEvent ){
        player.setMediaTime( new Time( 0 ) );
        player.stop();
    }
} // fim da classe interna PlayerEventHandler
```

Figura 13: Controle dos eventos do player

Como podemos observar na figura 13, ao se iniciar o evento de *prefetch* do *player* o método **getMediaCompontents()** da classe **JPlayer** é chamado para que os componentes visuais do *player* apareçam para o usuário, como mostrado na figura 14.

```
// obtém controles visuais para mídia e player
public void getMediaComponents(){
    // obtém componente visual do player
    visualMedia = player.getVisualComponent();

    // adiciona componente visual, se estiver presente
    if ( visualMedia != null )
        container.add( visualMedia, BorderLayout.CENTER );

    // obtém a GUI de controle do player
    mediaControl = player.getControlPanelComponent();

    // adiciona componente de controles, se estiver presente
    if ( mediaControl != null )
        container.add( mediaControl, BorderLayout.SOUTH );

} // fim do método getMediaComponents
```

Figura 14: Método de criação dos componentes visuais do *player*

O método **getVisualComponent()** irá recuperar do JMF os componentes necessários para que a reprodução da mídia possa ser exibida para o usuário. Já o método **getControlPanelComponent()** irá recuperar os botões de controle da mídia em execução, como o *pause* mostrado na figura 15.

Figura 15: Componentes visuais do *player* recuperados do JMF

Essa breve demonstração é apenas um pequeno exemplo do potencial da JMF e de suas classes. Muito mais pode ser feito, ficando a cargo da imaginação dos desenvolvedores e projetistas. Nos anexos, podem-se encontrar alguns exemplos de usos das classes da JMF que se utilizou na aplicação implementada para este trabalho.

4.3 – Log4J

Existe uma grande necessidade de geração de *logs* nas aplicações, tanto para depuração, quanto para verificações de falhas em um ambiente de produção. Como muitos problemas podem não se manifestar em ambientes de testes bem controlados, a geração de *log* deve fazer parte do ambiente de produção, e não apenas nos ambientes especiais de testes, com o intuito de achar esses problemas e solucioná-los.

Quando se trata de aplicações distribuídas, como é o caso da ferramenta de análise implementada neste trabalho, realizar depurações se torna muito mais difícil. Então é muito importante a geração de *log* para que se possam encontrar eventuais erros e para que se consiga colecionar dados importantes para futuras análises.

O Log4J [8] é um *framework* da *The Apache Software Foundation*, desenvolvido para a realização de *logs* em aplicações. A versão atual, no momento da elaboração deste capítulo, se encontra na 1.2.13. Como outros *frameworks* de *logger*, o Log4j permite salvar com facilidade o resultado em arquivos ou outros meio de persistência, permite, também, definir a quantidade de detalhamento que se quer ter em cada registro do *log* e facilita a modificação dos formatos dos registros. E tudo isso se dá através de arquivos de configurações, que descartam a necessidade de se compilar novamente a aplicação. Existem quatro configurações fundamentais:

- **Category**, que permite classificar os registros de *log* segundo a estrutura lógica da aplicação;
- **Level**, que classificam os registros de *log* de acordo com a severidade do evento registrado. Os *levels* são DEBUG, INFO, WARN, ERROR e FATAL;
- **Appender**, que indicam onde os registros de *log* serão salvos;
- **Formatter**, que especificam como serão formatadas as informações armazenadas no registro de *log*.

O centro da API do Log4J é a classe **Logger**. Está classe é instanciada passando o nome de uma *category*. Este nome é utilizado para obter do arquivo de configuração os vínculos entre o *level*, *appender* e *formatter*, com a classe da aplicação que está utilizando o *log*, como exemplificado na figura 16.

```
public class JPlayer extends JFrame implements Runnable {

        // log da classe pelo log4j
private static Logger log = Logger.getLogger( JPlayer.class );
        ...
}
```

Figura 16: Instanciação do Logger

Com o atributo estático *log* declarado e inicializado conforme o modelo da figura 16. Então, sempre que necessário gerar uma mensagem de *log*, bastará chamar o método com o nome do *level* desejado, como mostrado no código 17. O método **error()** irá capturar a exceção gerada e enviar para o arquivo de *log*, referenciado pela variável **log,** que irá marcar a exceção com o *level* ERROR, indicado pelo método **error().**

```
try {
...
}
 // não existe nenhum player ou o formato não é suportado
catch ( NoPlayerException e ) {
     log.error( "Player não achado", e );
}

// erro na leitura do arquivo
catch ( IOException e ) {
   log.error( "Erro com o arquivo de mídia", e );
}
...
```

Figura 17: Exemplo de utilização do *log*

Mas onde os *logs* são gerados e armazenados? Para responder a esta pergunta, devemos entender como configurar o Log4J. As configurações podem ser feito por código, mas é pouco usado e desaconselhável, caso se queira mudar algo no *log*, teria que compilar novamente a aplicação. Também pode ser feito através de *Extensible Markup Language* (XML), mas o mais comum é através de um arquivo *properties*, chamado *log4j.properties*, mostrado na figura 18.

```
log4j.appender.LogFile=org.apache.log4j.FileAppender
log4j.appender.LogFile.File=/log/application.log
log4j.appender.LogFile.layout=HTMLLayout
log4j.logger.br.uff.projetofinal.transmissorrtp=INFO,LogFile
```

Figura 18: Arquivo de configuração

A primeira linha está definindo o *appender LogFile* do tipo *FileAppender* que usaremos para gerar o *log*. O tipo *FileAppender* gera o *log* em um arquivo, outros tipos podem ser usados para gerar *log* diretamente no banco ou enviar por email, por exemplo.

Já na segunda linha, defini-se *path* onde o arquivo de *log* será gerado, no caso apresentado será armazenado no arquivo *application.log*. A linha seguinte define o padrão que os registros de *log* terão, vários *layouts* pode ser definidos, neste caso usa-se o *HTMLLayout*.

A última linha irá definir para o Log4J, que a *category* "br.uff.projetofinal.transmissorrtp" irá utilizar o *appender* "LogFile" para o armazenamento dos registro, e que o *level* de que serão armazenados é o "INFO". Isto significa que, todos as classes abaixo de "br.uff.projetofinal.transmissorrtp" irão utilizar o "LogFile" para fazer o registro do *log*, e que somente os *logs* com *level* com "INFO" e superior serão registrados, os *levels* abaixo serão descartados.

Deste modo conseguiu-se gerar *logs* que ajudarão no desenvolvimento das aplicações, para realizar depuração, e na solução de possíveis problemas que possam ocorrer na aplicação, quando esta estiver em um ambiente de produção.

O Log4J foi de grande valia para o desenvolvimento da ferramenta de análise, pois permitiu que ao longo do processo fosse possível descobrir erros que sem ele seria impossível de se descobrir. Além de permitir que dados importantes para as análises fossem registrados e guardados.

CAPÍTULO 5 – IMPLEMENTAÇÃO

5.1 - Introdução

Neste capitulo é apresentado com maior detalhe as implementações da arquitetura Com Servidor Refletor e da arquitetura Sem o Servidor Refletor mostradas nas Seções 4.2 e 4.3 respectivamente. Cada uma das arquiteturas foi implementada a partir de uma disposição básica dos componentes, o que explica a semelhança na distribuição dos componentes das arquiteturas. Como as classes que implementam os protocolos de comunicação, de captura, transmissão e reprodução do vídeo e organização das estruturas.

As funcionalidades da aplicação de videoconferência, em ambas as arquiteturas são as mesmas, como mostrado na figura 19. Tanto na primeira arquitetura quanto na segunda, o usuário precisa se conectar com o servidor para que as listas de usuários conectados, tanto no servidor quanto nos outros usuários, possam ser atualizadas.

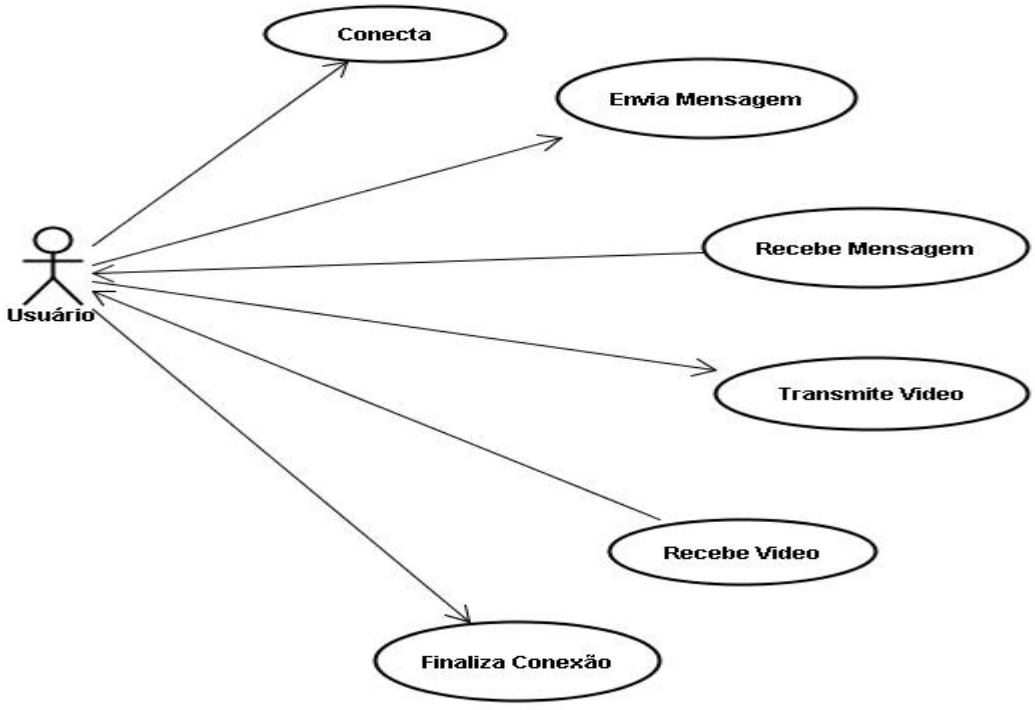

Figura 19: de Casos de uso da aplicação de videoconferência

Após uma conexão ter sido estabelecida, os usuários têm as opções de enviar mensagens para todos os outros usuários, de transmitirem o vídeo de sua webcam através da arquitetura ou ainda de se desconectarem. Dependendo de qual arquitetura estiver sendo usada, tanto o envio de mensagens quanto os vídeos terão comportamentos diferenciados no envio e no recebimento, dependendo de como estão dispostos os usuários e o servidor.

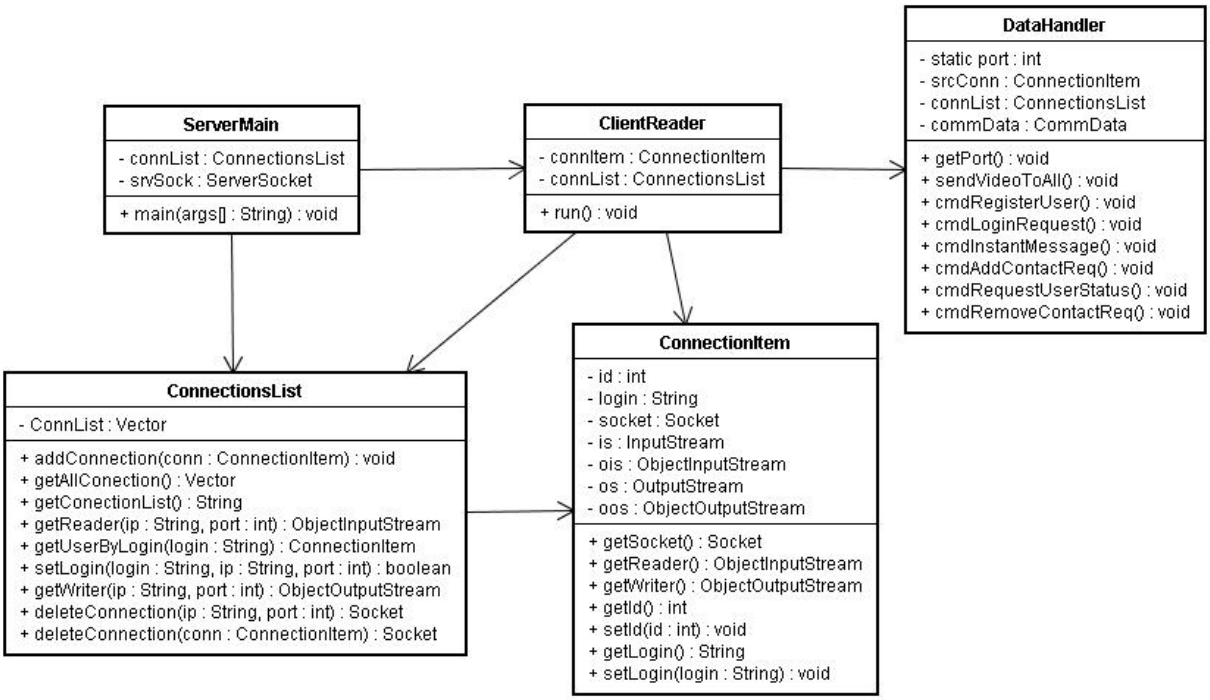

Figura 20: Diagrama de classes básicas do Servidor

A organização das classes básicas da aplicação é quase a mesma para ambas as arquiteturas implementadas. O servidor, em ambos os casos, possuirá conexões TCP com os transmissores/receptores para gerenciamento de portas e lista de usuários conectados, como se pode ver na figura 20 acima. As classes do cliente também serão as mesmas nas duas arquiteturas, a principal diferença existe na classe **DataHandler**, pois esta classe implementa o protocolo de comunicação usado nas arquiteturas, e possui métodos diferentes para cada uma. Há uma versão da **DataHandler** para cada arquitetura que será mostrada nas Seções correspondentes. O diagrama de classes do cliente é mostrado na figura 21.

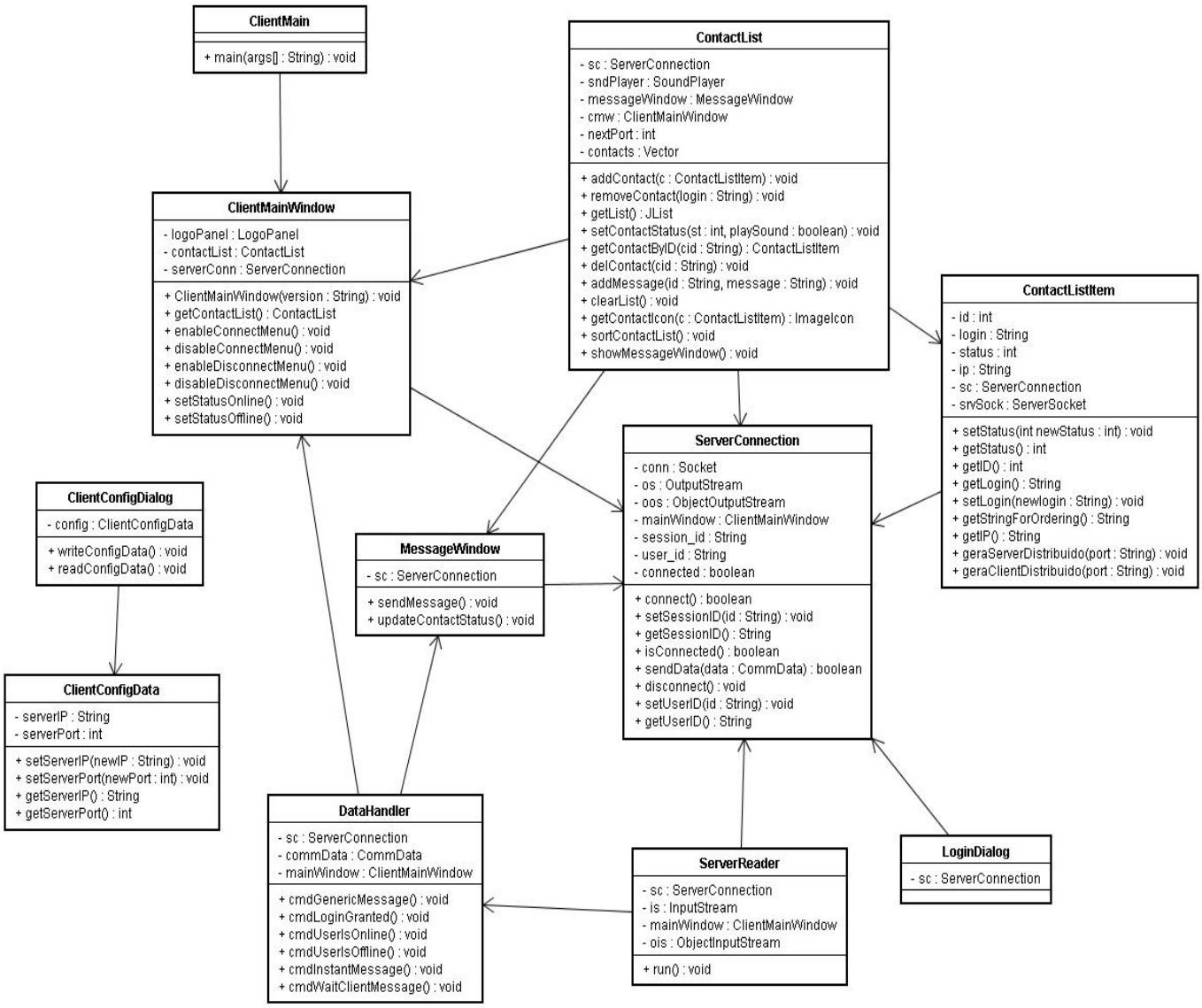

Figura 21: Diagrama de classes básicas do cliente

A classe **DataHandler** é responsável por identificar o tipo de mensagem recebida e executar ou delegar a execução de métodos que estejam no fluxo de execução do tipo da mensagem recebida. Já classe **ContactListItem**, tanto do servidor quanto a do cliente, possuem todas as informações necessárias dos usuários conectados ao servidor. Informações essas que serão necessárias para o envio do vídeo, como o IP do usuário. A classe **ContactList** possui a lista de **ContactListItem**, ou seja, a lista dos usuários que estão conectados. Pode-se visualizar o preenchimento desta lista através da interface de console da aplicação servidor, como mostrada nas figuras 22 e 23. A classe **ServerConnection** é a classe que possui os métodos para a comunicação entre servidor/cliente, ela é responsável pelo envio das mensagens de gerenciamento sobre as conexões TCP entre os usuários e o servidor. Para isso ela instancia uma *thread* da classe **ServerReader** que ser encarrega de realizar o envio da

mensagem. A **ClientMainWindow** é a classe que apresenta a interface principal do cliente, onde se tem o menu de configuração e navegação da ferramenta e a lista de usuários conectados. Esta é a classe central do Cliente, e a partir dela pode se navegar por todas as classes do sistema. A classe **MessageWindow** é utilizada para troca de mensagens de texto entre os clientes. É nesta classe que também disponibilizamos o evento de iniciação da videoconferência.

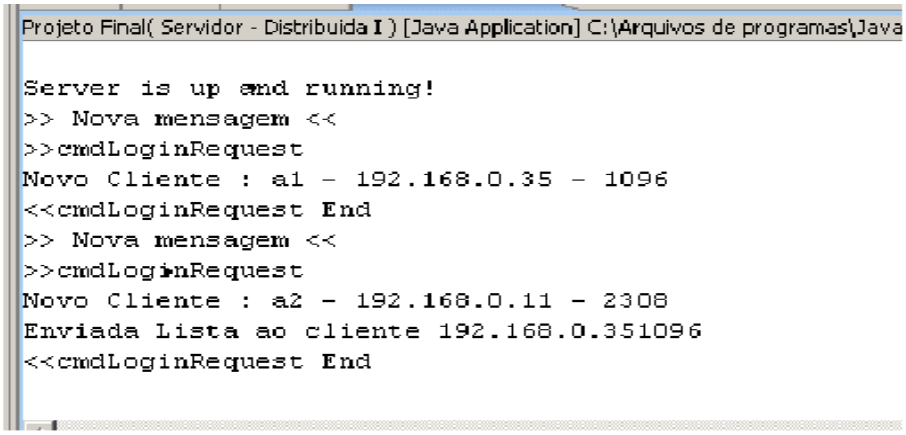

Figura 22 : Atualização da lista de clientes com os clientes a1 e a2

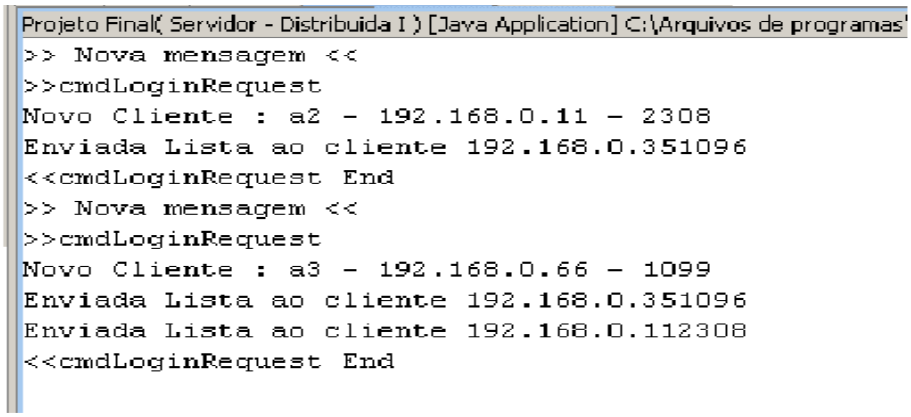

Figura 23: Atualização da lista de clientes com o cliente a3

Mas a classe mais importante é a **DataHandler** que implementa os protocolos de comunicação de cada arquitetura, que será mostrada nas próximas seções. Outro conjunto de classes importantes são as que realizam a captura, transmissão e reprodução do vídeo nos clientes. Nele temos a classe **JCapture** responsável por capturar da webcam o vídeo do usuário e armazená-lo em um **DataSource** para futuras reproduções. A classe **JTransmissor** é responsável por transmitir o vídeo contido no **DataSource** para outros usuários, precisando apenas saber o IP e porta para qual terá que transmitir. A última classe desse conjunto é a **JPlayer** que irá capturar o *stream* da rede, enviado pelo transmissor, e reproduzi-lo. Pode-se

ver a **JPlayer** em funcionamento, juntamente com a **ClientMainWindow**, na figura 24 abaixo.

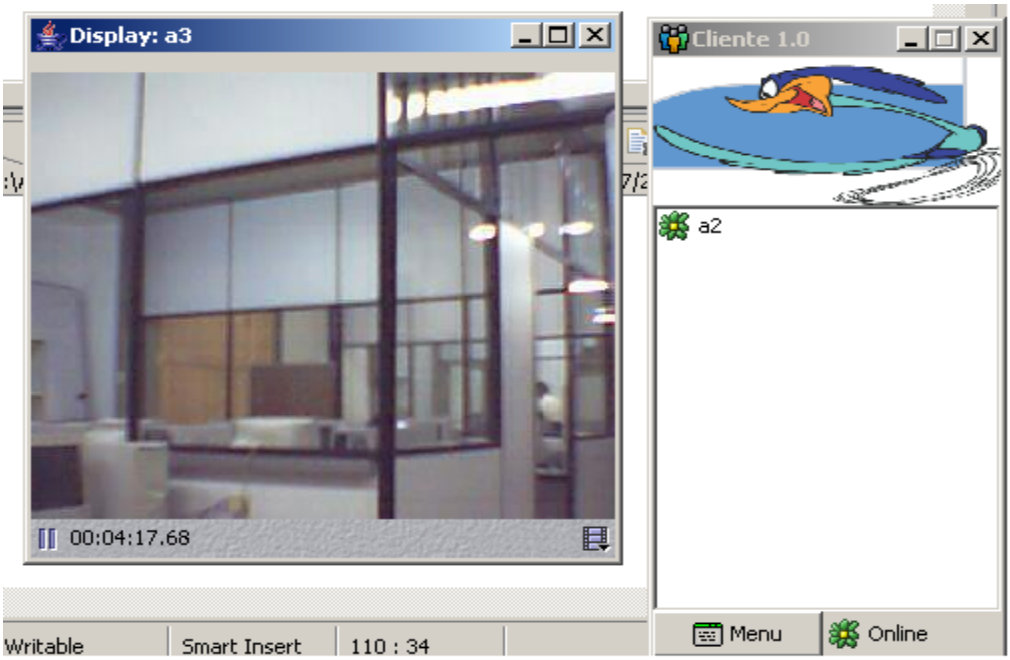

Figura 24: Classe JPlayer a esquerda e ClientMainWindow a direita

5.2 Servidor com Refletor

Na arquitetura com servidor com refletor, como descrito no Capítulo 3, se tem um servidor mais robusto que é o responsável não apenas pelo apresentação dos clientes, como também pela distribuição de toda a mídia para cada um dos clientes conectados.

Nesta arquitetura os clientes tem a sua implementação mais simples. Os clientes têm conhecimento apenas do seu servidor e somente com ele se comunicam. Não possuindo a lista de **ContactListItem** mencionada anteriormente. A classe **DataHandler** do cliente desta arquitetura irá tratar os seguintes eventos do protocolo estabelecido:

- CMD_THIS_PORT_MESSAGE: Evento que estabelece para o cliente qual deverá ser a porta utilizado por ele para o estabelecimento da comunicação de recepção, ou transmissão de vídeo do servidor ou para o servidor;
- CMD_SEND_VIDEO_MESSAGE: Evento que indica que o deve iniciar a transmissão do vídeo para o servidor;

- CMD_REVIC_VIDEO_MESSAGE: Evento que inicia a preparação do cliente para o recebimento do vídeo oriundo do servidor.

A classe **DataHandler** do servidor desta arquitetura trata os mesmos eventos, mas realizando diferentes funções, como mostrado a seguir:

- CMD_WHATS_PORT_MESSAGE: Evento que define a porta que o cliente usará para a transmissão ou recepção do vídeo;
- CMD_SEND_VIDEO_MESSAGE: Evento no qual o servidor, de posse da transmissão de vídeo do transmissor, inicia a transmissão do mesmo para os receptores e os avisa sobre está transmissão.

O protocolo pode ser visualizado no diagrama de seqüências da figura 25, onde pode se observar que antes do cliente iniciar uma transmissão do seu vídeo, ele antes deve negociar com o servidor qual porta deverá ser usada. Após o estabelecimento da porta, o cliente começa a enviar o vídeo para o servidor e o avisa. O servidor, ciente da mensagem enviada pelo cliente, começa a receber o vídeo e vai retransmitindo para cada um dos outros usuários. Para avisar estes usuários de que eles devem se preparar para reproduzir o vídeo, o servidor envia uma mensagem, que também irá informar qual porta deverá ser utilizada para a transmissão/recepção. Após receberem a mensagem, os receptores começam a reproduzir o vídeo enviado pelo servidor.

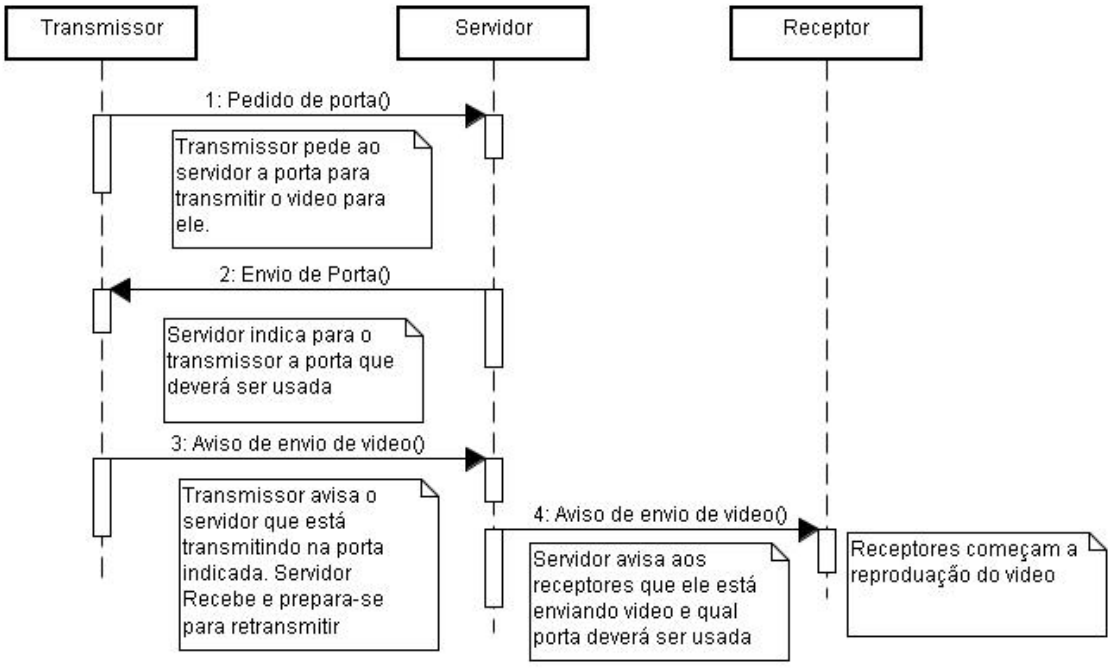

Figura 25: Diagrama de seqüência do protocolo da arquitetura com servidor refletor

Para esta arquitetura, existe apenas mais um item importante, é a classe **JTransmissor** do servidor, que captura o *stream* de vídeo enviado pelo usuário e o retransmite para os outros usuários.

5.3 Arquitetura sem Servidor Refletor

Na arquitetura distribuída os clientes estabelecem conexão diretamente com seus pares para a transmissão da mídia, e ao servidor cabe apenas a responsabilidade de apresentar os clientes uns aos outros.

Neste caso, os clientes tem a implementação completa que os clientes da implementação anterior. Onde em cada um deles deve ter um servidor para o caso deste ser o gerador da conexão de transmissão de mídia, pois o transmissor irá gerenciar as portas que serão usadas na transmissão da mídia.

Cada cliente, uma vez conectado, pode iniciar sua transmissão de mídia ao clicar na função de transmissão apresentada na tela de mensagens de texto. Ao iniciar a transmissão, todos os clientes conectados naquele instante receberam a transmissão.

A única classe relevante nesta arquitetura é a **DataHandler** do cliente, já que a **DataHandler** do servidor apenas irá redistribuir as mensagens de gerência para os clientes. A **DataHandler** do cliente deverá tratar os seguintes eventos:

- CMD_SEND_VIDEO: evento que indica que um cliente está enviado vídeo e que ele deverá se preparar para reproduzi-lo;
- CMD_NEW_PORT: evento que irá atualizar as portas para futuras transmissões de vídeo entre os clientes.

O protocolo implementado na **DataHandler** do cliente desta arquitetura é mais simples que o anterior, como mostrado na figura 26. O cliente transmissor começa a transmitir os seu vídeo para todos os outros clientes e envia uma mensagem para os mesmos de aviso. Os clientes receptores, ao receberem essa mensagem preparam e começam a reproduzir o vídeo enviado. Após cada início de transmissão de vídeo pelo cliente transmissor, o mesmo envia uma segunda mensagem de atualização das portas para futuras transmissões. O fato desta arquitetura possuir um evento CMD_NEW_PORT e a anterior não, se deve ao fato que na arquitetura onde o servidor é o refletor, e este realiza as conexões para transmissão de mídia com os receptores, conhecer quais as portas, onde as conexões estão sendo realizadas, dos receptores. Já nesta arquitetura, o servidor não possui esse conhecimento, por isso há a

necessidade de que os receptores e transmissores se informem quais portas estão disponíveis para a realização dessa transmissão de mídia.

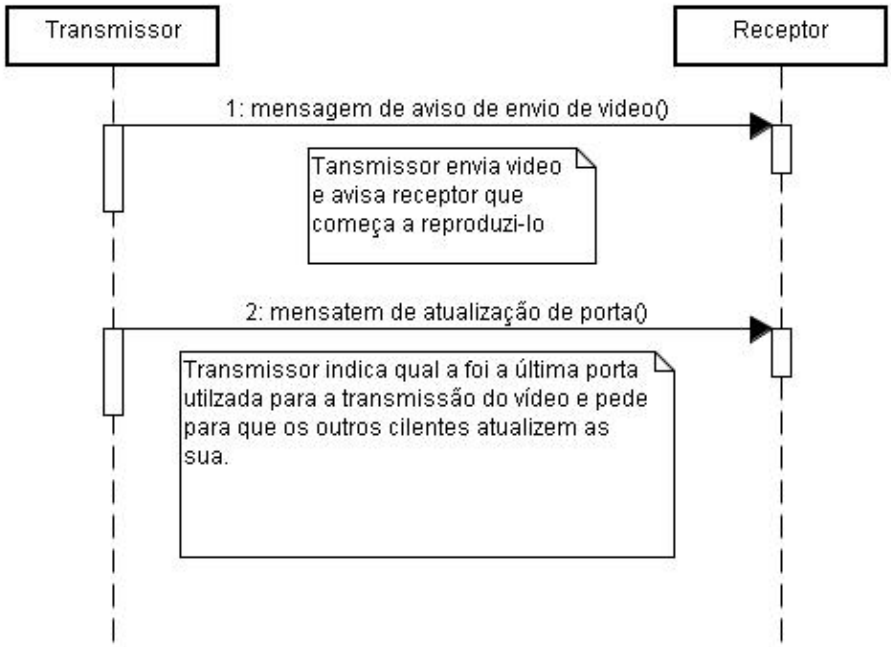

Figura 26: Diagrama de seqüência do protocolo da arquitetura sem servidor refletor

CAPÍTULO 6 - CONCLUSÃO

A transmissão de pacotes de mídia pela Internet está crescendo exponencialmente com a evolução das tecnologias de transmissão de dados, o que gera uma maior dificuldade de disponibilidade de banda para o tráfego de dados pela Internet. Mas a melhor qualidade no envio de dados não significa que podemos ser irresponsáveis com o envio de dados continuamente, como exige uma boa transmissão multimídia.

As aplicações multimídia têm características fortes quanto ao recebimento e envio de dados pela rede. Sabemos que estas aplicações têm alguma tolerância à perda de pacotes, mas exigem uma boa taxa de transmissão e recebimento de pacotes.

Tendo como meta a busca por um ambiente que viabilize a transmissão de mídia pela Internet, sem comprometer a qualidade de recepção de cada *host* conectado a sessão, surge a motivação para estudar arquiteturas que garantam a qualidade do serviço de transmissão multimídia. Por este motivo analisamos as três arquiteturas multiponto-multiponto apresentadas no Capítulo 3.

Atualmente não existe uma arquitetura padrão ou uma que seja considerada a mais poderosa para utilização em aplicações multimídia. Neste meio de estudo encontram-se diversas pesquisas em busca de uma arquitetura que, mesmo com as limitações dos meios de transmissão, consiga uma boa qualidade de recepção dos dados multimídia. A escolha da arquitetura correta dependerá sempre da aplicação. Para uma aplicação de videoconferência, como a que foi implementada neste trabalho, a arquitetura com receptores refletores, descrita na seção 3.4, se ajusta melhor a tentativa de minimizar o número de pacotes enviados sobre Internet e o caminho percorrido pelos pacotes.

Faz se necessária a realização de testes que verifiquem o atraso que os pacotes possuirão em cada arquitetura, a média de pacotes perdidos durante uma transmissão de vídeo e o gasto de processamento em cada *host* e no servidor quando estes estiverem transmitindo e recebendo. A arquitetura com receptores refletores deve ser implementada para a realização dos testes, para que se tenha parâmetros reais de comparação das potências de cada uma.

Junto ao crescimento das aplicações multimídia, tem-se o nascimento da ferramentas para facilitar o desenvolvimento de tais aplicações. Neste projeto foi introduzido o *framework* JMF (*Java Media Framework*) que se mostrou de grande valia na captura de mídia para posterior distribuição. Em conjunto com o JMF, utilizou-se como ferramenta, para coleta de

amostras do funcionamento de nossa implementação o *framework* Log4j, que é um poderoso gerador de *logs* e que será utilizado para a coleta dos dados para futura análise.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Carneiro, M. L. F., "Videoconferência – Ambiente para educação a distância", http://penta.ufrgs.br/pgie/workshop/mara.htm , julho de 2006.

[2] Cohen, B. "Incentives Build Robustness in BitTorrent" http://www.bittorrent.com/bittorrentecon.pdf , maio de 2003.

[3] Deitel, H. M. e Deitel, P. J.,"Java Como Programar", Volume 1, Quarta Edição, 2002.

[4] Internet Protocol (IP) Multicast: http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ipmulti.htm, julho de 2006.

[5] Java Media Framework (JMF). http://java.sun.com/products/java-media/jmf/, junho de 2006.

[6] Kurose, J. F., "Redes de Computadores e a Internet", Primeira Edição, São Paulo, p.381-382, 2003.

[7] Kurose, J. F., "Redes de Computadores e a Internet", Primeira Edição, São Paulo, 2003.

[8] Log4J, http://logging.apache.org/log4j/docs/index.htmlc, julho, 2006.

[9] Waitzman, D., Partridge , C. e Deering, S. "Distance Vector Multicast Routing Protocol", IETF RFC 1075, http://www.rfc-editor.org/rfc/rfc1075.txt, novembro de 1988.

[10] Socolofsky , T. e Kale ,C. "A TCP/IP Tutorial", IETF RFC 1180, http://www.rfc-editor.org/rfc/rfc1180.txt, janeiro de 1991.

[11] Schulzrinne , H., Casner, S., Frederick , R. e Jacobson ,V. "RTP: A Transport Protocol for Real-Time Application", IETF RFC 1889, http://www.rfc-editor.org/rfc/rfc1889.txt, janeiro de 1996.

[12] Fenner, R. "Internet Group Management Protocol, Version 2", IETF RFC 2236, http://www.rfc-editor.org/rfc/rfc2236.txt, novembro de 1997.

[13] Postel, J. "User Datagram Protocol", IETF RFC 768, http://www.rfc-editor.org/rfc/rfc768.txt, agosto de 1980.

[14] Postel, J. "Internet Protocol: DARPA Internet Program Protocol Specification", IETF RFC 791, http://www.rfc-editor.org/rfc/rfc791.txt, setembro de 1981.

[15] Sun Microsystems. http://java.sun.com/, junho de 2006.

[16] The Apache Foundation., http://www.apache.org/, julho de 2006.

[17] Vidal, M. T. V. L., “Especificação Formal, Verificação e Implementação de um protocolo de Comunicação Mulitponto-Multiponto e uma Aplicação de Texto-Conferência”. Tese de Mestrado, Programa de Engenharia Elétrica, COPPE/UFRJ, 1994.

[18] VRVS “Virtual Room Videoconferencing System” http://www.vrvs.org/?id=16|8 ,julho de 2006.

APÊNDICE

O código contido neste apêndice é apenas a parte referente aos transmissores e receptores de mídia, em cada arquitetura implementada. Todos os códigos estão sobre a licença GPL (http://www.magnux.org/doc/GPL-pt_BR.txt) e podem ser usados e modificados desde a mantenham.

Apêndice A
Arquitetura com Servidor Refletor

Código das classes de captura, transmissão e recepção dos clientes e a classe de transmissão do servidor.

Classes do Cliente

package br.uff.projetofinal.transmissorrtp;

```
import java.io.*;


import javax.media.*;
import javax.media.protocol.*;
import javax.media.control.*;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class JCapture{

        private static JCapture instance = null;
        private static final long serialVersionUID = 1L;

        //configuração do log4j
        private Logger log = Logger.getLogger(JCapture.class);
```

```
static{PropertyConfigurator.configure("log4j.properties");}

//controle dos sources
private int first = 1;

// formatos da mídia do dispositivo, formato escolhido pelo usuário
private Format selectedFormat, formats[] = {new Format("0. javax.media.format.RGBFormat RGB,
352x288, Length=304128, 24-bit, Masks=3:2:1, PixelStride=3, LineStride=1056, Flipped")} ;

// controles dos formatos de mídia do dispositivo
private FormatControl formatControls[];

// informações de especificação do dispositivo
private CaptureDeviceInfo deviceInfo;


// fontes de dados de entrada e saída
private DataSource inSource;

//método para instanciar a classe
public static synchronized JCapture getInstance(){
        if( null == instance )
                instance = new JCapture();

        return instance;
}

//construtor
private JCapture(){
        deviceInfo = new CaptureDeviceInfo("vfw:Microsoft WDM Image Capture (Win32):0",
                                                                        new
MediaLocator("vfw://0"),
                                                                        formats);

        // compatível com componentes GUI peso leve
        Manager.setHint( Manager.LIGHTWEIGHT_RENDERER,Boolean.TRUE );

        // se o dispositivo de captura anterior estiver aberto, desconecta-o
        if ( inSource != null )
                inSource.disconnect();
```

```
                // obtém dispositivo e configura seu formato
                try {
                        // cria fonte de dados a partir do MediaLocator do dispositivo
                        inSource = Manager.createCloneableDataSource( Manager.createDataSource(
deviceInfo.getLocator() ) );
                        inSource = Manager.createCloneableDataSource( inSource );

                        // obtém controles de configuração de formato para o dispositivo
                        formatControls = ( ( CaptureDevice ) inSource ).getFormatControls();

                        // obtém a configuração de formato do dispositivo desejada pelo usuário
                        selectedFormat = formats[0];

                        if ( selectedFormat == null )
                                return;

                        setDeviceFormat( selectedFormat );
                }
                catch ( NoDataSourceException e ) {
                        // não consegue encontrar DataSource a partir do MediaLocator
                        log.error( e.getMessage() +" - error 1");
                }
                catch ( IOException e ) {
                        //erro de conexão ao dispositivo
                        log.error( e.getMessage() +" - error 2");
                }
        } // fim do construtor

        public DataSource getDataSource(){
                DataSource result = null;
                if( 1 == first ){
                        first = 0;
                        result = inSource;
                }
                else{
                        result = ((SourceCloneable)(inSource)).createClone();
                }

                return result;
```

```
        }

        // configura o formato de saída da mídia capturada pelo dispositivo
        public void setDeviceFormat( Format currentFormat ){

                // configura formato desejado em todos os controles de formato
                for ( int i = 0; i < formatControls.length; i++ ) {

                        // assegura que o controle de formato pode ser configurado
                        if ( formatControls[ i ].isEnabled() ) {
                                formatControls[ i ].setFormat( currentFormat );

                                log.info( "Formato de saída " +     formatControls[ i ].getFormat() );
                        }

                } // fim do laço for
        }

        // exibe mensagens de erro
        public void showErrorMessage( String error ){
                log.error(error);
        }

} // fim da classe JCapture
```

package br.uff.projetofinal.transmissorrtp;

```
import java.io.*;
import java.net.*;

// Pacotes de extensão de Java
import javax.media.*;
import javax.media.protocol.*;
import javax.media.control.*;
import javax.media.rtp.*;
import javax.media.format.*;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
```

```
public class JTransmissor implements Runnable{
        //          configuração do log4j
        private Logger log = Logger.getLogger(JTransmissor.class);
        static{PropertyConfigurator.configure("log4j.properties");}

        // endereço IP, arquivo ou nome do MediaLocator, número da porta
        private String ipAddress;
        private int port;

        // processador que controla o fluxo de dados
        private Processor processor;

        //dados de entrada do processador
        private DataSource inSource;
        // dados de saída do processador a serem enviados
        private DataSource outSource;

        // controles configuráveis das trilhas de mídia
        private TrackControl tracks[];

        // gerenciador de sessão de RTP
        private RTPManager rtpManager[];

        // gerencia do trafico por RTCP
        private GlobalTransmissionStats globalTS;

        // captura do vídeo e do áudio
        private JCapture cap;

        // construtor para JTransmissor
        public JTransmissor( String ip, int portNumber ){
                port = portNumber;
                ipAddress = ip;
                cap = JCapture.getInstance();
        }

        // inicializa e configura o processador
        // devolve true se bem-sucedido, false caso contrário
        public void run(){
                // cria processador a partir de MediaLocator
```

```
            try {
                        inSource = Manager.createCloneableDataSource( cap.getDataSource() );
                        processor = Manager.createProcessor( inSource );
                        // registra um ControllerListener para o processador
                        // para esperar eventos de transição de estado
                        processor.addControllerListener( new ProcessorEventHandler() );

                        log.debug( "Processor configuring..." );

                        // configura o processador antes de ajustá-lo
                        processor.configure();
             }
             // erro de conexão com a fonte
             catch ( IOException e ) {
                        log.error(e.getMessage());
             }
             // exceção disparada quando nenhum processador
             // pode ser encontrado para fonte de dados específica
            catch ( NoProcessorException e ) {
                       log.error(e.getMessage());
             }

} // fim do método beginSession

// tratador ControllerListener para o processador
private class ProcessorEventHandler extends ControllerAdapter {

             // configura formato de saída e realiza
             // o processador configurado
             public void configureComplete(ConfigureCompleteEvent configureCompleteEvent ){
                        log.debug( "\nProcessor configured." );
                        setOutputFormat();

                        log.debug( "\nRealizing Processor...\n" );
                        processor.realize();
             }

             // começa a enviar quando o processador está realizado
             public void realizeComplete( RealizeCompleteEvent realizeCompleteEvent ){
                        if ( transmitMedia() == true )
```

```
                    log.debug( "\nTransmission setup OK" );
                else
                    log.debug( "\nTransmission failed." );
        }

        // faz parar a sessão de RTP quando não há mídia a enviar
        public void endOfMedia( EndOfMediaEvent mediaEndEvent ){
            stopTransmission();
            log.debug( "Transmission completed." );
        }
} // fim da classe interna ProcessorEventHandler

// configura o formato de saída de todas as trilhas na mídia
public void setOutputFormat(){
        // configura o tipo de conteúdo da saída para formato suportado por RTP
        processor.setContentDescriptor( new ContentDescriptor( ContentDescriptor.RAW_RTP ) );

        // obtém todos os controles de trilha do processador
        tracks = processor.getTrackControls();

        // formatos de uma trilha suportados por RTP
        Format rtpFormats[];

        // configura cada trilha para o primeiro formato suportado
        // por RTP encontrado naquela trilha
        for ( int i = 0; i < tracks.length; i++ ) {

            log.debug( "\nTrack #" + ( i + 1 ) + " supports " );

            if ( tracks[ i ].isEnabled() ) {
                rtpFormats = tracks[ i ].getSupportedFormats();

                // se existirem formatos da trilha suportados, exibe
                // todos os formatos suportados por RTP e configura
                // o formato de trilha para o primeiro formato suportado
                if ( rtpFormats.length > 0 ) {
                    for ( int j = 0; j < rtpFormats.length; j++ )
                        log.debug( rtpFormats[ j ] );

                    tracks[ i ].setFormat( rtpFormats[ 0 ] );
```

```
                                log.debug( "Track format set to " + tracks[ i ].getFormat() );
                        }
                        else
                                log.error(          "No supported RTP formats for track!" );

                } // fim do if

        } // fim do laço for

} // fim do método setOutputFormat

// envia mídia com valor booleano indicando sucesso
public boolean transmitMedia(){
        outSource = processor.getDataOutput();

        if ( outSource == null ) {
                log.debug( "No data source from media!" );
                return false;
        }

        // gerenciadores de stream de RTP para cada trilha
        rtpManager = new RTPManager[ tracks.length ];

        // endereços de sessão de RTP de destino e local
        SessionAddress localAddress, remoteAddress;
        // stream RTP que está sendo enviado
        SendStream sendStream;

        // endereço IP
        InetAddress ip;

        // inicializa endereços de transmissão e envia a mídia para a saída
        try {
                // transmite todas as trilhas na mídia
                for ( int i = 0; i < tracks.length; i++ ) {
                        // instancia um RTPManager
                        rtpManager[ i ] = RTPManager.newInstance();
                        // adiciona 2 para especificar o número de porta do próximo controle;
                        // (o RTP Session Manager usa 2 portas)
```

```
                    port += ( 2 * i );

                    // obtém endereço IP do host a partir do string ipAddress
                    ip = InetAddress.getByName( ipAddress );

                    // encapsula par de endereços IP para controle e
                    // dados com duas portas dentro do endereço de sessão local
                    localAddress = new SessionAddress( ip.getLocalHost(), port );

                    // obtém o endereço de sessão remoteAddress
                    remoteAddress = new SessionAddress( ip, port );

                    // inicializa a sessão
                    rtpManager[ i ].initialize( localAddress );

                    // abre a sessão de RTP para o destino
                    rtpManager[ i ].addTarget( remoteAddress );

                    log.debug( "\nStarted RTP session: " + ipAddress + " " + port);

                    // cria stream de envio na sessão de RTP
                    sendStream = rtpManager[ i ].createSendStream( outSource, i );

                    // criar global Recieve e Transmite do RTCP
                    globalTS = rtpManager[i].getGlobalTransmissionStats();

                    // começa a enviar o stream
                    sendStream.start();
                    log.debug( "Transmitting Track #" + ( i + 1 ) + " ... " );

            } // fim do laço for

            // começa carga da mídia
            processor.start();


} // fim de try
// endereço local desconhecido ou endereço remoto não pode ser resolvido
catch ( InvalidSessionAddressException e ) {
            log.error(e.getMessage());
```

```
                return false;
            }
            // erro de conexão com a DataSource
            catch ( IOException e ) {
                log.error(e.getMessage());
                return false;
            }
            // formato não configurado ou formato inválido configurado na fonte de stream
            catch ( UnsupportedFormatException e ) {
                log.error(e.getMessage());
                return false;
            }
            // transmissão inicializada com sucesso
            return true;

        } // fim do método transmitMedia

        // faz parar a transmissão e fecha recursos
        public void stopTransmission(){
            if ( processor != null ) {

                // faz parar o processador
                processor.stop();

                // descarta o processador
                processor.close();

                if ( rtpManager != null ){

                    //verificar estatisticas da transmissão;
                    log.info("Número de pacotes RTP enviados: " + String.valueOf(
globalTS.getRTCPSent() ) );
                    log.info("Número de pacotes RTP que falharam :" + String.valueOf(
globalTS.getTransmitFailed() ) );


                    // fecha alvos de destino
                    // e descarta gerenciadores de RTP
                    for ( int i = 0; i < rtpManager.length; i++ ) {
```

```
                                        // fecha streams para todos os destinos
                                        // com um motivo para terminar
                                        rtpManager[ i ].removeTargets( "Session stopped." );

                                        // libera os recursos da sessão de RTP
                                        rtpManager[ i ].dispose();
                                    }
                                }

                        } // fim de if

                        log.debug( "Transmission stopped." );

                } // fim do método stopTransmission

} // fim da classe JTransmissor
```

package br.uff.projetofinal.transmissorrtp;

```
import java.awt.*;
import java.io.*;

// Pacotes de extensão de Java
import javax.swing.*;
import javax.media.*;
import javax.media.rtp.GlobalReceptionStats;
import javax.media.rtp.RTPManager;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class JPlayer extends JFrame implements Runnable {

        private Logger log = Logger.getLogger(JPlayer.class);
        static{PropertyConfigurator.configure("log4j.properties");}
        private static final long serialVersionUID = 1L;

        // reprodutor de mídia em Java
        private Player player;
```

```
// componente para conteúdo visual
private Component visualMedia;

// componentes de controle para a mídia
private Component mediaControl;

// contêiner principal
private Container container;

// endereços do arquivo de mídia e da mídia
private String address;

//gerenciador de sessão de RTP
private RTPManager rtpManager;

// gerencia do trafico por RTCP
private GlobalReceptionStats globalRS;

// construtor para JPlayer
public JPlayer( String ip , int port){
        super( "Java Media Player" );
        log.info("JPlayer aberto");
        container = getContentPane();

        // painel que contém botões
        JPanel buttonPanel = new JPanel();
        container.add( buttonPanel, BorderLayout.NORTH );

        this.address = "rtp://"+ip+":"+port+"/video";

        // liga a geração leve nos players para permitir
        // melhor compatibilidade com componentes GUI de peso leve
        Manager.setHint( Manager.LIGHTWEIGHT_RENDERER,Boolean.TRUE );

} // fim do construtor JPlayer

// método utilitário para mensagens de erro "pop-up"
public void showErrorMessage( String error ){
        log.error( error );
}
```

```
// obtém arquivo do computador
public File getFile(){
        JFileChooser fileChooser = new JFileChooser();

        fileChooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
        int result = fileChooser.showOpenDialog( this );

        if ( result == JFileChooser.CANCEL_OPTION )
                return null;
        else
                return fileChooser.getSelectedFile();
}

// obtém endereço da mídia digitado pelo usuário
public String getMediaLocation(){
        String input = JOptionPane.showInputDialog( this, "Enter URL" );

        // se o usuário pressionar OK sem digitar dados
        if ( input != null && input.length() == 0 )
                return null;

        return input;
}

// cria player com o endereço da mídia
public void run( ){
        // restaura o player e a janela se houver player anterior
        if ( player != null )
                removePlayerComponents();

        // endereço da origem da mídia
        MediaLocator mediaLocator = new MediaLocator( address );

        if ( mediaLocator == null ) {
                showErrorMessage( "Error opening file" );
                return;
        }

        // cria um player a partir de MediaLocator
```

```
        try {
                player = Manager.createPlayer( mediaLocator );
                // registra ControllerListener para tratar de eventos do Player
                player.addControllerListener( new PlayerEventHandler() );

                this.setSize( 300, 300 );
                this.setLocation( 300, 300 );
                this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
                this.setVisible( true );

                // chama realize para permitir a geração da mídia do player
                player.realize();
        }

        // não existe nenhum player ou o formato não é suportado
        catch ( NoPlayerException e ) {
                log.error( e.getMessage() );
        }

        // erro na leitura do arquivo
        catch ( IOException e ) {
                log.error( e.getMessage() );
        }

} // fim do método makePlayer

// devolve o player para os recursos do sistema
// e restaura a mídia e os controles
public void removePlayerComponents(){
        // remove componente de vídeo anterior, se existe um
        if ( visualMedia != null )
                container.remove( visualMedia );

        // remove controle de mídia anterior, se existe um
        if ( mediaControl != null )
                container.remove( mediaControl );

        // faz parar o player e devolve os recursos alocados
        player.close();
}
```

```
// obtém controles visuais para mídia e player
public void getMediaComponents(){
        // obtém componente visual do player
        visualMedia = player.getVisualComponent();

        // adiciona componente visual, se estiver presente
        if ( visualMedia != null )
                container.add( visualMedia, BorderLayout.CENTER );
        // obtém a GUI de controle do player
        mediaControl = player.getControlPanelComponent();

        // adiciona componente de controles, se estiver presente
        if ( mediaControl != null )
                container.add( mediaControl, BorderLayout.SOUTH );

} // fim do método getMediaComponents

// tratador para os eventos ControllerEvents do player
private class PlayerEventHandler extends ControllerAdapter {

        // carrega antecipadamente a mídia assim que o player é realizado
        public void realizeComplete( RealizeCompleteEvent realizeDoneEvent ){
                player.prefetch();
        }

        // player pode começar a mostrar a mídia após a carga antecipada
        public void prefetchComplete( PrefetchCompleteEvent prefetchDoneEvent ){
                getMediaComponents();

                // assegura um leiaute válido para a frame
                validate();

                // começa a reproduzir a mídia
                player.start();

        } // fim do método prefetchComplete

        // se fim da mídia, restaura para o início e pára de reproduzir
        public void endOfMedia( EndOfMediaEvent mediaEndEvent ){
```

```
                    player.setMediaTime( new Time( 0 ) );
                    player.stop();
                }
        } // fim da classe interna PlayerEventHandler

} // fim da classe JPlayer
```

Classes do Servidor

package br.uff.projetofinal.transmissorrtp;

```
import java.io.IOException;

import javax.media.Manager;
import javax.media.MediaLocator;
import javax.media.NoDataSourceException;
import javax.media.protocol.DataSource;
import javax.media.protocol.SourceCloneable;

import org.apache.log4j.Logger;

public class JCaptureTransmission {

        private static JCaptureTransmission instance = null;
        private Logger log = Logger.getLogger( JCaptureTransmission.class );
        private DataSource inSource;
        private int first = 1;

//          endereço IP, arquivo ou nome do MediaLocator, número da porta
          private String ipTo;
          private String ipFrom;
          private int portFrom;
          private int portTo;

          private static boolean firstInicialization = true;

        private JCaptureTransmission(){
        }

        public static synchronized JCaptureTransmission getInstance(){
                if( null == instance )
                        instance = new JCaptureTransmission();
```

```
            return instance;
        }

        public synchronized void inicialization( String ipFromHost, int portNumberFromHost, String ipToHost,
int portNumberToHost ){
            if( firstInicialization ){
                portFrom = portNumberFromHost;
                ipTo = ipToHost;
                ipFrom = ipFromHost;
                portTo = portNumberToHost;
                configure();
                firstInicialization = false;
            }
        }

        private synchronized void configure(){
            log.debug(">> Criando Processor para a transmissão com ip: " + ipTo );
            log.debug(">> Recuperando méida de: " + ipFrom );
            MediaLocator mediaLocator = new MediaLocator( "rtp://" + ipFrom + ":" + portFrom
+"/video" );
            log.debug(">> MediaLocator criado");
            log.debug("Protocolo do media locator: "+ mediaLocator.getProtocol() );


            try{
                log.debug("crinado o data source iSource");
                inSource = Manager.createDataSource( mediaLocator );
                inSource = Manager.createCloneableDataSource( inSource );
                log.debug("data source criado");
            }
            catch( NoDataSourceException e ){
                log.error( e.getMessage() );
            }
            catch( IOException e ){
                log.error( e.getMessage() );
            }
        }

        public synchronized DataSource getDataSource(){
```

```
            DataSource result = null;
            if( 1 == first ){
                    first = 0;
                    result = inSource;
            }
            else{
                    result = ((SourceCloneable)(inSource)).createClone();
            }

            return result;
 }

public String getIpFrom() {
            return ipFrom;
}

public void setIpFrom(String ipFrom) {
            this.ipFrom = ipFrom;
}

public String getIpTo() {
            return ipTo;
}

public void setIpTo(String ipTo) {
            this.ipTo = ipTo;
}

public int getPortFrom() {
            return portFrom;
}

public void setPortFrom(int portFrom) {
            this.portFrom = portFrom;
}

public int getPortTo() {
            return portTo;
}
```

```
	public void setPortTo(int portTo) {
		this.portTo = portTo;
	}

}// fim da classe JCaptureTransmission
```

package br.uff.projetofinal.transmissorrtp;

```
//classes base java
import java.io.IOException;
import java.net.InetAddress;

//classes do jmf
import javax.media.ConfigureCompleteEvent;
import javax.media.ControllerAdapter;
import javax.media.EndOfMediaEvent;
import javax.media.Format;
import javax.media.Manager;
import javax.media.MediaLocator;
import javax.media.NoDataSourceException;
import javax.media.NoProcessorException;
import javax.media.Processor;
import javax.media.RealizeCompleteEvent;
import javax.media.control.TrackControl;
import javax.media.format.UnsupportedFormatException;
import javax.media.protocol.ContentDescriptor;
import javax.media.protocol.DataSource;
import javax.media.rtp.InvalidSessionAddressException;
import javax.media.rtp.RTPManager;
import javax.media.rtp.SendStream;
import javax.media.rtp.SessionAddress;

//classes do log4j
import org.apache.log4j.Logger;


 public class JTransmissor implements Runnable{
	//	configuração do log4j
	private Logger log = Logger.getLogger(JTransmissor.class);

	private int portTo;
```

```
        private String ipTo;

        // processador que controla o fluxo de dados
        private Processor processor;

        //dados de entrada
        private DataSource inSource;

        // dados de saída do processador a serem enviados
        private DataSource outSource;

        // controles configuráveis das trilhas de mídia
        private TrackControl tracks[];

        // gerenciador de sessão de RTP
        private RTPManager rtpManager[];

        // recupernado transmissão
        private JCaptureTransmission ct;

        //verifica se é primeira vez
        private static boolean first = true;

        // construtor para JTransmissor
        public JTransmissor( String ipFromHost, int portNumberFromHost, String ipToHost, int
portNumberToHost ){

                ct = JCaptureTransmission.getInstance();

                ct.inicialization(ipFromHost, portNumberFromHost, ipToHost, portNumberToHost );
                first = false;

                ipTo = ct.getIpTo();
                portTo = ct.getPortTo();

        }


        public void run(){
                // cria processador a partir de MediaLocator
```

```
                try {
//                      log.debug(">> Criando Processor para a transmissão com ip: " + ipTo );
//                      log.debug(">> Recuperando méida de: " + ipFrom );
//                      MediaLocator mediaLocator = new MediaLocator( "rtp://" + ipFrom + ":" + portFrom
+"/video" );
//                      log.debug(">> MediaLocator criado");
//                      log.debug("Protocolo do media locator: "+ mediaLocator.getProtocol() );
                        //log.debug("URL do media locator: "+ mediaLocator.getURL() );
//
//                      try{
//                              log.debug("crinado o data source iSource");
//                              inSource = Manager.createDataSource( mediaLocator );
//                              log.debug("data source criado");
//                      }
//                      catch( NoDataSourceException e ){
//                              log.error( e.getMessage() );
//                      }
                        log.debug(">>Criando processor");
                        processor = Manager.createProcessor( ct.getDataSource() );
//                      processor = Manager.createProcessor( mediaLocator );
                        log.debug(">> Processor criado");
                        // registra um ControllerListener para o processador
                        // para esperar eventos de transição de estado
                        log.debug(">> Adicionado o Listener do processor");
                        processor.addControllerListener( new ProcessorEventHandler() );

                        log.debug( "Processor configuring..." );

                        // configura o processador antes de ajustá-lo
                        processor.configure();
                }
                // erro de conexão com a fonte
                catch ( IOException e ) {
                        log.error(e.getMessage());
                }
                // exceção disparada quando nenhum processador
                // pode ser encontrado para fonte de dados específica
                catch ( NoProcessorException e ) {
                        log.error(e.getMessage());
                }
```

```
} // fim do método beginSession

// tratador ControllerListener para o processador
private class ProcessorEventHandler extends ControllerAdapter {

        // configura formato de saída e realiza
        // o processador configurado
        public void configureComplete(ConfigureCompleteEvent configureCompleteEvent ){
                log.debug( "\nProcessor configured." );
                setOutputFormat();

                log.debug( "\nRealizing Processor...\n" );
                processor.realize();
        }

        // começa a enviar quando o processador está realizado
        public void realizeComplete( RealizeCompleteEvent realizeCompleteEvent ){
                if ( transmitMedia() )
                        log.debug( "\nTransmission setup OK" );
                else
                        log.debug( "\nTransmission failed." );
        }

        // faz parar a sessão de RTP quando não há mídia a enviar
        public void endOfMedia( EndOfMediaEvent mediaEndEvent ){
                stopTransmission();
                log.debug( "Transmission completed." );
        }
} // fim da classe interna ProcessorEventHandler

// configura o formato de saída de todas as trilhas na mídia
public void setOutputFormat(){
        // configura o tipo de conteúdo da saída para formato suportado por RTP
        processor.setContentDescriptor( new ContentDescriptor( ContentDescriptor.RAW_RTP ) );

        // obtém todos os controles de trilha do processador
        tracks = processor.getTrackControls();

        // formatos de uma trilha suportados por RTP
        Format rtpFormats[];
```

```
            // configura cada trilha para o primeiro formato suportado
            // por RTP encontrado naquela trilha
            for ( int i = 0; i < tracks.length; i++ ) {

                        log.debug( "\nTrack #" + ( i + 1 ) + " supports " );

                        if ( tracks[ i ].isEnabled() ) {
                                    rtpFormats = tracks[ i ].getSupportedFormats();

                                    // se existirem formatos da trilha suportados, exibe
                                    // todos os formatos suportados por RTP e configura
                                    // o formato de trilha para o primeiro formato suportado
                                    if ( rtpFormats.length > 0 ) {
                                                for ( int j = 0; j < rtpFormats.length; j++ )
                                                            log.debug( rtpFormats[ j ] );

                                                tracks[ i ].setFormat( rtpFormats[ 0 ] );

                                                log.debug( "Track format set to " + tracks[ i ].getFormat() );
                                    }
                                    else
                                                log.error(          "No supported RTP formats for track!" );

                        } // fim do if

            } // fim do laço for

} // fim do método setOutputFormat

 // envia mídia com valor booleano indicando sucesso
 public boolean transmitMedia(){
            outSource = (DataSource) processor.getDataOutput();

            if ( outSource == null ) {
                        log.debug( "No data source from media!" );
                       return false;
            }

            // gerenciadores de stream de RTP para cada trilha
```

```
rtpManager = new RTPManager[ tracks.length ];

// endereços de sessão de RTP de destino e local
SessionAddress localAddress, remoteAddress;
// stream RTP que está sendo enviado
SendStream sendStream;

// endereço IP
InetAddress ip;

// inicializa endereços de transmissão e envia a mídia para a saída
try {
        // transmite todas as trilhas na mídia
        for ( int i = 0; i < tracks.length; i++ ) {
                // instancia um RTPManager
                rtpManager[ i ] = RTPManager.newInstance();
                // adiciona 2 para especificar o número de porta do próximo controle;
                // (o RTP Session Manager usa 2 portas)
                portTo += ( 2 * i );

                // obtém endereço IP do host a partir do string ipAddress
                ip = InetAddress.getByName( ipTo );

                // encapsula par de endereços IP para controle e
                // dados com duas portas dentro do endereço de sessão local
                log.debug( "Criando SeesionAddress local" );
                log.debug( "ip.getLocalHost= " + ip.getLocalHost());
                log.debug( "portTo= " + portTo );
                localAddress = new SessionAddress( ip.getLocalHost(), portTo );
                log.debug( "Sesseion Address local criado" );

                // obtém o endereço de sessão remoteAddress
                log.debug( "Criando SeesionAddress remoto" );
                remoteAddress = new SessionAddress( ip, portTo );
                log.debug( "Sesseion Address remoto criado" );

                // inicializa a sessão
                log.debug( "inicializando sessão" );
                log.debug( "localAddress host=" + localAddress.getDataHostAddress());
                log.debug( "localAddress port=" + localAddress.getDataPort());
```

```
                    rtpManager[ i ].initialize( localAddress );
                    log.debug( "sessão inicializado" );

                    // abre a sessão de RTP para o destino
                    log.debug( "abrindo sessão rtp para destino" );
                    rtpManager[ i ].addTarget( remoteAddress );
                    log.debug( "sessão rtp apra destino aberta" );

                    log.debug( "\nStarted RTP session: " + ipTo + " " + portTo);

                    // cria stream de envio na sessão de RTP
                    log.debug( "criando Stream com outSource" );
                    sendStream = rtpManager[ i ].createSendStream( outSource, i );

                    // começa a enviar o stream
                    log.debug( "enviando stream" );
                    sendStream.start();

                    log.debug( "Transmitting Track #" + ( i + 1 ) + " ... " );

            } // fim do laço for

            // começa carga da mídia
            processor.start();

} // fim de try
// endereço local desconhecido ou endereço remoto não pode ser resolvido
catch ( InvalidSessionAddressException e ) {
            log.error(e.getMessage());
            return false;
}
// erro de conexão com a DataSource
catch ( IOException e ) {
            log.error(e.getMessage());
            return false;
}
// formato não configurado ou formato inválido configurado na fonte de stream
catch ( UnsupportedFormatException e ) {
            log.error(e.getMessage());
            return false;
```

```
                }
                // transmissão inicializada com sucesso
                return true;

        } // fim do método transmitMedia

        // faz parar a transmissão e fecha recursos
        public void stopTransmission(){
                if ( processor != null ) {

                        // faz parar o processador
                        processor.stop();

                        // descarta o processador
                        processor.close();

                        if ( rtpManager != null ){

                                // fecha alvos de destino
                                // e descarta gerenciadores de RTP
                                for ( int i = 0; i < rtpManager.length; i++ ) {

                                        // fecha streams para todos os destinos
                                        // com um motivo para terminar
                                        rtpManager[ i ].removeTargets( "Session stopped." );

                                        // libera os recursos da sessão de RTP
                                        rtpManager[ i ].dispose();
                                }
                        }

                } // fim de if

                log.debug( "Transmission stopped." );

        } // fim do método stopTransmission

} // fim da classe JTransmissor
```

Apêndice B
Arquitetura sem Servidor Refletor

Código das classes de captura, transmissão e recepção dos clientes, contudo, o servidor não possuir nenhuma classe desse tipo nesta arquitetura.

Classes do Cliente

```
package br.uff.projetofinal.transmissorrtp;

 import java.io.*;


 import javax.media.*;
 import javax.media.protocol.*;
 import javax.media.control.*;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

 public class JCapture{

        private static JCapture instance = null;
        private static final long serialVersionUID = 1L;

        //configuração do log4j
        private Logger log = Logger.getLogger(JCapture.class);
        static{PropertyConfigurator.configure("log4j.properties");}

         //controle dos sources
         private int first = 1;

         // formatos da mídia do dispositivo, formato escolhido pelo usuário
         private Format selectedFormat, formats[] = {new Format("0. javax.media.format.RGBFormat RGB,
352x288, Length=304128, 24-bit, Masks=3:2:1, PixelStride=3, LineStride=1056, Flipped")} ;

         // controles dos formatos de mídia do dispositivo
         private FormatControl formatControls[];
```

```
        // informações de especificação do dispositivo
        private CaptureDeviceInfo deviceInfo;


        // fontes de dados de entrada e saída
        private DataSource inSource;

        //método para instanciar a classe
        public static synchronized JCapture getInstance(){
                if( null == instance )
                        instance = new JCapture();

                return instance;
        }

        //construtor
        private JCapture(){
                deviceInfo = new CaptureDeviceInfo("vfw:Microsoft WDM Image Capture (Win32):0",
                                                                                new
MediaLocator("vfw://0"),
                                                                                formats);

                // compatível com componentes GUI peso leve
                Manager.setHint( Manager.LIGHTWEIGHT_RENDERER,Boolean.TRUE );

                // se o dispositivo de captura anterior estiver aberto, desconecta-o
                if ( inSource != null )
                        inSource.disconnect();

                // obtém dispositivo e configura seu formato
                try {
                        // cria fonte de dados a partir do MediaLocator do dispositivo
                        inSource = Manager.createCloneableDataSource( Manager.createDataSource(
deviceInfo.getLocator() ) );
                        inSource = Manager.createCloneableDataSource( inSource );

                        // obtém controles de configuração de formato para o dispositivo
                        formatControls = ( ( CaptureDevice ) inSource ).getFormatControls();

                        // obtém a configuração de formato do dispositivo desejada pelo usuário
```

```
                selectedFormat = formats[0];

                if ( selectedFormat == null )
                        return;

                setDeviceFormat( selectedFormat );
        }
        catch ( NoDataSourceException e ) {
                // não consegue encontrar DataSource a partir do MediaLocator
                log.error( e.getMessage() +" - error 1");
        }
        catch ( IOException e ) {
                //erro de conexão ao dispositivo
                log.error( e.getMessage() +" - error 2");
        }
} // fim do construtor

public DataSource getDataSource(){
        DataSource result = null;
        if( 1 == first ){
                first = 0;
                result = inSource;
        }
        else{
                result = ((SourceCloneable)(inSource)).createClone();
        }

        return result;
}

// configura o formato de saída da mídia capturada pelo dispositivo
public void setDeviceFormat( Format currentFormat ){

        // configura formato desejado em todos os controles de formato
        for ( int i = 0; i < formatControls.length; i++ ) {

                // assegura que o controle de formato pode ser configurado
                if ( formatControls[ i ].isEnabled() ) {
                        formatControls[ i ].setFormat( currentFormat );
```

```
                                log.info( "Formato de saída " +     formatControls[ i ].getFormat() );
                        }

                } // fim do laço for
        }

        // exibe mensagens de erro
        public void showErrorMessage( String error ){
                log.error(error);
        }

} // fim da classe JCapture

package br.uff.projetofinal.transmissorrtp;

import java.io.*;
import java.net.*;

// Pacotes de extensão de Java
import javax.media.*;
import javax.media.protocol.*;
import javax.media.control.*;
import javax.media.rtp.*;
import javax.media.format.*;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class JTransmissor implements Runnable{
        //          configuração do log4j
        private Logger log = Logger.getLogger(JTransmissor.class);
        static{PropertyConfigurator.configure("log4j.properties");}

        // endereço IP, arquivo ou nome do MediaLocator, número da porta
        private String ipAddress;
        private int port;

        // processador que controla o fluxo de dados
        private Processor processor;
```

```
//dados de entrada do processador
private DataSource inSource;
// dados de saída do processador a serem enviados
private DataSource outSource;

// controles configuráveis das trilhas de mídia
private TrackControl tracks[];

// gerenciador de sessão de RTP
private RTPManager rtpManager[];

private JCapture cap;

// construtor para JTransmissor
public JTransmissor( String ip, int portNumber ){
        port = portNumber;
        ipAddress = ip;
        cap = JCapture.getInstance();
}

// inicializa e configura o processador
// devolve true se bem-sucedido, false caso contrário
public void run(){
        // cria processador a partir de MediaLocator
        try {
                inSource = Manager.createCloneableDataSource( cap.getDataSource() );
                processor = Manager.createProcessor( inSource );
                // registra um ControllerListener para o processador
                // para esperar eventos de transição de estado
                processor.addControllerListener( new ProcessorEventHandler() );

                log.debug( "Processor configuring..." );

                // configura o processador antes de ajustá-lo
                processor.configure();
        }
        // erro de conexão com a fonte
        catch ( IOException e ) {
                log.error(e.getMessage());
        }
```

```
            // exceção disparada quando nenhum processador
            // pode ser encontrado para fonte de dados específica
            catch ( NoProcessorException e ) {
                        log.error(e.getMessage());
            }

} // fim do método beginSession

// tratador ControllerListener para o processador
private class ProcessorEventHandler extends ControllerAdapter {

            // configura formato de saída e realiza
            // o processador configurado
            public void configureComplete(ConfigureCompleteEvent configureCompleteEvent ){
                        log.debug( "\nProcessor configured." );
                        setOutputFormat();

                        log.debug( "\nRealizing Processor...\n" );
                        processor.realize();
            }

            // começa a enviar quando o processador está realizado
            public void realizeComplete( RealizeCompleteEvent realizeCompleteEvent ){
                        if ( transmitMedia() == true )
                                    log.debug( "\nTransmission setup OK" );
                        else
                                    log.debug( "\nTransmission failed." );
            }

            // faz parar a sessão de RTP quando não há mídia a enviar
            public void endOfMedia( EndOfMediaEvent mediaEndEvent ){
                        stopTransmission();
                        log.debug( "Transmission completed." );
            }
} // fim da classe interna ProcessorEventHandler

// configura o formato de saída de todas as trilhas na mídia
public void setOutputFormat(){
            // configura o tipo de conteúdo da saída para formato suportado por RTP
            processor.setContentDescriptor( new ContentDescriptor( ContentDescriptor.RAW_RTP ) );
```

```
            // obtém todos os controles de trilha do processador
            tracks = processor.getTrackControls();

            // formatos de uma trilha suportados por RTP
            Format rtpFormats[];

            // configura cada trilha para o primeiro formato suportado
            // por RTP encontrado naquela trilha
            for ( int i = 0; i < tracks.length; i++ ) {

                  log.debug( "\nTrack #" + ( i + 1 ) + " supports " );

                  if ( tracks[ i ].isEnabled() ) {
                        rtpFormats = tracks[ i ].getSupportedFormats();

                        // se existirem formatos da trilha suportados, exibe
                        // todos os formatos suportados por RTP e configura
                        // o formato de trilha para o primeiro formato suportado
                        if ( rtpFormats.length > 0 ) {
                              for ( int j = 0; j < rtpFormats.length; j++ )
                                    log.debug( rtpFormats[ j ] );

                              tracks[ i ].setFormat( rtpFormats[ 0 ] );

                              log.debug( "Track format set to " + tracks[ i ].getFormat() );
                        }
                        else
                              log.error(          "No supported RTP formats for track!" );

                  } // fim do if

            } // fim do laço for

      } // fim do método setOutputFormat

      // envia mídia com valor booleano indicando sucesso
      public boolean transmitMedia(){
            outSource = processor.getDataOutput();
```

```
if ( outSource == null ) {
            log.debug( "No data source from media!" );
           return false;
}

// gerenciadores de stream de RTP para cada trilha
rtpManager = new RTPManager[ tracks.length ];

// endereços de sessão de RTP de destino e local
SessionAddress localAddress, remoteAddress;
// stream RTP que está sendo enviado
SendStream sendStream;

// endereço IP
InetAddress ip;

// inicializa endereços de transmissão e envia a mídia para a saída
try {
            // transmite todas as trilhas na mídia
            for ( int i = 0; i < tracks.length; i++ ) {
                        // instancia um RTPManager
                        rtpManager[ i ] = RTPManager.newInstance();
                        // adiciona 2 para especificar o número de porta do próximo controle;
                       // (o RTP Session Manager usa 2 portas)
                        port += ( 2 * i );

                        // obtém endereço IP do host a partir do string ipAddress
                        ip = InetAddress.getByName( ipAddress );

                        // encapsula par de endereços IP para controle e
                        // dados com duas portas dentro do endereço de sessão local
                        localAddress = new SessionAddress( ip.getLocalHost(), port );

                        // obtém o endereço de sessão remoteAddress
                        remoteAddress = new SessionAddress( ip, port );

                        // inicializa a sessão
                        rtpManager[ i ].initialize( localAddress );

                        // abre a sessão de RTP para o destino
```

```
                        rtpManager[ i ].addTarget( remoteAddress );

                        log.debug( "\nStarted RTP session: " + ipAddress + " " + port);

                        // cria stream de envio na sessão de RTP
                        sendStream = rtpManager[ i ].createSendStream( outSource, i );

                        // começa a enviar o stream
                        sendStream.start();

                        log.debug( "Transmitting Track #" + ( i + 1 ) + " ... " );

                } // fim do laço for

                // começa carga da mídia
                processor.start();

        } // fim de try
        // endereço local desconhecido ou endereço remoto não pode ser resolvido
        catch ( InvalidSessionAddressException e ) {
                log.error(e.getMessage());
                return false;
        }
        // erro de conexão com a DataSource
        catch ( IOException e ) {
                log.error(e.getMessage());
                return false;
        }
        // formato não configurado ou formato inválido configurado na fonte de stream
        catch ( UnsupportedFormatException e ) {
                log.error(e.getMessage());
                return false;
       }
        // transmissão inicializada com sucesso
        return true;

} // fim do método transmitMedia

// faz parar a transmissão e fecha recursos
public void stopTransmission(){
```

```
                if ( processor != null ) {

                        // faz parar o processador
                        processor.stop();

                        // descarta o processador
                        processor.close();

                        if ( rtpManager != null ){

                                // fecha alvos de destino
                                // e descarta gerenciadores de RTP
                                for ( int i = 0; i < rtpManager.length; i++ ) {

                                        // fecha streams para todos os destinos
                                        // com um motivo para terminar
                                        rtpManager[ i ].removeTargets( "Session stopped." );

                                        // libera os recursos da sessão de RTP
                                        rtpManager[ i ].dispose();
                                }
                        }

                } // fim de if

                log.debug( "Transmission stopped." );

        } // fim do método stopTransmission

} // fim da classe JTransmissor

package br.uff.projetofinal.transmissorrtp;


import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

// Pacotes de extensão de Java
```

```
 import javax.swing.*;
import javax.media.*;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

 public class JPlayer extends JFrame implements Runnable {

            private Logger log = Logger.getLogger(JPlayer.class);
            static{PropertyConfigurator.configure("log4j.properties");}
            private static final long serialVersionUID = 1L;

            // reprodutor de mídia em Java
             private Player player;

             // componente para conteúdo visual
             private Component visualMedia;

             // componentes de controle para a mídia
             private Component mediaControl;

             // contêiner principal
             private Container container;

             // endereços do arquivo de mídia e da mídia
             private File mediaFile;
             private URL fileURL;
             private String address;

             // construtor para JPlayer
             public JPlayer( String login, String ip , int port){
                        super( "Display: " + login);

                        this.setSize( 300, 300 );
                        this.setLocation( 300, 300 );
                        this.setDefaultCloseOperation( EXIT_ON_CLOSE );
                        this.setVisible( true );

                        log.info("JPlayer aberto");
                        container = getContentPane();
```

```
        // painel que contém botões
        JPanel buttonPanel = new JPanel();
        container.add( buttonPanel, BorderLayout.NORTH );

        this.address = "rtp://"+ip+":"+port+"/video";
        log.info("Enviado vídeo para: " + this.address );

        // liga a geração leve nos players para permitir
        // melhor compatibilidade com componentes GUI de peso leve
        Manager.setHint( Manager.LIGHTWEIGHT_RENDERER,Boolean.TRUE );

} // fim do construtor JPlayer

// método utilitário para mensagens de erro "pop-up"
public void showErrorMessage( String error ){
        log.error( error );
}

// obtém arquivo do computador
public File getFile(){
        JFileChooser fileChooser = new JFileChooser();

        fileChooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
        int result = fileChooser.showOpenDialog( this );

        if ( result == JFileChooser.CANCEL_OPTION )
                return null;
        else
                return fileChooser.getSelectedFile();
}

// obtém endereço da mídia digitado pelo usuário
public String getMediaLocation(){
        String input = JOptionPane.showInputDialog( this, "Enter URL" );

        // se o usuário pressionar OK sem digitar dados
        if ( input != null && input.length() == 0 )
                return null;
```

```
        return input;
}

// cria player com o endereço da mídia
public void run( ){
        // restaura o player e a janela se houver player anterior
        if ( player != null )
                removePlayerComponents();

        // endereço da origem da mídia
        MediaLocator mediaLocator = new MediaLocator( address );

        if ( mediaLocator == null ) {
                showErrorMessage( "Error opening file" );
                return;
        }

        // cria um player a partir de MediaLocator
        try {
                player = Manager.createPlayer( mediaLocator );
                // registra ControllerListener para tratar de eventos do Player
                player.addControllerListener(
                new PlayerEventHandler() );

                this.setVisible(true);

                // chama realize para permitir a geração da mídia do player
                player.realize();

        }

        // não existe nenhum player ou o formato não é suportado
        catch ( NoPlayerException e ) {
                log.error( e.getMessage() );
        }

        // erro na leitura do arquivo
        catch ( IOException e ) {
                log.error( e.getMessage() );
        }
```

```
} // fim do método makePlayer

// devolve o player para os recursos do sistema
// e restaura a mídia e os controles
public void removePlayerComponents(){
        // remove componente de vídeo anterior, se existe um
        if ( visualMedia != null )
                container.remove( visualMedia );

        // remove controle de mídia anterior, se existe um
        if ( mediaControl != null )
                container.remove( mediaControl );

        // faz parar o player e devolve os recursos alocados
        player.close();
}

// obtém controles visuais para mídia e player
public void getMediaComponents(){
        // obtém componente visual do player
        visualMedia = player.getVisualComponent();

        // adiciona componente visual, se estiver presente
        if ( visualMedia != null )
                container.add( visualMedia, BorderLayout.CENTER );
        // obtém a GUI de controle do player
       mediaControl = player.getControlPanelComponent();

        // adiciona componente de controles, se estiver presente
        if ( mediaControl != null )
                container.add( mediaControl, BorderLayout.SOUTH );

} // fim do método getMediaComponents

// tratador para os eventos ControllerEvents do player
private class PlayerEventHandler extends ControllerAdapter {

        // carrega antecipadamente a mídia assim que o player é realizado
        public void realizeComplete( RealizeCompleteEvent realizeDoneEvent ){
```

```
                player.prefetch();
            }

            // player pode começar a mostrar a mídia após a carga antecipada
            public void prefetchComplete( PrefetchCompleteEvent prefetchDoneEvent ){
                getMediaComponents();

                // assegura um leiaute válido para a frame
                validate();

                // começa a reproduzir a mídia
                player.start();

            } // fim do método prefetchComplete

            // se fim da mídia, restaura para o início e pára de reproduzir
            public void endOfMedia( EndOfMediaEvent mediaEndEvent ){
                player.setMediaTime( new Time( 0 ) );
                player.stop();
            }
        } // fim da classe interna PlayerEventHandler

} // fim da classe JPlayer
```