Universidade Federal Fluminense

Um Novo Mecanismo de Compartilhamento de Recursos para Transmissão de Vídeo com Alta Interatividade e Experimentos

Tese submetida para a obtenção do título de

Mestre em Computação

ao Curso de Pós-Graduação em Computação da UFF

por

Melba Lima Gorza

Dezembro 2003

Um Novo Mecanismo de Compartilhamento de Recursos para Transmissão de Vídeo com Alta Interatividade e Experimentos

Melba Lima Gorza

DISSERTAÇÃO APRESENTADA AO CURSO DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO DA UNIVERSIDADE FEDERAL FLUMINENSE COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM COMPUTAÇÃO.

Aprovada por:

Profa. Anna Dolejsi dos Santos, Dra.

Prof. Edmundo Albuquerque de Souza e Silva, Ph.D.

Profa. Rosa Maria Meri Leão, Dra.

Profa. Morganna Carmen Diniz, Dra.

NITERÓI, RJ - BRASIL

DEZEMBRO DE 2003

Dedico este trabalho a meus pais
e meus irmãos com muito amor

Agradecimentos

Agradeço a Deus pela oportunidade de ter alcançado mais um objetivo de minha vida; à minha família que, mesmo distante, soube estar presente me encorajando nos momentos difíceis; ao amor e à atenção de meus pais, Edelmo e Neuza, pois sem eles tudo seria muito mais difícil e aos meus irmãos, Eneida e Ronan, por serem tão importantes em minha vida, por saberem me escutar e acreditar em meus passos.

Gostaria de agradecer à professora Anna por ser não só minha orientadora, mas uma amiga; pela sua contribuição em meu crescimento acadêmico e pela crença em meu trabalho.

Agradeço ao professor Edmundo que me aceitou no LAND, possibilitando um enorme crescimento na área da computação - um verdadeiro orientador, apaixonado pelo que faz, e que nos motiva a cada dia.

Gostaria de agradecer também ao pessoal do LAND, em especial a Carolina e Isabela pela amizade, ao Flávio pela paciência, confiança e ajuda em muitos passos deste trabalho, ao Kleber pelas revisões em meu texto, dicas e esclarecimentos de detalhes importantes; e às minhas amigas, Ilza e Cristiane, pela força e por tantos momentos alegres nesta “terra” tão distante.

Meus agradecimentos à Capes por ter financiado minha bolsa de estudos.

Por fim, gostaria de agradecer ao meu namorado Ezequiel pela compreensão nos momentos de ausência e pelo apoio para finalizar este trabalho e aos seus pais, Ivone e Pedro, por terem me acolhido como uma verdadeira filha.

Resumo da Tese apresentada à UFF como parte dos requisitos necessários para a obtenção do grau de Mestre em Computação (M.Sc.)

Um Novo Mecanismo de Compartilhamento de Recursos para Transmissão de Vídeo com Alta Interatividade e Experimentos

Melba Lima Gorza

Dezembro/2003

Orientadores: Anna Dolejsi dos Santos
Edmundo de Souza e Silva

Programa: Engenharia da Computação

O desenvolvimento de novas tecnologias viabilizou o uso das aplicações multimídia de tempo real. Porém, estas aplicações tem severas restrições de tempo e necessitam de largura de banda suficiente para garantir a qualidade de serviço desejável. Para solucionar estes problemas existem algumas técnicas que podem ser incorporadas aos servidores. Neste trabalho são apresentados os aspectos relacionados a servidores multimídia, a arquitetura do servidor RIO e as técnicas de compartilhamento de recursos que foram estudadas com objetivo de melhorar a qualidade de serviço oferecida aos usuários da aplicação. Para minimizar o uso da banda do servidor para transmissão da mídia, a técnica de *patching* foi incorporada à arquitetura do servidor RIO. Assim, os cliente compartilham a banda do servidor através de transmissões *multicast*. As operações de interatividade foram tratadas durante a implementação da técnica. Foi desenvolvido um cliente para o RIO que executa em plataforma Windows, com objetivo de expandir a utilização do RIO em outro sistema operacional. Além disso, foi desenvolvido um emulador que visa a simulação de cargas reais no servidor RIO. Foi criado um modelo que simula a técnica implementada no servidor. Os resultados obtidos do emulador e do modelo foram analisados e comparados. Como este trabalho tem como foco o ensino a distância, mais especificamente o projeto CEDERJ (Centro de Educação a Distância

do Estado do Rio de Janeiro), o servidor RIO será utilizado para armazenamento e distribuição dos vídeos gravados dos seus cursos.

Abstract of Thesis presented to UFF as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A New Mechanism of Resources Sharing to Video Transmitions
With High Interactive And Experiments

Melba Lima Gorza

December/2003

Advisors: Anna Dolejsi dos Santos
Edmundo de Souza e Silva
Department: Computer Science Engineering

The new technology development made possible the use of real time multimedia applications. Nevertheless this applications have strict time demands and need enough bandwidth to guarantee the desirable QoS. To solve this problems there are some techniques that can be applied to servers. This work introduce the characteristics of multimedia servers, the RIO server architecture and the resources sharing techniques that were studied to improve the offered QoS to the users application. To minimize the server bandwidth required to transmit the media, the patching technique was incorporated to RIO server architecture. Thus, the clients share the server bandwidth through multicast transmissions. The interactive operations were handled during the implementation of this technique. A Windows client application was developed to RIO to spread this platform in another operation system. Moreover, a emulator was developed to perform real loads in RIO server. A model was created to simulate the interactive patching technique in RIO. The results gathered from the emulator and the model were analyzed and compared. As the main focus of this work is distance learning, more specifically the CEDERJ project, the RIO server will be used to store and distribute the record movies during the courses.

Palavras-chave

1. Servidor Multimídia
2. Técnica de Patching
3. Interatividade
4. Compartilhamento de Banda
5. Simulação

Glossário

CBR	:	Taxa Constante de Bits (*Constant Bit Rate*);
VBR	:	Taxa Variável de Bits (*Variable Bit Rate*);
QoS	:	Qualidade de Serviço (*Quality of Service*);
RIO	:	Operações de Entrada e Saída Aleatórias (*Randomized I/O*);
PI	:	Patching Interativo;
RT	:	Tempo Real (*Real Time*);
NRT	:	Não é de Tempo Real (*Non Real Time*);
FIFO	:	O Primeiro a Chegar é o Primeiro a Sair (*First In First Out*);

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

COM o desenvolvimento da Internet, as aplicações multimídia passaram a fazer parte do dia-a-dia dos usuários. As aplicações de voz, videoconferência e televisão digital ganharam um grande destaque. Porém, estas novas funcionalidades requerem uma infra-estrutura capaz de suportar suas exigências, como processamento rápido e largura de banda suficiente para a distribuição desses tipos de dados.

Para permitir o acesso eficiente de um número significativo de usuários a um servidor, técnicas de compartilhamento de banda, implementadas na camada de aplicação, tem sido propostas na literatura.

Este trabalho focaliza o problema de acesso de múltiplos usuários a um servidor multimídia, em particular para aplicações de ensino a distância.

1.1 Motivação

As aplicações multimídia distribuídas mudaram o cenário das aplicações tradicionais, incorporando novas funcionalidades como visualização de textos e imagens em tempo real, ambientes virtuais e interação simultânea de vários usuários localizados em diferentes ambientes. Porém, estas aplicações apresentam desafios aos projetos de seus sistemas. Para garantir a qualidade desejável dos serviços oferecidos, elas possuem requisitos estreitos de tempo, necessitam de largura de banda suficiente para transmissão dos dados e grande espaço de armazenamento nos servidores.

Como este trabalho engloba um projeto de ensino a distância, a alta interatividade dos usuários também deve ser tratada para garantir a qualidade de apresentação dos dados a todos os clientes que utilizam o servidor. Estes fatores evidenciam a necessidade do estudo e desenvolvimento de técnicas de compartilhamento de recursos, em particular recursos de banda do canal de acesso, de forma a permitir o aumento do número de usuários acessando o sistema sem a degradação dos serviços oferecidos.

1.2 Objetivos e Contribuições

Este trabalho faz uso do servidor multimídia RIO [34], focalizando em aplicações de ensino a distância. Estas aplicações possuem comportamento diferenciado de uma aplicação de distribuição de vídeo, por exemplo, devido a alta interatividade dos usuários. Por este motivo, técnicas de compartilhamento de banda existentes na literatura (*Patching, Hierarchical Stream Merging, Bandwidth Skimmimg* [25, 19, 17], entre outras) podem ter sua eficiência comprometida.

Os objetivos desse trabalho são:

- Desenvolver um cliente para o servidor RIO, que execute em plataforma Windows. Assim, o servidor RIO pode ser utilizado em ambiente Linux e Windows;

- Estudar as técnicas de compartilhamento de banda para distribuição de vídeo, e a eficiência destas técnicas em aplicações onde um usuário acessa aulas de um curso previamento gravado, e não um vídeo para entretenimento, isto é, aplicações de ensino a distância;

- Investigar um nova técnica de compartilhamento de banda que seja simples e mais eficiente que as técnicas existentes para alicações de ensino a distância;

- Aumentar a eficiência do servidor RIO para atender um maior número de clientes, através do projeto e implementação desta nova técnica elaborada, retratando a realidade das aplicações de ensino a distância.

Devido à complexidade do trabalho, incluindo o entendimento do funcionamento do servidor RIO, e desenvolvimento e implementação da nova técnica de compartilhamento de banda, várias tarefas deste trabalho foram realizadas em conjunto com outro trabalho, que possui o mesmo objetivo, e encontra-se em andamento no laboratório LAND [7].

As tarefas de estudo do servidor, desenvolvimento de uma nova técnica, projeto das novas funcionalidades do servidor RIO, incluindo a implementação da nova técnica, foram realizadas em conjunto [31]. Entretanto, neste trabalho foi implementado um ambiente de testes e medições reais na rede do laboratório LAND para avaliar as novas funcionalidades. No trabalho [31] foi desenvolvido um modelo detalhado de simulação, de forma que fosse possível comparar o resultado das medições com os de modelagem e análise do mecanismo proposto e implementado.

Foi feito um estudo aprofundado do servidor RIO (desenvolvido em projeto de cooperação entre a *University of California Los Angeles* (UCLA), Universidade Federal do Rio de Janeiro (UFRJ) e Universidade Federal de Minas Gerais (UFMG) para entender sua arquitetura, a comunicação com os clientes, os serviços oferecidos e foram visualizadas as melhorias que podem contribuir no aumento da QoS oferecida pelo servidor. O servidor RIO será utilizado em um projeto de ensino à distância do CEDERJ (Centro de Ensino a distância do Estado do Rio de Janeiro), do qual fazem parte as universidade UFF (Universidade Federal Fluminense) e UFRJ.

Foram implementados novos mecanismos para a comunicação cliente-servidor, para o gerenciamento de clientes e suas requisições e para controle dos eventos gerados pelas operações de interatividade. Para isso foi desenvolvido um módulo, a nível de aplicação, denominado PI (*Patching*) Interativo. Ele faz o tratamento dos blocos de dados solicitados pelos clientes antes que os pedidos sejam enviados ao servidor, de forma a prover o compartilhamento da banda pelos clientes de maneira eficiente. O módulo foi projetado de forma a manter a estrutura do servidor RIO. Desta maneira, os clientes que acessam um vídeo podem escolher entre uma conexão direta com o servidor ou através do módulo PI.

A análise da requisição do cliente é feita pelo módulo PI, para permitir a busca antecipada e o armazenamento dos dados a serem transmitidos. A cada novo cliente, é feita uma verificação na lista de clientes já admitidos, obtendo-se as informações necessárias para a transmissão da mídia a este novo cliente. Além da implementação da técnica de compartilhamento, foi desenvolvido um emulador que visa a simulação de cargas reais de clientes de vídeo acessando o servidor RIO. O desempenho do sistema pode ser analisado através do resultados obtidos do emulador e comparado com um modelo do sistema desenvolvido em [31].

1.3 Organização do texto

No Capítulo 2 é feito um breve estudo sobre servidores multimídia e descrita a arquitetura do servidor RIO, a técnica utilizada para a organização de dados e alguns aperfeiçoamentos que foram implementados em [14]. O Capítulo 3 apresenta algumas técnicas existentes de compartilhamento de banda e descreve os problemas que não são tratados por elas. O Capítulo 4 descreve a implementação do cliente Windows e apresenta os clientes existentes. Os Capítulos 5 descreve a implementação do módulo PI. No Capítulo 6 são apresentados os testes realizados e os resultados obtidos. Além disso, é apresentado o emulador criado para simular as cargas reais para o servidor RIO e testar a eficácia do módulo implementado. No Capítulo 7 são apresentadas as conclusões deste trabalho e sugestões para trabalhos futuros.

Capítulo 2

Servidor Multimídia RIO

ESTE capítulo fará uma introdução geral sobre servidores multimídia. A arquitetura do servidor RIO será apresentada, incluindo seus componentes e a interação entre eles. Será explicada também a técnica de armazenamento de dados utilizada pelo servidor RIO e alguns aperfeiçoamentos implementados em [14].

2.1 Servidores Multimídia

Servidores multimídia são os componentes responsáveis pelo armazenamento e distribuição do conteúdo multimídia (vídeo, música, texto, fotografia, etc.) e ainda o gerenciamento das requisições dos clientes. Eles podem conter um ou mais processadores e, em geral, vários discos. Os objetos armazenados pelos servidores são, usualmente, divididos em blocos e distribuídos nos discos que compõe o sistema. Esta distribuição é feita de acordo com a política de organização de dados utilizada [21]. Neste capítulo, utilizamos o servidor RIO para exemplificar o funcionamento de um servidor.

Quando o servidor recebe uma requisição de bloco, ele verifica em qual disco o bloco está armazenado para poder recuperá-lo. A Figura 2.1 ilustra o fluxo dos dados no servidor multimídia RIO. Após a recuperação do bloco no disco, ele é armazenado em um *buffer* para posterior envio ao cliente. Os blocos recebidos pelo cliente antes do momento de sua exibição são armazenados em um *playout buffer*, usado para evitar que haja interrupção por falta de blocos, devido aos retardos aleatórios na recuperação e transmissão de informação.

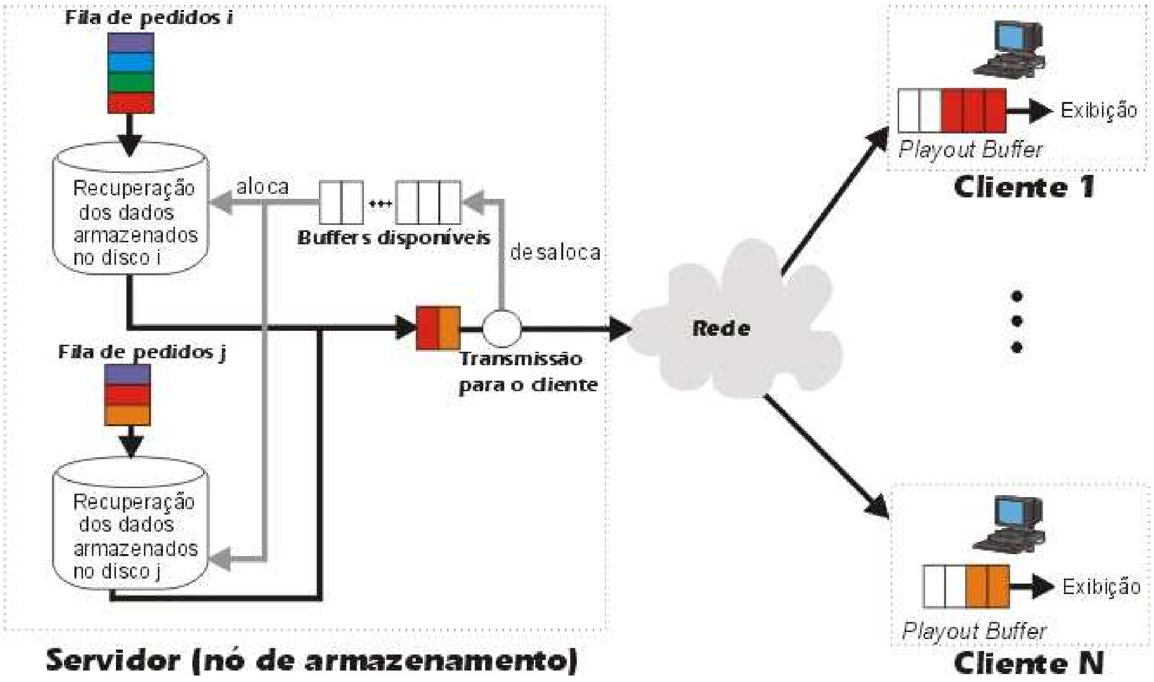

Figura 2.1: Fluxo dos Dados no Servidor Multimídia

Os dados (blocos recuperados) podem ser enviados ao cliente através de transmissão *unicast* ou *multicast*. Esta política de transmissão, a política de acesso e organização dos dados, o escalonamento dos pedidos e o controle de admissão de usuários devem ser considerados para atender aos requisitos de desempenho especificado para o servidor. Essas políticas serão apresentadas nas próximas seções, e algumas delas serão descritas junto com a arquitetura do RIO.

2.1.1 Políticas de Acesso

Um servidor multimídia pode enviar os blocos de dados aos clientes com taxa variável (VBR) ou constante (CBR), dependendo do mecanismo de compressão usado no fluxo a ser transmitido e da política de suavização (*smoothing*) usada. No envio com taxa variável, ou os blocos tem tamanho constante e o intervalo de tempo entre o envio de blocos consecutivos é variável, ou então, o tamanho dos blocos enviados é variável e o intervalo de tempo para envio é constante. No caso de envio com taxa constante, tanto o tamanho do bloco quanto o intervalo de tempo de envio destes é constante.

Existem duas políticas de acesso aos dados: *Constant Time Length* (CTL) e *Constant Data Length* (CDL). Em CDL, a quantidade de dados enviados é constante e o tempo de envio é variável. O contrário é determinado pela CTL, onde a quantidade de dados enviados é variável e o tempo de envio é constante. Essas políticas estão definidas em [32].

2.1.2 Escalonamento dos Pedidos

A ordem na qual os pedidos são enviados para cada disco é determinada pela política de escalonamento utilizada no servidor. Para recuperação de um bloco, deve-se determinar sua posição no disco que o contém. Para isso, a cabeça de leitura/escrita é movimentada do cilindro atual até o cilindro desejado. Este tempo é denominado *seek time*. Uma vez que a cabeça esteja posicionada corretamente, é necessário esperar que o setor inicial passe sob a cabeça, antes de iniciar a transferência. Este tempo é conhecido como tempo de rotação. O *seek time*, o tempo de rotação e ainda o tempo de espera na fila do disco determinam o tempo de serviço de um pedido de bloco. Em [23, 34] são apresentadas algumas técnicas para reduzir estes tempos. As técnicas SCAN e *bidirectional* SCAN reduzem o *seek time*. Os dois tempos envolvidos no tempo de serviço são minimizados pelas técnicas *Shortest Time First* (STF), *Grouped Shortest Time First* (GSTF) e *Weigthed Shortest Time First* (WSTF).

Em sistemas de tempo real, além da otimização do tempo de serviço feita nos algoritmos, as restrições de tempo para todos os clientes admitidos devem ser garantidas. Por isso, foram propostas algumas melhorias nos algoritmos de escalonamento dos discos. O *Earliest Deadline First* (EDF) determina que os pedidos são atendidos de acordo com o tempo que o bloco tem que ser enviado para o cliente. O algoritmo *Static Priority* (SP) implementa o atendimento dos pedidos baseados em sua prioridade. Já no *First Come First Served* (FCFS) os pedidos são atendidos de acordo com a ordem de chegada. Estes algoritmos podem ser combinados para alcançar resultados melhores.

2.2 Arquitetura do RIO

Neste trabalho foi utilizado o servidor multimídia RIO (*Randomized I/O Multimedia Storage Server*) [34]. Ele foi, inicialmente, desenvolvido na universidade da Califórnia (*University of California Los Angeles* - UCLA) e, atualmente, é utilizado em projetos de pesquisa pela Universidade Federal Fluminense (UFF), Universidade Federal do Rio de Janeiro (UFRJ) e Universidade Federal de Minas Gerais (UFMG). A versão utilizada neste trabalho é a 0.2, implementada em C++ para o sistema operacional Linux. O servidor RIO foi escolhido por ser um servidor multimídia universal, que suporta o armazenamento de diversos tipos de objetos (vídeo, áudio, texto, imagem) e possui flexibilidade para a utilização de técnicas que aumentem a QoS oferecida. O servidor RIO também faz parte do projeto de ensino a distância do CEDERJ (Centro de Ensino Superior a Distância do Estado do Rio de Janeiro) [1].

Alguns artigos foram publicados sobre RIO, dentre eles, a descrição do seu projeto e de sua implementação é encontrada em [30] e em [33] é feita a comparação e a análise de desempenho da técnica de armazenamento, utilizada pelo servidor RIO com a técnica tradicional de *striping* [33], que será vista na Seção 2.4.

A arquitetura do RIO é composta pelo nó servidor, pelos nós de armazenamento e pelos clientes como mostra a Figura 2.2. A comunicação entre estes módulos é realizada utilizando-se os protocolos TCP e UDP [27]. Através do protocolo TCP, são enviadas mensagens entre os clientes e o nó servidor e entre este e os nós de armazenamento. Estas mensagens são informações de controle (pedidos e comandos) e pedidos de blocos. Os dados solicitados são transmitidos através do protocolo UDP.

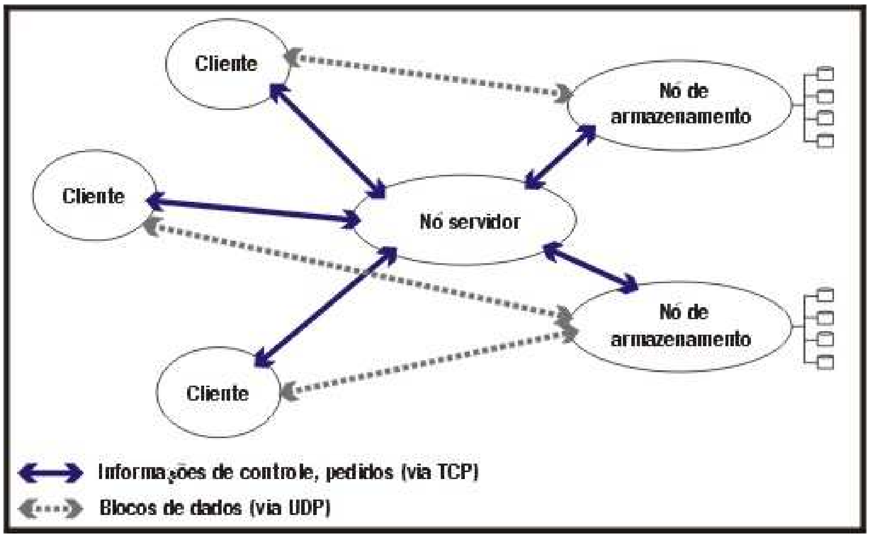

Figura 2.2: Arquitetura Básica do Servidor RIO

2.2.1 Nó Servidor

O nó servidor é responsável pelo gerenciamento do servidor RIO. É ele que faz o atendimento aos clientes e o processamento de suas requisições. Ele possui arquivos de configuração, dos objetos armazenados e dos discos utilizados. Cada arquivo representa um objeto do sistema e possui informações de seus atributos. Através dele é feito o mapeamento dos blocos para determinar sua localização nos discos do servidor.

O nó servidor mantém informações de todos os fluxos que se encontram abertos. Quando uma requisição de bloco é feita, ele faz o mapeamento do bloco, para verificar o disco que o contém, e envia a requisição ao nó de armazenamento responsável pelo processamento do pedido.

2.2.2 Nós de Armazenamento

Os nós de armazenamento são máquinas que guardam, em seus discos, os objetos solicitados ao servidor. Eles podem executar na mesma máquina do nó servidor e possuem um gerenciador, o *Storage Server* que atende a solicitação repassada pelo nó servidor e a executa se for válida.

2.2.3 Clientes

Os clientes são as aplicações responsáveis por interagir com o servidor. São eles que enviam pedidos de abertura de sessão, para comunicação com o nó servidor, abertura de fluxo, solicitam o tipo de operação que deve ser realizada (leitura ou escrita), informam o tipo de tráfego (de tempo real ou não) e fazem os pedidos de dados. Eles armazenam no *buffer*, (uma região de memória que é alocada) os dados recebidos dos nós de armazenamento e repassam ao tocador para que possam ser visualizados pelos usuários.

A comunicação entre os clientes e o nó servidor é feita por uma conexão TCP. Quando o nó servidor recebe uma requisição de um cliente ele verifica se será ele quem irá tratá-la ou se deve repassá-la aos nós de armazenamento. Se a requisição for, por exemplo, um pedido de abertura de sessão, de abertura de fluxo ou de abertura de objeto, então, será tratada pelo nó servidor. Caso seja um pedido de bloco, então o nó servidor repassará o pedido ao nó de armazenamento responsável por processá-la. A comunicação entre os clientes e os nós de armazenamento é feita por uma conexão UDP. Os clientes do servidor RIO serão vistos com maiores detalhes no Capítulo 4.

2.3 Componentes da Arquitetura do Servidor RIO

Nesta seção será dada um visão mais detalhada dos componentes nó servidor e nós de armazenamento, que fazem parte da arquitetura do servidor RIO. Eles são apresentados na Figura 2.3. O nó servidor possui interfaces para comunicação e tratamento de requisições, *SessionManager, StreamManager, ObjectManager* e *Router*. Cada nó de armazenamento possui um gerenciador, *Storage Server*, formado pelos seguintes componentes: *RouterInterface, ClienteInterface, StorageManager* e *StorageDevice*. Os detalhes de cada componente estão em [14].

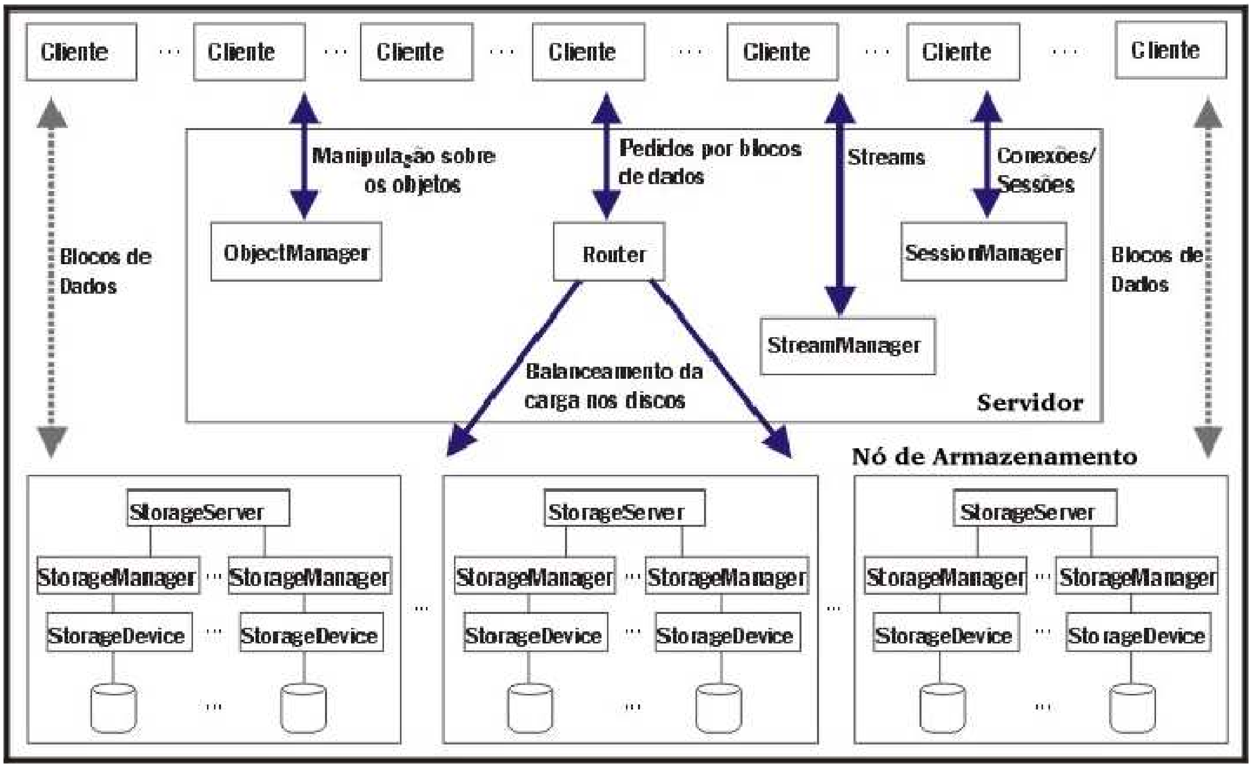

Figura 2.3: Componentes da Arquitetura do Servidor RIO

2.3.1 Interfaces do Nó Servidor

Gerenciador de Sessão (*SessionManager*)

O *SessionManager* é a "porta de entrada" dos clientes para o servidor RIO. Quando um novo cliente se conecta, o *SessionManager* cria uma *thread* para atendê-lo, e envia a resposta ao cliente se conseguiu ou não criar a sessão. Todos os pedidos deste cliente serão atendidos por sua *thread*.

Gerenciador de Fluxos (*StreamManager*)

Um cliente pode abrir mais de um fluxo dentro de uma mesma sessão, por exemplo, um fluxo sem restrições de tempo e outro de tempo real. A função de gerenciar os fluxos abertos e abrir novos fluxos é de responsabilidade do *StreamManager*. Durante a abertura dos novos fluxos, o *StreamManager* executa um controle de admissão para garantir a QoS de todos os clientes e a utilização eficiente dos recursos do sistema.

O controle de admissão utilizado pelo RIO é simples. O servidor mantém atualizadas a largura de banda disponível (Taxa_Total) e a taxa reservada para fluxos que não são de tempo real (Taxa_Reservada_NRT). Quando um novo cliente solicita a abertura de um fluxo, o servidor verifica os parâmetros informados no pedido, tipo de tráfego (de tempo real ou não), direção do tráfego (escrita ou leitura) e largura de banda solicitada (taxa solicitada para transmissão da mídia). Se o pedido feito for para um fluxo de tempo real, o servidor faz uma computação para verificar se pode admitir o novo cliente. Ele verifica se a taxa disponível para fluxos de tempo real (Taxa_Total - Taxa_Reservada_NRT) é maior ou igual à taxa disponível do sistema (Taxa_Alocada + Taxa_Solicitada), ou seja:

$$(Taxa\_Total - Taxa\_Reservada) \geq (Taxa\_Alocada + Taxa\_Solicitada) \quad (2.1)$$

O parâmetro Taxa_Alocada é inicialmente igual a zero. Se o cliente for admitido, a Taxa_Alocada passa a ser igual à Taxa_Alocada + Taxa_Solicitada, ou seja:

$$Taxa\_Alocada = Taxa\_Alocada + Taxa\_Solicitada \quad (2.2)$$

O gerenciador de fluxo coordena os pedidos feitos ao servidor. Cada fluxo possui uma fila de pedidos, conforme Figura 2.4. As filas são verificadas periodicamente pelo gerenciador de fluxo, que envia os pedidos ao *router*. Os fluxos de tempo real possuem prioridade no atendimento.

Gerenciador de Objetos (*ObjectManager*)

Também podem ser abertos mais de um objeto em um mesmo fluxo. O *ObjectManager* gerencia os metadados de todos os objetos armazenados no servidor. Metadados são os arquivos que contém informações dos objetos e discos do servidor. Cada objeto e cada disco possuem seu respectivo arquivo. É através dele que operações de criação, abertura, fechamento e exclusão de objetos são efetuadas.

Durante as operações de criação e fechamento de objetos é feita a alocação/desalocação

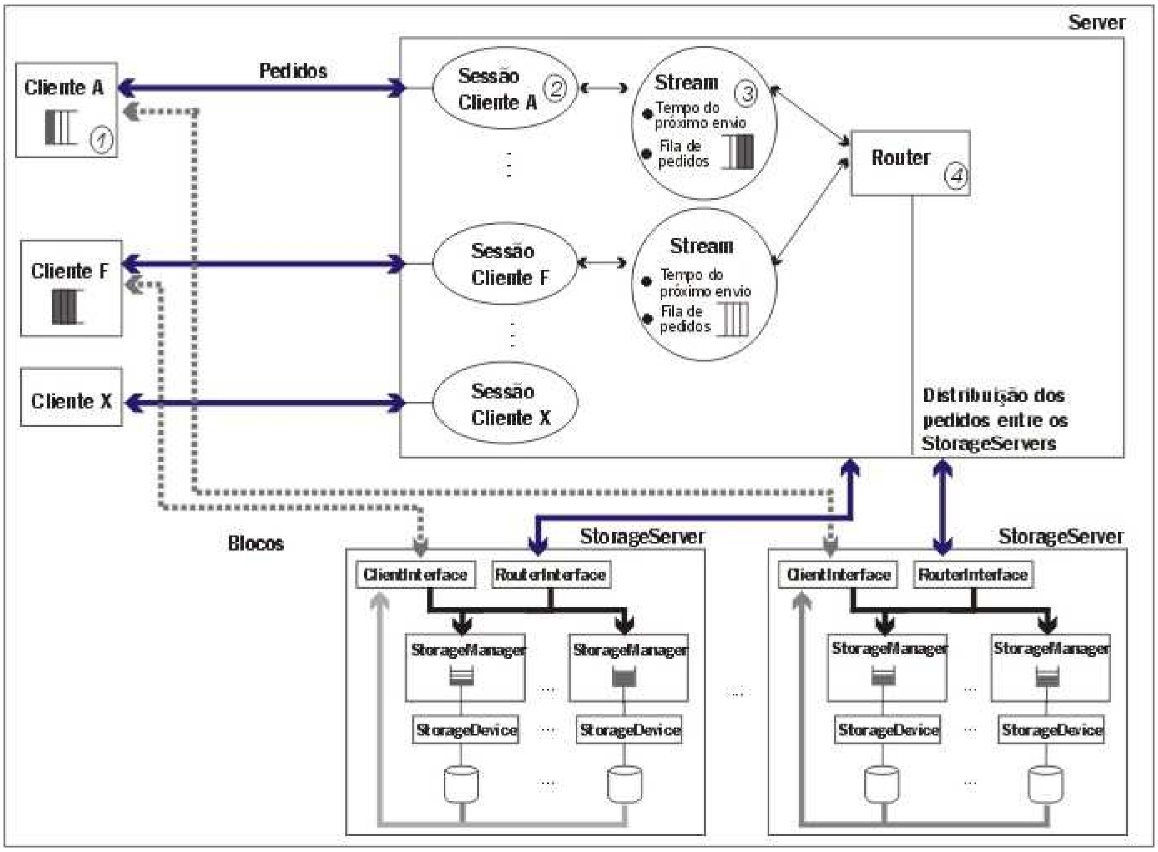

Figura 2.4: Sessões e Fluxos no Servidor RIO

de cada bloco e de suas réplicas, e os metadados dos objetos e dos discos são sempre atualizados.

Roteador (*Router*)

Cada disco possui uma fila para os pedidos de tempo real (RT) e outra para os pedidos sem restrições de tempo (NRT), como mostra a Figura 2.5.

O *Router* é responsável por encaminhar o pedido recebido do *Stream Manager* para a fila correta do disco. Como os blocos do RIO possuem réplica, o *Router* deve encaminhar o pedido do bloco solicitado para o disco que possuir a menor fila.

Caso seja um pedido de escrita, o *router* adiciona requisições em todas as filas dos discos que possuem o bloco e suas réplicas para que todos sejam atualizados.

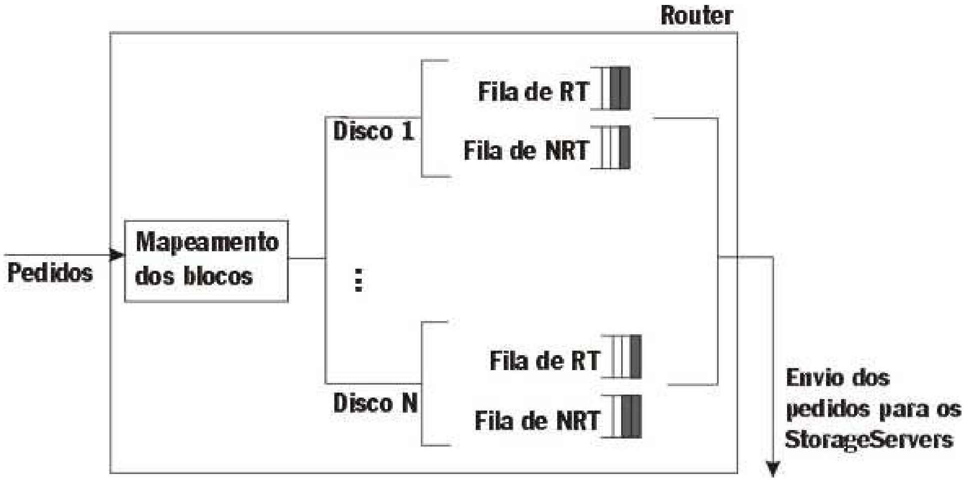

Figura 2.5: Router

2.3.2 Componentes do *Storage Server*

A Figura 2.6 ilustra os componentes do *Storage Server*.

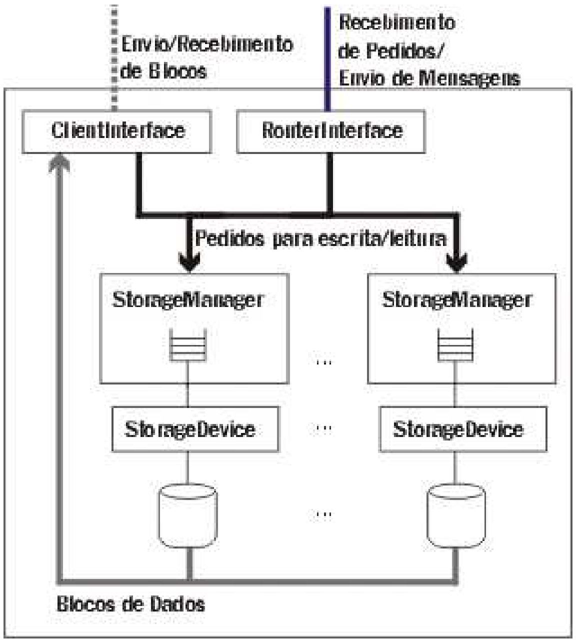

Figura 2.6: StorageServer

Interface com o Roteador (*RouterInterface*)

A comunicação entre o nó servidor e os nós de armazenamento é feita através de uma conexão TCP e gerenciada pelo *RouterInterface*. Ele é responsável pelo envio de mensagens ao nó servidor e pelo recebimento dos pedidos de bloco por ele repassados. Se a requisição recebida for de leitura/escrita de bloco, o *RouterInterface* repassa o pedido ao *StorageManager*.

Gerenciador de Armazenamento (*StorageManager*)

Cada *StorageDevice* é gerenciado por um *StorageManager*, que possui uma fila de pedidos a serem atendidos. Os pedidos recebidos do *RouterInterface* são escalonados na fila e atendidos utilizando o algoritmo de escalonamento FIFO (*First In First Out*).

No caso de um pedido de leitura, após a realização da operação, o *StorageManager* solicita ao *ClientInterface* o envio do bloco para o cliente. Para os pedidos de escrita, os blocos recebidos pelo *ClientInterface* são encaminhados ao *StorageManager* para o armazenamento em disco.

Dispositivo de Armazenamento (*StorageDevice*)

O *StorageDevice* realiza as operações de E/S no disco do sistema através de chamadas ao sistema operacional. Cada disco possui uma instância do *StorageDevice*.

Interface com o Cliente (*ClientInterface*)

Este componente é responsável pelo envio/recebimento dos blocos de dados. Estas operações entre o cliente e a *ClientInterface* são feitas através de conexão UDP. Foi desenvolvido um protocolo específico para o tratamento dos dados recebidos/enviados. Cada bloco possui 128KB e é dividido em 90 fragmentos. Entre as informações gerenciadas pelo protocolo estão: o endereço origem e destino, o número de fragmentos recebidos de cada bloco e o tratamento dos cabeçalhos que possuem a identificação do fragmento e do bloco que o compõe.

2.4 Organização dos Dados nos Discos

Os objetos armazenados nos servidores multimídia são divididos em blocos. Cada bloco é armazenado em um disco do sistema, de acordo com a técnica de armazenamento utilizada pelo servidor. Todos os blocos de um mesmo objeto podem ser armazenados em um mesmo disco, porém esta não é uma boa solução dado que o disco pode ficar sobrecarregado se o objeto armazenado nele for popular. Isto pode resultar no desbalanceamento de carga do sistema e limitar o número de clientes servidos. Para solucionar estes problemas, técnicas mais sofisticadas foram propostas, dentre elas a mais comum é a técnica de *Striping* [33, 34, 23, 30]. Ela é utilizada em servidores multimídia, cujo padrão de acesso é seqüencial com exigências de fluxos constante (*Constant Bit Rate* - CBR). Na técnica de *Striping*, os objetos são divididos em blocos de dados de tamanho fixo, chamados *stripe units*, e são distribuídos nos discos do sistema utilizando-se a política *round robin*, onde os blocos consecutivos de um objeto são armazenados em discos consecutivos. Os blocos são recuperados dos discos em ciclos com tempo de duração constante, ou seja, em um ciclo de duração T o bloco é recuperado do disco. No próximo ciclo o bloco será "consumido" pelo cliente enquanto o servidor recupera o próximo bloco do disco. Este comportamento é ilustrado na Figura 2.7. Esta figura apresenta a técnica de *Data Striping*. Neste esquema, em cada ciclo, o servidor recupera um bloco de dados para cada fluxo ativo. Na Figura 2.7, podemos ver a recuperação do objeto i, e na Figura 2.8 os ciclos para recuperação desse objeto.

No primeiro ciclo k, o primeiro bloco de dados (1) do objeto i é recuperado do disco 1. No próximo ciclo, k+1, o bloco de dados (1) recuperado é enviado e consumido pelo cliente, enquanto o sistema lê o próximo bloco de dados (2) do disco 2. Este esquema continua até que todo o objeto seja entregue ao cliente. A sobrecarga de qualquer disco é evitada pois, em cada ciclo, são lidos blocos de discos diferentes. Um novo cliente é admitido se houver largura de banda disponível em algum disco para atendê-lo. Ele deve aguardar até que o disco que contém o primeiro bloco do objeto requisitado seja alcançado.

A vantagem desta técnica onde objetos diferentes são armazenados em discos

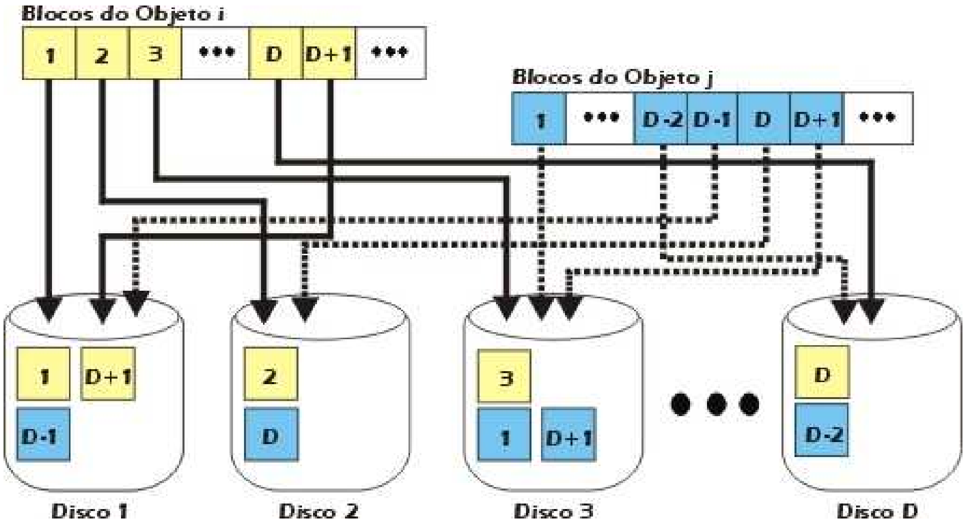

Figura 2.7: Técnica de Striping

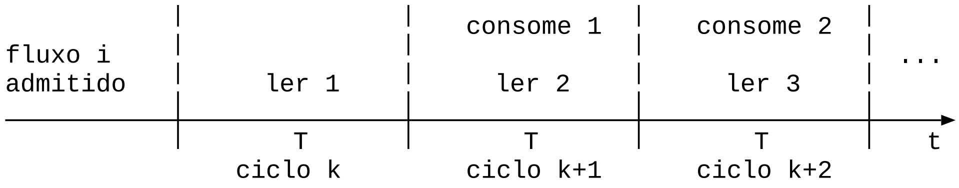

Figura 2.8: Ciclos com Tempo de Duração Constante

diferentes, é que ela evita o desbalanceamento de carga do sistema quando objetos populares são acessados. Entretanto, ela possui limitações. Primeiro, seu desempenho é vantajoso em cenários onde os cliente possuem requisitos de banda semelhantes e fluxos com taxas constantes (CBR), sendo reduzido quando fluxos com taxa variável (*Variable Bit Rate* - VBR) são utilizados. Segundo, para garantia de qualidade de serviço em operações de tempo real, o escalonamento dos discos é baseado na sincronização dos ciclos de todos os discos. Com isto, muitos discos podem ficar ociosos até o final do ciclo, reduzindo o desempenho do sistema.

Para solucionar essas dificuldades, foi proposta a técnica de *Random Data Allocation* [34, 33], onde os sistemas podem suportar qualquer tipo de aplicação multimídia, incluindo VBR e aplicações interativas com diferentes padrões de acesso. Esta técnica é a utilizada no servidor RIO. Ela se baseia na divisão de cada objeto em blocos de tamanho fixo que são armazenados em posições aleatórias de um disco. A escolha do disco também é aleatória, conforme Figura 2.9.

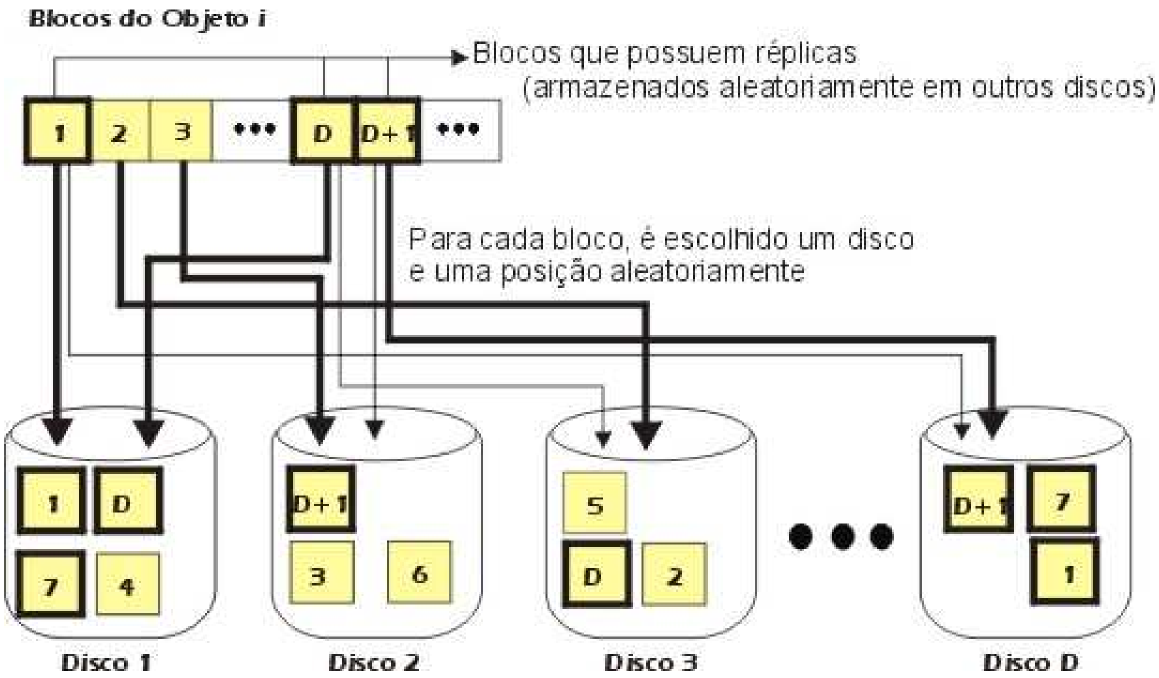

Figura 2.9: Técnica Random Data Allocation

Assim, qualquer padrão de acesso é mapeado no padrão de acesso aleatório. Entretanto, o sistema pode ficar desbalanceado se ocorrerem flutuações na carga dos discos. Para resolver este problema são utilizados, para o atendimento dos pedidos, a replicação dos blocos e um algoritmo que escolhe o disco com a menor fila [34, 30].

Com a técnica de *Random Data Allocation*, cada pedido é mapeado para o disco que contém a cópia do bloco requisitado e que possui a menor fila. O escalonamento dos discos não é baseado em ciclos de duração constante e eles não são sincronizados com os outros discos do sistema como na técnica de *Striping*. Todos os pedidos de um determinado disco são atendidos independentemente dos demais discos. O algoritmo de escalonamento utilizado nas filas dos discos é o FIFO. O balanceamento de carga e a confiabilidade do sistema são melhorados com o uso da replicação dos dados.

Em [33], são feitas comparações entre as técnicas *Random Data Allocation* com replicação de blocos, e a técnica de *Data Striping*. Os resultados obtidos mostram que a técnica de *Random Data Allocation* obtém desempenho competitivo com a técnica de *Data Striping* se não for utilizada a replicação de blocos. Com a utilização desta técnica, mesmo que parcial, a *Random Data Allocation* possui desempenho superior. A razão da técnica *Data Striping* ter desempenho inferior é a variabilidade no tempo das operações de I/O nos discos, onde estes ficam tempos ociosos.

2.5 Gerenciamento de *Buffers* e Controle de Admissão de Usuários

O gerenciamento de *buffers*, utilizado no RIO, implementa o mecanismo de *prefetching* de blocos de dados e possibilita a obtenção de vários atributos de um único cliente [14]. Entre eles, estão o tempo de ocorrência do último pedido efetuado pelo cliente, o RTT (*Round Trip Time*) deste cliente, informações dos *buffers*, tanto no servidor quanto no cliente, um vetor que contém informações dos blocos armazenados no *buffer* do servidor, entre outros. A implementação do gerenciamento de *buffers* está descrita em [14].

Para o mecanismo de *prefetching* utilizado, cada cliente possui um *buffer* associado ao servidor, onde são armazenados os próximos blocos a serem transmitidos. A cada pedido efetuado por um cliente, o bloco já armazenado no *buffer* é enviado, e, em seguida, é feita a requisição para a leitura do próximo bloco a ser armazenado nesta posição do *buffer*. Com isso, as flutuações na carga dos discos são compensadas.

O controle de admissão de usuários utilizado pelo RIO foi descrito na Seção 2.3.1, e um novo controle foi proposto em [14]. Sua idéia é utilizar todos os tempos envolvidos no atendimento de um pedido, juntamente com as informações, a *priori*, dos vídeos armazenados e as informações de cada cliente, para prever se um novo cliente pode ser admitido. Para isto, o servidor, no momento de admissão, faz uma simulação do sistema para verificar se o novo cliente pode ser admitido sem violar os requisitos de QoS. Em [14], o algoritmo utilizado para simulação do sistema é apresentado. Com o gerenciamento de *buffers* e com o controle de admissão propostos, é possível ter um controle detalhado do sistema através das medidas coletadas do servidor.

Diversos aperfeiçoamentos foram implementadas no servidor RIO para aumentar a QoS oferecida aos clientes [14], porém o gargalo (ponto de congestionamento) na entrega dos dados para as aplicações de mídia contínua está na transmissão do conteúdo pela rede até os clientes, e não no servidor RIO. Para minimizar a necessidade de largura de banda do servidor para transmissão da mídia, é necessário que ele possua mecanismos de compartilhamento de fluxo pelos clientes que requisitam o mesmo objeto. Estes mecanismos, que serão vistos no próximo capítulo, devem ter um tempo de resposta imediato (sem introdução de latência no atendimento dos pedidos) e possuir flexibilidade para tratar as operações de interatividade dos clientes.

Capítulo 3

Técnicas de Compartilhamento de Recursos

AS aplicações multimídia de tempo real possuem exigências de largura de banda da rede e serviço imediato. Estes requisitos são verdadeiros desafios para os sistemas multimídia, pois a Internet não provê garantias de qualidade de serviço na entrega dos dados. Neste capítulo serão abordadas algumas técnicas existentes para solucionar estes problemas, garantindo qualidade de serviço às aplicações de mídia contínua. Estas técnicas propoem o compartilhamento da banda do servidor utilizando-se transmissões *multicast*.

Ao longo do capítulo, serão discutidos os protocolos existentes para transmissão dos dados, transmissões *unicast* e *multicast*, as técnicas existentes para compartilhamento do canal e a questão da interatividade dos clientes.

3.1 Mídia Contínua

As aplicações multimídia de tempo real são denominadas aplicações de mídia contínua, pois são compostas por séries de dados que são apresentadas com certas restrições de tempo. São exemplos as aplicações interativas 3D, as aplicações de voz e vídeo e a televisão digital. Elas são altamente sensíveis ao atraso fim-a-fim e a variação de atraso, mas podem tolerar perda de dados. Estas características devem ser observadas durante a escolha do protocolo de comunicação a ser utilizado pela aplicação.

A Internet oferece dois protocolos de transporte: TCP (*Transmission Control Protocol*) e UDP (*User Datagram Protocol*) [27]. Eles se diferenciam pelos serviços oferecidos.

O protocolo TCP oferece diversos serviços às aplicações. Ele é um protocolo orientado à conexão, ou seja, antes que uma aplicação possa enviar dados para outra, os dois processos devem trocar mensagens para o estabelecimento da transmissão. Ele provê transferência confiável assegurando que os dados enviados por um processo serão recebidos corretamente e em ordem pelo outro. Para isso, são utilizados controle de fluxo, mensagens de reconhecimento e *timers*. O controle de fluxo estabelece a taxa máxima de envio de dados no emissor, baseado nas informações da quantidade de dados que o receptor pode processar. O TCP também provê comunicação *full duplex*, ou seja, dois processos em máquinas diferentes podem se comunicar simultaneamente. Além disso, ele faz um controle de congestionamento, diminuindo a taxa de transmissão da aplicação quando a rede encontra-se congestionada, prejudicando, assim, o desempenho de aplicações multimídia. Outra característica é a detecção de erros através de campos que são inseridos no cabeçalho dos pacotes, comum aos dois protocolos.

Já o UDP não é orientado a conexão. Deste modo, não há atraso no início da transmissão. Ele é conhecido como um protocolo não-confiável, pois não há garantias de entrega dos dados, e caso eles cheguem ao receptor, podem estar fora de ordem. Além disso, ele não faz controle de congestionamento, tendo uma latência menor que o TCP. Sendo assim, é o protocolo de transporte mais indicado em aplicações de tempo real que dependem da entrega rápida dos pacotes. Outra vantagem do UDP é a sua utilização em transmissões *multicast*, não sendo possível no protocolo TCP, pois as conexões são sempre ponto-a-ponto.

Devido às vantagens da utilização do UDP em aplicações de tempo real, ele é utilizado nos servidores multimídia para transmissão dos dados aos clientes. Esta transmissão pode ser feita de duas formas: *unicast* ou *multicast*.

Em uma transmissão *unicast*, é enviada uma cópia do pacote de dados para cada cliente que fez a solicitação. Este tipo de transmissão pode tornar-se ineficiente quando um mesmo conjunto de dados é enviado para muitos receptores ao mesmo tempo, pois no mesmo canal podem estar passando várias cópias de um mesmo pacote. Ela pode ser otimizada, reduzindo o número de pacotes duplicados no canal com o uso da transmissão *multicast*, que é voltada para aplicações do tipo um-para-muitos, onde somente um *host* envia os dados para os demais e muitos-para-muitos, onde todos os *hosts* podem enviar e receber dados simultaneamente. Nesta transmissão apenas um pacote de dados é transmitido na rede para um endereço de grupo *multicast*. Todos os clientes que recebem transmissões *multicast* de um mesmo vídeo devem pertencer ao mesmo grupo. Os roteadores enviam, então, uma réplica do pacote a todos os clientes que pertencerem ao grupo. Assim, com a transmissão *multicast*, a rede é utilizada de forma mais eficiente.

A maioria dos servidores multimídia existentes são implementados utilizando-se um canal dedicado a cada cliente para a transmissão da mídia (transmissão *unicast*) [21]. Com isso, pode-se acabar a largura de banda disponível do servidor. Para solucionar este problema, foram propostas diversas técnicas de compartilhamento da banda pelos clientes para que haja uma melhor utilização dos recursos da rede.

3.2 Técnicas de Compartilhamento de Recursos

De acordo com [21], os mecanismos de compartilhamento de banda são divididos em dois grupos: orientado a pedidos ou *client request oriented* e transmissões periódicas ou *periodic broadcast*. No primeiro, o servidor envia os blocos de dados para os clientes, à medida que são solicitados. No segundo, o servidor envia os blocos periodicamente.

Servidores *proxy* também podem ser utilizados para aproveitar melhor os recursos da rede. Sua implementação depende da infra-estrutura do sistema. Ele pode ser combinado com as técnicas que serão vistas neste capítulo. Existem propostas de utilizar transmissão *unicast*, entre o servidor e os *proxies*, e transmissão *multicast*, entre os *proxies* e os clientes.

Alguns dos mecanismos de compartilhamento de banda que se baseiam na orientação a pedidos são: *Batching* [16], *Stream Tapping* [15], *Patching* [13, 25], *Controlled Multicast* [22], *Piggybacking* [24, 29], *Hierarchical Stream Merging* (HSM) [19, 20] e *Bandwidth Skimming* [17]. Estes mecanismos serão vistos na Seção 3.2.1.

Skyscraper Broadcasting (SB) [26], *Pyramid Broadcasting* (PB) [36] e *Permutation-Based Pyramid Broadcasting* (PPB) [11] são alguns mecanismos baseados em transmissões periódicas, vistos na Seção 3.2.2.

3.2.1 Técnicas Orientadas a Pedidos

A técnica de *batching* possui uma abordagem bastante simples. Ela se baseia no acúmulo dos pedidos feitos pelo mesmo vídeo em um curto período de tempo, formando-se um grupo, para que todos possam ser servidos por um único fluxo *multicast*. Existem duas abordagens em relação à banda do servidor. O esquema que supõe a banda do servidor infinita, utiliza uma janela para que os clientes possam ser servidos. Esta janela é determinada de acordo com o tempo que o cliente está proposto a esperar pela transmissão. Nesta situação, quando um novo pedido chega ao servidor, a janela é iniciada. Todos os pedidos que chegarem para o mesmo vídeo e estiverem no limite da janela, pertencerão ao mesmo grupo. Quando a janela expira, a transmissão é iniciada. A Figura 3.1 ilustra a técnica de *batching*. O primeiro pedido r_0 para o fluxo s_i inicia a janela do *batching*. Os pedidos r_1 e r_2, também para o fluxo s_i, chegam dentro do limite da janela e são agrupados. Quando a janela expira, eles são servidos pela mesma transmissão *multicast* de s_i.

A outra abordagem admite que a banda do servidor é limitada devido à escassez dos canais de transmissão. Assim, os clientes que requisitam o mesmo vídeo também são agrupados para serem servidos, e como podem existir vários grupos esperando a transmissão de vídeos diferentes, o servidor deve adotar uma política de escalonamento para selecionar o vídeo que deve ser transmitido em determinado momento,

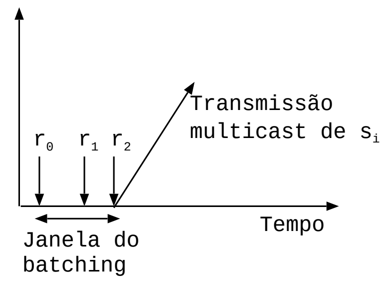

Figura 3.1: Técnica de Batching

e que satisfaça um número máximo de pedidos.

Em [16] é feita a análise de algumas políticas de escalonamento: FCFS (*First Come First Served*), FCFS-n (n é a quantidade de vídeos mais populares) e MQL (*Maximum Queue Lenght*). Os objetivos propostos foram reduzir a probabilidade dos clientes finalizarem a conexão devido ao atraso inicial que é introduzido pela técnica de *batching*, e reduzir o tempo médio de espera antes de um cliente ser servido. A política FCFS sempre seleciona o cliente que esteja esperando a mais tempo. Com isso, todos os clientes que estejam esperando a transmissão do mesmo vídeo são servidos pela transmissão *multicast*. Já a MQL escolhe o vídeo que possui um número maior de clientes na espera. Os resultados obtidos mostraram que a política FCFS possui um desempenho melhor, caso a probabilidade de desistência dos clientes dependa do tempo de espera. Com isso, foram extraídas informações sobre o tempo mínimo e máximo de espera dos clientes. A política FCFS teve melhor desempenho quando a alocação de canais dedicados a vídeos populares, foi feita baseada nas informações de tempo máximo de espera dos clientes.

Embora a utilização de transmissões *multicast* diminua a necessidade da largura da banda do servidor e reduza o custo do sistema, apenas os vídeos populares são beneficiados com a utilização da técnica de *batching*. Os vídeos não-populares são prejudicados pelo aumento do tempo de espera dos clientes e pela baixa probabilidade da chegada de um novo pedido para o mesmo vídeo. Com isso, longos tempos de espera podem causar a desistência de um cliente.

Por isso, foram desenvolvidas algumas técnicas que oferecem serviço imediato aos clientes, porém possuem particularidades quanto à largura de banda e ao espaço do *buffer* dos clientes. A técnica de *piggybacking* requer valores mínimos para a banda de transmissão e para o *buffer* dos clientes [20]. Porém, se a banda de transmissão dos clientes for maior do que a *playrate* (taxa na qual um arquivo deve ser transmitido para que chegue no tempo previsto pelo cliente), algumas técnicas podem alcançar melhoras significativas [15, 13, 22, 36, 26].

Stream Tapping/Patching/Controlled Multicast

A técnica de *patching* introduz uma melhora significativa em relação à técnica de *batching*. Neste esquema, o servidor também mantém uma fila de espera, e todos os pedidos que chegam são adicionados à fila para serem servidos quando um canal torna-se disponível. A política de escalonamento utilizada pode ser FIFO, MQL ou MFQ (*Maximum Factored Queue Length First*) - uma melhora das duas anteriores. Os pedidos admitidos recebem os dados através de uma transmissão *multicast*. Caso exista uma transmissão do mesmo vídeo, já iniciada (fluxo 1), o novo grupo admitido "escuta" a transmissão em andamento, armazenando os dados recebidos em um *buffer*. Para receber a parte inicial da transmissão que foi perdida, é aberto um novo canal com o servidor, conhecido como *patch*. Assim que o cliente recebe estes dados, eles são imediatamente visualizados. Quando o cliente tiver recebido todos os dados do *patch*, este canal é liberado. O cliente passará a "consumir" os dados armazenados em seu *buffer* e continuará recebendo os dados do fluxo 1 até o final da transmissão. Por isso, o cliente deve ser capaz de receber dois fluxos simultâneos.

Ao contrário da técnica de *batching*, onde cada canal alocado transmite todo o vídeo, a técnica de *patching* oferece serviço imediato, já que não é necessário esperar o fim de uma transmissão *multicast*. Com isso, as transmissões existentes podem ser compartilhadas melhorando, assim, a eficiência do *multicast*.

A Figura 3.2 ilustra a técnica de *patching*. O pedido r_0 inicia a transmissão *multicast* para o fluxo s_i e a janela do *patching*. Os pedidos r_1 e r_2, feitos para o mesmo fluxo, chegam dentro da janela. Assim, estes pedidos "escutam" a transmissão *multicast* de s_i e recebem o *patching* nos instantes $(t_1 - t_0)$ para r_1 e $(t_2 - t_0)$ para r_2.

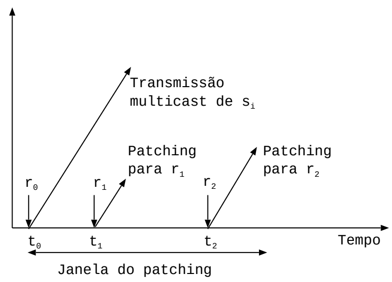

Figura 3.2: Técnica de Patching

Duas outras técnicas propostas, *Stream Tapping/Controlled Multicast*, também utilizam a idéia de *merging*. Elas são muito similares e se diferem no cálculo de uma janela ótima (w_i) [22, 13] para cada fluxo. A janela é o intervalo mínimo entre os instantes iniciais de duas transmissões completas do mesmo fluxo [21], ou seja, período de tempo de uma transmissão *multicast* no qual um *patching* pode ser utilizado. Ela não pode ser muito grande senão a maioria dos canais disponíveis serão alocados para o *patching*, porém não pode ser muito pequena, senão o compartilhamento dos fluxos será ineficiente. Assim, é necessário que o tamanho da janela seja suficiente para aumentar a eficiência destas técnicas.

Em [13] é desenvolvida uma fórmula para determinar o tamanho para a janela ótima do *patching*. São feitas simulações e comparações com outras técnicas existentes, mostrando o desempenho do *patching*. Em [22] é proposto e implementado o algoritmo CIWP (*Client-Initiated-With-Prefeching*), que utiliza uma janela ótima para a transmissão de fluxos completos e parciais. É feita a derivação da janela que minimiza o número de canais necessários para transmissão.

Adaptive Piggybacking

Outra técnica utilizada para reduzir a largura da banda do servidor é chamada *Adaptive Piggybacking*. Seu objetivo é unir os fluxos que transmitem o mesmo vídeo, através da alteração da taxa de exibição. Ela funciona da seguinte maneira: caso existam dois fluxos transmitindo o mesmo vídeo, a taxa de exibição de um fluxo é aumentada e a do outro é diminuída, até que as transmissões se tornem idênticas. Quando este ponto é alcançado, elas se unem e um canal de transmissão é liberado.

Umas das maneiras de diminuir a taxa de exibição é através da adição de *frames* ao vídeo, ou seja, um tocador que possui uma taxa de exibição constante, necessitará sempre de um número fixo de *frames* para exibir o vídeo. Caso este número aumente, a taxa de exibição diminui pois ele levará mais tempo para "consumir" uma quantidade maior de dados. Com isso, a taxa de exibição poderá diminuir até que seja alcançada por outra transmissão. Em [24], é feito o estudo desta técnica utilizada em conjunto com a técnica de *batching*. Sua limitação é que a variação da taxa de exibição deve ser de ±5%, senão a qualidade de exibição é prejudicada e percebida pelo usuário. Este fato limita o número de *merges* no sistema.

Hierarchical Stream Merging

A técnica HSM (*Hierarchical Stream Merging*) se baseia na formação de *merges* hierárquicos. Utilizando-se a idéia do *patching*, o cliente deve "escutar" dois fluxos simultâneos: o fluxo iniciado por ele e outro já existente, se existir. Quando o cliente se unir a um fluxo existente, o canal alocado para sua transmissão deve ser liberado. O novo fluxo do *merging* começa então a "escutar" uma outra transmissão, já em andamento, do mesmo vídeo até poder se unir a esta. Dessa forma, cria-se uma árvore de *merges* onde os clientes irão sempre se unir a fluxos existentes, compartilhando os canais de transmissão e liberando os que não são mais utilizados. A Figura 3.3 mostra a árvore de *merging* do HSM.

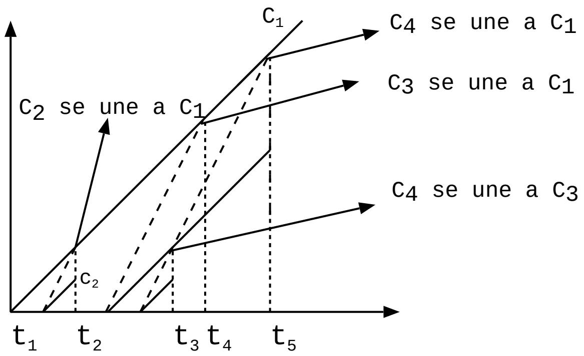

Figura 3.3: Técnica Hierarquical Stream Merging

Os clientes c_1, c_2, c_3 e c_4 fazem pedido para o mesmo fluxo s_i. O servidor inicia um fluxo *multicast* para cada um dos clientes. c_1 chega no tempo t_1. Em t_2, o cliente c_2 que está "escutando" o fluxo de c_1 se une a ele. No tempo c_3, o cliente c_4 se une a transmissão de c_3. Então, após t_3, o cliente c_4 pode "escutar" a transmissão *multicast* iniciada em t_1. Em t_5, o cliente c_4 se une a transmissão *multicast* iniciada em c_1.

Em [19], são propostas algumas políticas de *merges*, chamadas de *early merging*, para determinar a ordem de união dos fluxos que transmitem o mesmo vídeo, que minimize a largura total de banda do servidor. As políticas propostas foram *Earliest Reachable Merge Target* (ERMT), *Simple Reachable Merge Target* (SRMT) e *Closest Target* (CT). Na primeira, a junção de fluxos é sempre realizada com o fluxo mais próximo, desde que nenhum outro fluxo tardio se junte antes. A política SRMT simplifica a anterior, determinando os pontos de união alcançáveis mais próximos. A terceira define qual será o próximo fluxo para o *merge*, escolhendo o fluxo que foi iniciado mais cedo e que é o mais próximo. Os resultados obtidos em [19] demonstram que estas políticas alcançam um desempenho próximo ao do *offline hierarquical merging*, onde todos os instantes de chegada de pedidos são conhecidos a priori.

Bandwidth Skimming

Bandwidth Skimming (BS) é uma técnica de compartilhamento de banda que utiliza o *merging* hierárquico. Nesta técnica, utiliza-se uma banda remanescente, chamada de *skim*, para realizar o *merging* hierárquico quase ótimo e, com isso, conseguir reduções significativas na banda do servidor. Para obter a banda remanescente, a mídia é codificada em uma taxa um pouco menor que a banda do cliente. Porém, como a banda remanescente é pequena, a tarefa de realizar *merges* torna-se difícil.

Em [17], são propostas algumas políticas de BS (*Partition, Latest Patch, Shared Latest Patch, Discontinuous Patch*) que utilizam uma pequena e ajustável banda remanescente, suficientemente pequena para não comprometer a qualidade para conseguir um *merging* hierárquico quase ótimo e alcançar reduções nos custos de banda no servidor e na rede.

Em [20], é feito um estudo da largura de banda necessária pelas técnicas de *merging*. Assumindo que a chegada dos clientes é *Poisson*, a largura de banda necessária para os esquemas de *patching, stream tapping e controlled multicast* é calculada através da fórmula:

$$B = \sqrt{2N_i + 1} - 1 \tag{3.1}$$

Onde N_i é o número médio de pedidos para o arquivo i que chegam durante o período de tempo T_i ($N_i = \lambda_i T_i$). Portanto, a largura de banda do servidor cresce com a raiz quadrada da taxa de pedidos do cliente.

Para *bandwidth skimming* e *hierarchical stream merging*, a largura de banda necessária no servidor, definida em [21], é dada por:

$$B \approx \eta_b ln(N_i/\eta_b + 1) \tag{3.2}$$

Onde N_i é o número médio de pedidos para o arquivo i que chegam durante o período de tempo T_i ($N_i = \lambda_i T_i$) e η_b é uma constante real positiva que satisfaz:

$$\eta_b[1 - (\eta_b/(\eta_b + 1))^b] = 1 \tag{3.3}$$

3.2.2 Técnicas com Transmissões Periódicas

As técnicas que fazem parte deste grupo são baseadas na divisão do fluxo em segmentos e na largura de banda disponível em k canais lógicos. Os segmentos são transmitidos simultaneamente e em tempos periódicos nos diferentes canais.

Pyramid Broadcasting (PB) e *Permutation-Based Pyramid Broadcasting* (PPB)

Em [36], a técnica PB é introduzida. A idéia é dividir o objeto requisitado em k segmentos de tamanhos diferentes, de acordo com uma série geométrica, e a largura de banda do servidor é dividida em k canais lógicos. Os segmentos do vídeo são transmitidos nos k canais existentes. Como a taxa de transmissão dos canais é muito alta, a largura de banda dos clientes também é muito alta. Com isso, o *buffer* do cliente deve ter capacidade de armazenar uma grande quantidade de dados. Assim, faz-se necessária a utilização de discos, com largura de banda extremamente alta, para armazenar os dados tão rápido quanto são recebidos. A técnica PPB é similar à PB, porém ela faz a divisão de um canal em k sub-canais com taxa de transmissão menor. Como os sub-canais utilizam largura de banda menor para transmitir os segmentos, a largura de banda do disco para armazenar os dados recebidos também é menor. Para reduzir as exigências de disco, PPB pode pausar o fluxo de chegada do cliente para que ele possa obter os dados na taxa de *playback*. Para isso, o cliente deve ser capaz de obter o mesmo segmento, um pouco mais tarde, de outro sub-canal que esteja transmitindo-o. Porém, isto é significantemente mais complexo que a técnica PB e mais difícil de implementar, dado que o cliente pode sair de um canal e entrar em outro durante uma tranmissão.

Skyscraper Broadcasting **(SB)**

Esta técnica também faz a divisão da banda de transmissão em k canais lógicos e a divisão do vídeo em k segmentos. Cada segmento é transmitido à taxa de *playback* do vídeo em um canal dedicado. A Figura 3.4 mostra o tamanho dos segmentos sendo transmitidos nos canais alocados, de acordo com a série [1,2,2,5,5,12,12,25,25,52,52,...], onde W é o maior segmento restringindo o aumento do *buffer* de recebimento, que pode ser, no máximo, igual a W. A recepção dos clientes é feita em termos de grupos, dado que segmentos consecutivos têm o mesmo tamanho. Assim, o primeiro segmento forma o primeiro grupo, o segundo e o terceiro formam o segundo grupo e assim por diante.

A Figura 3.4 mostra o tempo de chegada de dois clientes no canal de transmissão: um anterior ao terceiro segmento e outro anterior ao 18^o segmento. O escalonamento de transmissão dos clientes é representado pelos segmentos em destaque. O tamanho máximo do *buffer* do cliente é igual ao tamanho do maior segmento.

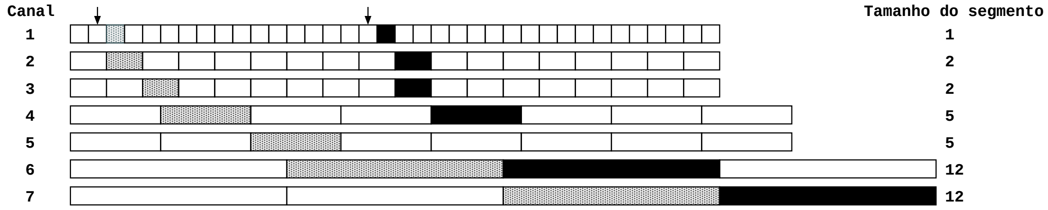

Figura 3.4: Técnica Skyscraper Broadcasting

Em [26], a técnica SB é proposta e comparada com as técnicas de PB e PPB. Para análise das exigências de armazenamento em *buffer* da técnica SB, foram feitas derivações matemáticas. Em [18], é proposta a técnica *Dynamic Skyscraper*, que melhora o desempenho da técnica SB. Ela considera a popularidade dos vídeos e assume taxas menores de chegada dos clientes.

Em [20], a largura de banda necessária do servidor para técnica SB foi analisada e o resultado alcançado foi uma significante diminuição da banda necessária, em comparação com as técnicas *optimized stream tapping, optimized patching* e *optimized controlled multicast*. Foi proposta também uma nova técnica chamada *Hierarchical Multicast Stream Merging*. Ela demonstrou maior economia na largura de banda necessária do servidor quando comparada a técnica *Dynamic Skyscraper*.

3.2.3 Servidores *Proxy*

As técnicas, citadas anteriormente, propõem o uso de transmissões *multicast* entre os clientes e o servidor. Porém, a utilização do IP *multicast* ainda é muito restrita, pois estes endereços não estão configurados nos roteadores da Internet. Isto se deve à complexidade e aos problemas de escalabilidade nos roteadores por manterem uma tabela com o estado de todos os clientes do grupo *multicast*. Por isso, há necessidade do desenvolvimento de estratégias eficientes que utilizem transmissões *unicast*. A utilização de *proxies*, combinada com as técnicas vistas anteriormente e com a utilização de transmissões *multicast* e *unicast*, pode reduzir a carga gerada no servidor, a latência de acesso e o tráfego na rede. Porém, a eficiência destes mecanismos depende da forma de armazenamento utilizados pelos *proxies*. Eles podem utilizar o armazenamento dos quadros iniciais dos vídeos, também chamado de prefixo do vídeo, ou dos quadros restantes, denominado sufixo. Como a maioria das transmissões começa com pedidos ao início do vídeo, o armazenamento do prefixo pode reduzir o custo das transmissões.

Em [35], é descrita a implementação de *proxy* que utiliza o armazenamento dos quadros iniciais do vídeo. Eles são enviados assim que o cliente faz um pedido, enquanto o restante do vídeo é solicitado ao servidor. Para isso, o *proxy* utiliza dois *buffers*: um para armazenar os quadros recebidos do servidor e outro para armazenar os quadros iniciais que serão enviados ao cliente.

Em [37], as técnicas orientadas a pedidos são combinadas com o uso de *proxies*. Os seguintes cenários foram analisados: *Unicast suffix batching (SBatch)*, *Unicast patching with prefix caching (UPatch)*, *Multicast patching with prefix caching (MPatch)* e *Multicast merging with prefix caching (MMerge)*.

O primeiro cenário, *SBatch*, permite ao cliente exibir o vídeo instantaneamente, pois o prefixo já está armazenado no *proxy*. Quando um cliente solicita um vídeo, o *proxy* lhe transmite imediatamente o prefixo. Todos os clientes que chegam durante o tempo V (duração do prefixo) são agrupados para receber o restante do vídeo. O sufixo é enviado pelo servidor para o *proxy* através de uma conexão *unicast*. Esta técnica não introduz *delay* no início da transmissão como a técnica de *batching*.

A técnica *UPatch* é utilizada no contexto *unicast*, pois o *proxy* pode encaminhar uma cópia para vários clientes. Este esquema é ilustrado na Figura 3.5. Quando ocorre a chegada de pedido, o *proxy* envia o prefixo armazenado (v_i). Se a chegada de um novo cliente ocorre após o término do envio do prefixo (t_2), o *proxy* pode escalonar a transmissão do sufixo completo, ou um *patch* do sufixo ($t_2 - v_i$), desde que o segmento $L_i - t_2$ seja escalonado para ser transmitido imediatamente. A decisão de transmitir o sufixo completo ou o *patch* depende do sufixo *threshold*. Se o novo cliente chegar antes do sufixo *threshold*, então, ele recebe o *patch*, senão ele recebe o sufixo completo.

MPatch é a técnica que permite ao *proxy* usar o esquema de transmissão *multicast* para os clientes. Na chegada do primeiro cliente, este recebe imediatamente o prefixo

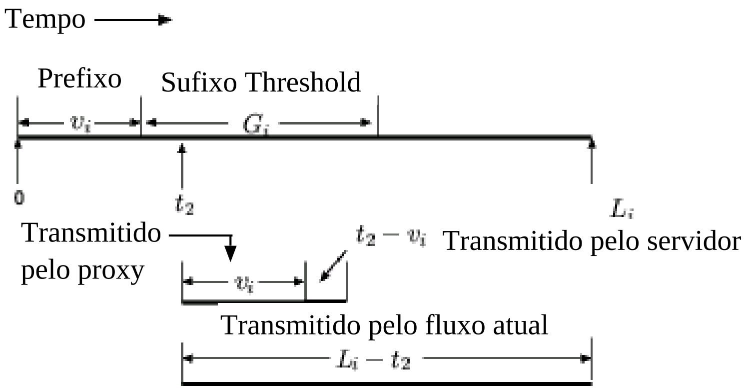

Figura 3.5: Unicast patching com prefix caching (UPatch)

em uma transmissão *multicast*. O servidor inicia a transmissão do sufixo no tempo V e o *proxy* transmite os dados recebidos via *multicast* para os clientes. As outras requisições podem começar uma nova transmissão *multicast* ou entrar no grupo já existente e usar um fluxo *unicast* separado para receber os dados que estão faltando.

MMerge utiliza a técnica de *stream merging* com a política *Closest Target*. Quando um cliente faz uma requisição, se o prefixo do vídeo requisitado estiver armazenado no *proxy*, ele é enviado imediatamente. O sufixo, que fica armazenado no servidor, é transmitido direto deste, a tempo de não comprometer a exibição do vídeo no cliente.

Ainda neste trabalho [37], foi proposta uma técnica para determinar analiticamente qual a alocação ótima de prefixo na *cache* do *proxy*. Os resultados obtidos mostraram que o custo da transmissão, quando o prefixo do vídeo é armazenado no *proxy*, é menor do que o armazenamento completo do vídeo.

3.3 Problemas em Aberto

As técnicas analisadas neste capítulo se diferenciam pelos serviços oferecidos: transmissão imediata ou não, número de canais lógicos alocados pelo servidor - e, com isso, espaço de armazenamento no cliente - técnicas para escolha da transmissão à qual será feito o *merging*, janela ótima para que o cliente possa compartilhar uma transmissão, combinação de transmissões *multicast* e *unicast*. Os casos estudados demonstram que há economia da banda do servidor quando os clientes podem compartilhar as transmissões já existentes. As simulações feitas admitem que a banda de transmissão é ilimitada, o tamanho do *buffer* dos clientes não diminiu a performance das técnicas, o acesso é seqüencial e as requisições possuem o tamanho de todo o objeto e são entregues sem interrupção até o fim [25, 20, 15, 19].

Porém, existem aplicações que exigem um comportamento diferenciado destas técnicas. É o caso das aplicações de ensino a distância, onde não se pode admitir que os clientes irão assistir ao vídeo até o final. Essas aplicações são caracterizadas por uma alta taxa de interatividade dos clientes. Vamos supor o caso onde vários clientes estejam assistindo a um vídeo de uma aula. Eles iniciam a visualização da aula, porém, em determinados momentos, os clientes voltam o vídeo para entender melhor alguma eaplicação feita pelo professor, pausam o vídeo para fazer alguma anotação, avançam no vídeo, pois podem ter entendido a parte inicial da aula e desejam avançar até o ponto chave para entender melhor, e até podem parar de assistir a aula antes que esta termine. Estas ações tendem a acontecer várias vezes durante a visualização de uma aula, por isso esta questão é muito importante no contexto das aplicações de ensino a distância.

Existem alguns estudos sobre interatividade em aplicações de tempo real. Em [28], é utilizado o modelo de cadeias de Markov ocultas (HMM) para modelar o comportamento dos clientes utilizando o sistema *Multimedia Asynchronous Networked Individualized Courseware* (MANIAC). As ações de cada cliente foram coletadas e gravadas em um arquivo utilizado no modelo. Três algoritmos foram propostos, utilizando-se HMM de forma diferente, e foram feitas previsões do comportamento futuro do cliente. Em [20], é feita uma breve reflexão sobre as questões de interatividade. Mecanismos que incorporam funções de interatividade em aplicações com transmissão *multicast* são propostos em [12]. Alguns mecanismos incluem a reserva de canais de emergência para prover a continuidade das funções de interatividade. Foram feitas simulações em sistemas com e sem interatividade, utilizando transmissões *unicast* e *multicast*. Os resultados obtidos mostraram que o sistema com transmissão *multicast* contínua é altamente complexo, o cenário com transmissão *multicast* descontínua tem complexidade moderada e quando as transmissões são *unicast* contínuas a complexidade do sistema é menor.

Esta questão ainda é pouco explorada na literatura. Como este trabalho também abrange um projeto de ensino a distância, a implementação de uma técnica de compartilhamento de banda no servidor RIO deve tratar o comportamento dos clientes, garantindo QoS e economia de banda do sistema.

Capítulo 4

Cliente Windows

NESTE capítulo são apresentados os clientes existentes para a interação com o servidor RIO e o cliente Windows desenvolvido neste trabalho. Serão explicados a necessidade e o funcionamento de cada cliente, bem como os módulos existentes nessas aplicações.

Como já mencionado no Capítulo 3, o objetivo deste trabalho é atender a aplicação de ensino a distância que possui alta interatividade dos clientes. Nesta aplicação, os clientes assistem a vídeos de aulas gravadas, através dos clientes desenvolvidos, podendo interagir com o sistema.

4.1 Introdução

Como foi visto no Capítulo 2, os clientes são as aplicações que se comunicam com o servidor. Os clientes desenvolvidos para o servidor RIO são do tipo "*client driven*", portanto fazem o envio de pedidos de dados além de trocar mensagens de controle, armazenar e gerenciar os dados recebidos. São ainda responsáveis por repassar os dados para o tocador. O tocador é o programa utilizado para visualização dos vídeos pelos usuários, como o *Windows Media Player*. A Figura 4.1 mostra os componentes básicos do sistema. Os clientes possuem uma interface com o servidor RIO, que é composta por módulos responsáveis para gerenciamento das etapas de comunicação.

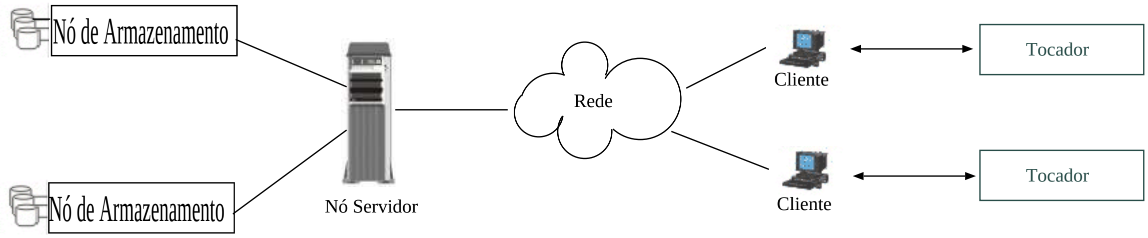

Figura 4.1: Componentes do Sistema

Existem quatro clientes que interagem com o servidor RIO. São eles: RIOMM-Client, RIOMTV, RIOSH, RIOWIN. O RIOMMClient executa em sistema operacional Linux e possui interface gráfica para interação do usuário. Ele é utilizado para visualização de vídeo, sincronizado com transparências. O cliente RIOMTV também executa em sistema operacional Linux e possui um tocador, mtvp, para visualização de vídeo, porém ele não faz a visualização de transparências e não permite a interação do usuário. O cliente RIOSH possui uma interface gráfica que permite a atualização e manutenção dos arquivos no servidor RIO. O cliente RIOWIN executa em sistema operacional Windows e é utilizado para visualização de vídeo e interação com o usuário. Eles foram implementados em linguagens C e C++ para sistemas operacionais diferentes (Windows e Linux), e possuem características distintas. Neste trabalho foi desenvolvido o cliente RIOWIN e o RIOMMClient foi adaptado, em cooperação com um aluno de iniciação científica do laboratório LAND [7], de forma a usar as melhorias projetadas para o sistema. Os detalhes de cada cliente serão vistos adiante.

4.2 Cliente RIOMTV

O cliente RIOMTV é utilizado para visualização de vídeo através do aplicativo mtvp [5]. Este cliente não possui uma interface para executar comandos, tais como avançar, voltar ou pausar. Sendo assim, após o início da visualização do vídeo, este é mostrado até o seu término.

Para compensar o *jitter* (variação dos tempos de chegada entre pacotes de um mesmo fluxo), o cliente RIOMTV possui um *buffer* que é preenchido antes do começo da visualização e utilizado para armazenar os blocos solicitados ao servidor. O cliente envia bloco por bloco, armazenado no seu *buffer*, ao tocador para sua visualização. O sistema é "*client driven*" e portanto a cada posição liberada, é solicitado um novo bloco ao servidor. O cliente só envia o bloco, já armazenado no seu *buffer*, ao mtvp no momento de sua exibição. A Figura 4.2 ilustra o controle do *buffer* do cliente riomtv.

Figura 4.2: Cliente RIOMTV

4.3 Cliente RIOWIN

O cliente RIOWIN foi desenvolvido em linguagem C++ na ferramenta *Microsoft Visual C++* (MVC++) [4]. Ele possui dois módulos. O módulo que faz toda a comunicação com o servidor RIO (composto pela interface) e outro que faz a comunicação com o tocador. O tocador utilizado pelo cliente é o *Windows Media Player* [10].

4.3.1 Motivação para Desenvolvimento do RIOWIN

Como no início deste trabalho já existiam os clientes que executam em sistema operacional Linux, houve a necessidade de implementar um cliente para o servidor RIO para executar no sistema operacional Windows. Apesar do cliente RIOMM-Client oferecer mais funcionalidades que o RIOMTV, ele não foi utilizado como referência no início do desenvolvimento do cliente RIOWIN por ser mais complexo e possuir as mesmas etapas que o RIOMTV para comunicação e recepção de dados do RIO. Além disso, a ferramenta utilizada para desenvolvimento (MVC++) possui módulos para construção da interface gráfica, diferente da forma implementada no cliente RIOMMClient.

A etapa inicial deste projeto foi o estudo de aplicações de visualização de vídeo em Windows. A importância deste estudo se deve ao fato do RIOMTV utilizar um tocador para sistema operacional Linux, o mtvp, e a comunicação ser feita através de um *pipe*, que é um canal de comunicação unidirecional entre dois processos. Os estudos mostraram que não era possível utilizar *pipe* no Windows pois a comunicação da aplicação com o tocador é feita de forma diferente. Foram estudadas algumas aplicações que utilizavam filtros [4] para as operações de recebimento dos dados (*source filter*), decodificação, transformação (*transform filter*), escalonamento e visualização do conteúdo de áudio, vídeo e outros tipos de dados (*renderer filter*). Os filtros são conectados através do construtor *GraphBuilder*. Este modelo não foi adotado pois o filtro lia apenas arquivos locais, e não existia um que recebesse os dados da rede.

O RIO possui uma interface de comunicação, que faz parte tanto do servidor quando do cliente. Assim, foi feito um estudo aprofundado desta interface que será vista na seção 4.3.2 e, com base no RIOMTV foram determinadas as etapas que o cliente deveria seguir e os requisitos de cada uma delas. Com isto, foi necessário migrar o código da interface para plataforma Windows. Durante este processo, foram analisadas as diferenças das bilbiotecas, funções e chamadas ao sistema operacional. Algumas bibliotecas utilizadas em Linux não eram necessárias no Windows e outras até nem existiam. As bibliotecas compatíveis com as do Linux foram pesquisadas (através da biblioteca man do Linux e de outras do Windows) e então substituídas. As funções compatíveis foram encontradas através das bibliotecas e algumas chamadas ao sistema operacional foram modificadas pois eram implementadas de forma diferente no Windows (exemplo: *pthread_join(), sleep()*).

Foram feitos também alguns testes com aplicações de visualização de vídeo existentes, como será visto na seção 4.3.3, para validar a aplicação.

4.3.2 Componentes da Interface de Comunicação

Esta interface é composta pelos componentes *SessionManager*, *StreamManager*, *ObjectManager* e *NetManager*. Cada um destes módulos gerencia uma etapa de comunicação com o RIO e provê informações ao cliente. Como foi visto no Capítulo 2, a comunicação entre o cliente e o servidor RIO é feita através de transmissões TCP e UDP. Através do protocolo TCP são enviados os pedidos e são trocadas mensagens de controle e as transmissões UDP são para o envio dos dados do/para o servidor.

Gerenciador da Sessão (*SessionManager*)

Através do gerenciador da sessão, o cliente faz a solicitação de abertura de sessão, que possuirá um identificador após sua criação. Em uma sessão o cliente pode executar as seguintes operações: listar/apagar/renomear/criar/obter informações de objetos e diretórios no servidor. O gerenciador deve sempre identificar a sessão em suas requisições. No momento de sua finalização, todos os recursos/objetos que estão sendo utilizados são liberados/fechados. Os objetos do RIO são compostos por blocos de mesmo tamanho, cujo padrão é de 128 KB. É através do gerenciador da sessão que o cliente toma conhecimento deste valor. A abertura de fluxo também é realizada nesta etapa. Para isso, as informações do tipo de tráfego (tempo real ou não), direção (escrita ou leitura) e taxa de transmissão (padrão: 1,5 Mbits/seg) são enviadas para o servidor. Caso o tipo de tráfego não seja de tempo real, o cliente não precisa informar a taxa de transmissão. O cliente pode ainda estimar a capacidade do servidor obtendo algumas informações de configuração como: número de discos, número e informações dos nós de armazenamento e o número máximo de sessões ativas.

Gerenciador de Fluxo (*StreamManager*)

O gerenciador de fluxo faz a solicitação de abertura de objeto. As informações necessárias nesta operação são: identificação do fluxo, tipo de acesso (leitura, escrita) e nome do objeto. Como a abertura de objeto só pode ser executada especificando a qual fluxo ele pertence, um cliente pode abrir mais de um objeto dentro do mesmo fluxo. Assim, as exigências de fluxo (de tempo real ou sem restrições de tempo) devem ser iguais. Cada objeto aberto possui um identificador, que é retornado pelo servidor.

Quando o cliente envia pedido de abertura de objeto, o servidor verifica se o objeto existe e se é válido. Caso seja, ele já preenche seu *playout buffer* com os blocos que o cliente irá solicitar. Para o cliente iniciar a exibição do vídeo ele faz a verificação do seu *buffer* e do *buffer* do servidor através do gerenciador de fluxo. Ele também é responsável pelo fechamento de todos os fluxos abertos, ao final da transmissão ou quando ocorre algum erro na aplicação.

Gerenciador de Objeto (*ObjectManager*)

O gerenciador de objeto provê ao cliente informações do tamanho do objeto solicitado e do número de blocos do objeto. Ao final da transmissão, o gerenciador de objeto fecha o objeto aberto.

Gerenciador da Rede (*NetManager*)

O servidor RIO possui um módulo que implementa um protocolo proprietário (RioNeti) e utiliza o UDP. Ele provê serviços tipo um controle de fluxo, detecção de erros, fragmentação e autenticação dos dados.

O gerenciador da rede é responsável por repassar as informações do endereço e da porta de comunicação do cliente, para inicialização do módulo. Ele também repassa os pedidos de blocos. Os blocos de dados do RIO são divididos em fragmentos para serem transmitidos na rede. Quando eles chegam no receptor, o módulo RioNeti é responsável por organizar os fragmentos recebidos. Quando o bloco estiver completo, o cliente é avisado pelo gerenciador da rede. Quando o cliente precisa fazer um pedido de cancelamento de bloco, esta interface é quem faz a solicitação ao módulo.

Em resumo, o gerenciador da rede é a interface de comunicação entre o módulo UDP e o cliente.

4.3.3 Funcionamento

O cliente Windows possui uma interface gráfica, conforme Figura 4.3, pela qual o usuário identifica os parâmetros de inicialização da aplicação, como nome do servidor (*Server Machine*), tamanho do *buffer* (*Buffer Size*), caminho do arquivo requisitado (*Video Name*), *login* (*User ID*) e senha de acesso ao servidor (*Password*).

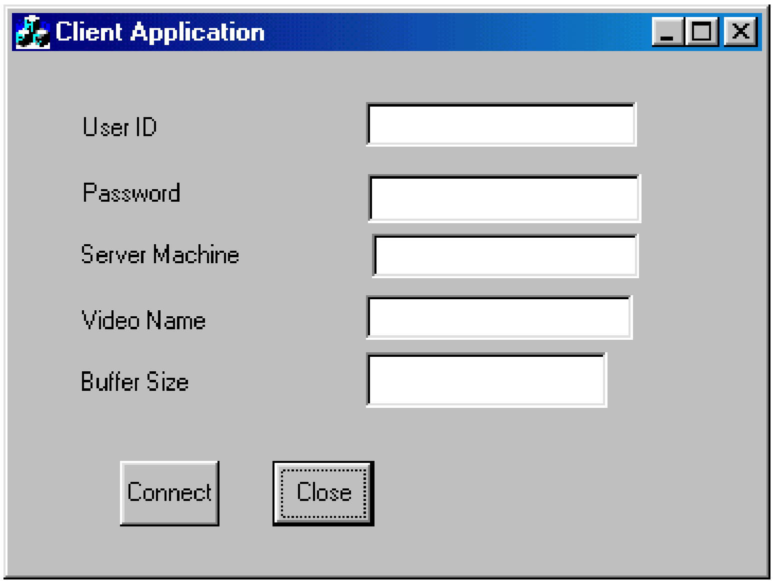

Figura 4.3: Interface Cliente Windows

Através desta interface, o cliente faz toda a inicialização do sistema. É criada uma conexão TCP onde são enviadas as requisições de abertura de sessão, fluxo (identificando o tipo de tráfego, direção e taxa de transmissão) e objeto (informando o tipo de acesso e o objeto requisitado). Alguns parâmetros adicionais, necessários em cada um dessas etapas, são configurados pelo cliente e é feito o tratamento das mensagens enviadas pelo servidor.

Se as requisições enviadas foram aceitas, o cliente inicia o pedido de blocos, já que ele possui informações do número de blocos que compõe o objeto. Ele possui um *playout buffer*, cujo tamanho é determinado pelo usuário na inicialização do sistema, para armazenar os blocos recebidos e enviá-los ao tocador. O pedido inicial do cliente é determinado pelo tamanho de seu *buffer*, ou seja, se o tamanho for igual a dois então serão requisitados inicialmente dois blocos.

Para que os blocos recebidos sejam visualizados, eles devem ser repassados ao tocador. Para selecionar o tocador que seria utilizado neste trabalho, foram testadas várias aplicações: *videolan* [3], *quicktime* [8], *realplayer* [9], *windows media player*, entre outras. Foi utilizado um programa de captura de dados que executa na plataforma Windows, o *ethereal* [2], para verificar o comportamento desses tocadores e as mensagens trocadas com a aplicação. Através de testes realizados com o servidor RIO, foi possível verificar as mensagens enviadas para estabelecimento de conexão e recepção de dados. Com isso, houve a necessidade da implementação de um servidor http simplificado. Este servidor trata as mensagens recebidas do tocador para estabelecimento de conexão e pedido de dados, e lhe envia respostas. O tocador escolhido foi o *windows media player*, por ser estável e ser amplamente utilizado pela maioria dos usuários da Internet.

Devido a comunicação com o aplicativo *windows media player* ser realizada utilizando-se o protocolo HTTP, foi desenvolvido um servidor http simplificado no cliente Windows, que cria duas *threads* durante sua inicialização. Uma *thread* que inicia o tocador, envia as requisições para o RIO e faz o armazenamento dos dados, e outra que inicia o mini-servidor http e fica esperando a conexão do tocador. Quando ele envia o pedido do objeto, o mini-servidor http trata o cabeçalho recebido e envia uma mensagem de resposta contendo o tamanho do objeto requisitado e algumas informações adicionais para controle, como a versão do protocolo http utilizado.

O cliente só inicia a visualização dos dados quando o *buffer* do servidor e o seu *playout buffer* estiverem completos. Esse tratamento foi implementado para compensar as flutuações na carga dos discos e o *jitter* da rede. Todos os dados recebidos são gravados em disco devido a duas razões: a primeira é porque o tocador pode finalizar a conexão no início da transmissão e abrir uma nova, com isso o mini-servidor http começa a enviar os dados desde o primeiro bloco; a segunda é devido as ações interativas dos usuários. Os blocos são requisitados quando uma posição do *buffer* é liberada, e o gerenciador da rede é responsável por avisar ao cliente a chegada de um bloco.

Toda a comunicação entre o cliente e o servidor RIO é feita através das interfaces vistas anteriormente, e entre ele e o tocador é gerenciada pelo mini-servidor http. A Figura 4.4 ilustra os componentes apresentados.

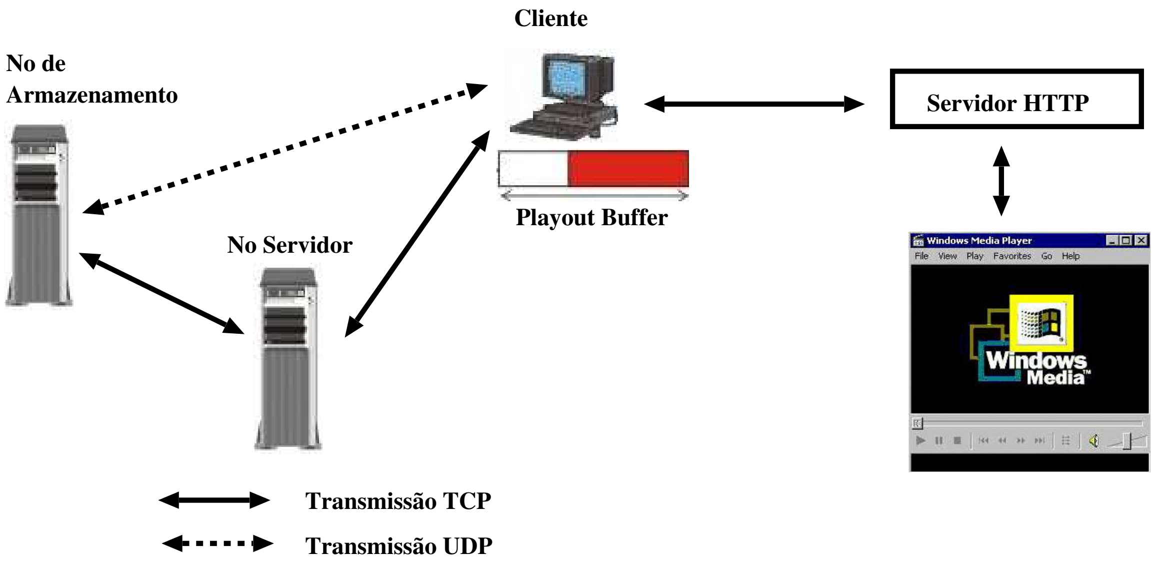

Figura 4.4: Arquitetura Cliente-Servidor

4.4 Cliente *RIOMMClient*

Este cliente foi adequado às funcionalidade adicionadas ao servidor RIO. Ele utiliza o tocador mplayer [6] para visualização dos dados, o aplicativo TGIF para visualização de transparências que são sincronizadas com o vídeo e possui uma interface gráfica para execução de comandos. Ele possui flexibilidade para escolher uma conexão direta com o servidor RIO ou utilizar o módulo de *patching* interativo implementado neste trabalho, que será visto no Capítulo 5. Como este módulo faz o envio dos dados através de transmissões *multicast*, compartilhando a banda do servidor, o cliente *riommclient* utiliza, além das interfaces já mencionadas, um novo componente para tratamento do recebimento *multicast* e de suas operações de interatividade. Os detalhes deste componente serão vistos no próximo capítulo. O cliente *riommclient* pode ser visualizado através da Figura 4.5.

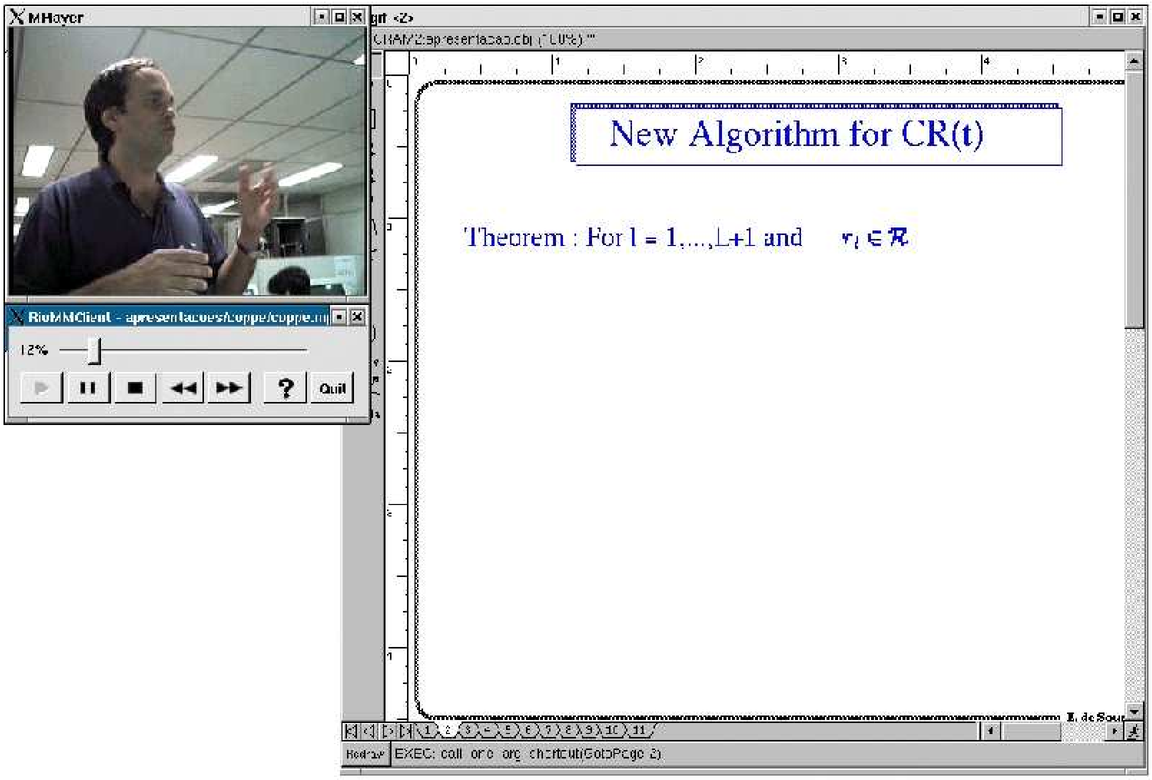

Figura 4.5: Cliente RioMMClient

4.5 Cliente RIOSH

O RIOSH foi desenvolvido para permitir a atualização e manutenção dos arquivos no servidor RIO. No desenvolvimento deste trabalho ele foi atualizado por um aluno de iniciação científica do laboratório LAND. Foi incorporada uma interface gráfica (baseada no Windows Explorer) e acrescentadas novas funções, como: transferência de arquivos simultaneamente, conexões em servidores diferentes e transferência de arquivos entre os servidores. A Figura 4.6 apresenta o cliente riosh.

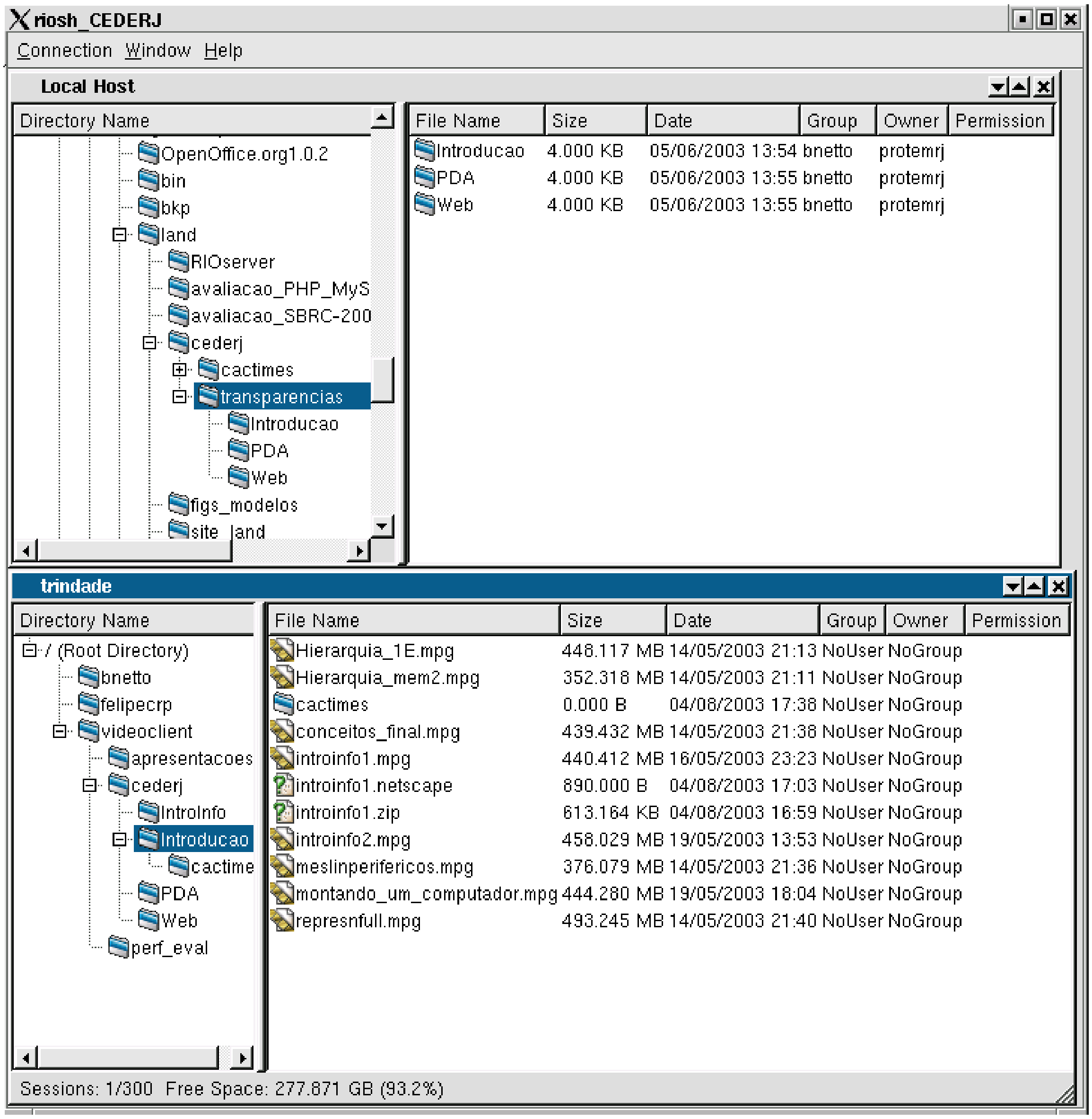

Figura 4.6: Cliente RIOSH

Capítulo 5

Módulo de *Patching* Interativo

NESTE capítulo é apresentado o mecanismo da banda de transmissão incorporado ao servidor RIO. Este mecanismo baseia-se na técnica de *patching*, e foi elaborado para manter a eficiência em aplicações de ensino a distância, com interatividade dos clientes.

Inicialmente será justificada a escolha da técnica de *patching* e feita uma breve descrição da arquitetura do servidor com o módulo implementado. Em seguida será apresentada a arquitetura do módulo de *patching* interativo, onde é descrito o seu funcionamento e de seus componentes, e ainda, a janela ótima para transmissão de novos fluxos. Os grupos *multicast* criados para o gerenciamento dos fluxos transmitidos são apresentados em 5.5. E, por fim, é apresentado o algoritmo do *Patching* Interativo e a interface de comunicação criada para o controle das ações de interatividade dos clientes com o mecanismo de *patching*.

5.1 Introdução

Conforme visto no Capítulo 2, a implementação do compartilhamento da banda é necessária para minimizar a utilização dos recursos da rede. Dentre as técnicas descritas no Capítulo 3, a técnica de *patching* foi escolhida por ter baixa latência e desempenho semelhante ao da técnica HSM, para filmes com popularidade média, e ser menos complexa que esta [20]. A popularidade dos filmes é definida em [17] e pode ser visualizada na Tabela 5.1, onde N é a taxa de chegada dos clientes.

Baixa	Média	Alta
$N < 10$	$N \geq 10$	$N \geq 100$

Tabela 5.1: Popularidade do Filmes

Um aspecto a ser observado na técnica de *patching* é a questão da interatividade dos usuários. O compartilhamento dos fluxos é realizado apenas para os clientes que "chegam" dentro da janela ativa, ou seja, quando o cliente realiza uma movimentação no vídeo para uma posição diferente da janela ativa, um novo fluxo é iniciado para este cliente. Como este trabalho focaliza as aplicações de ensino a distância, deve-se considerar a alta interatividade do usuários que acessam um servidor multimídia.

5.2 Algoritmo do *Patching* Interativo

O algoritmo proposto neste trabalho baseia-se na técnica de *patching*. A figura 5.1 ilustra o compartilhamento de um fluxo no *patching* interativo. O servidor inicia a transmissão do objeto s_1 para o cliente c_1 e dispara a janela de transmissão. Quando o cliente c_2 solicita o objeto s_1, o servidor verifica se o cliente está dentro da janela de transmissão. Se estiver, o cliente c_2 compartilha o fluxo e recebe a parte inicial do objeto que foi perdida através de uma transmissão *unicast*.

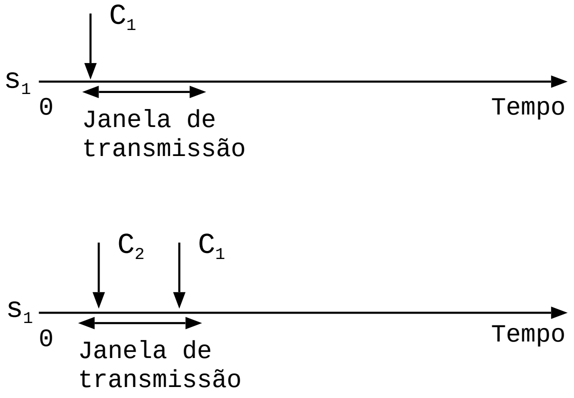

Figura 5.1: Ilustração do Compartilhamento de Fluxo no Patching Interativo

Para o tratamento da movimentação dos clientes, foram criados dois parâmetros, denominados *Delta_A* e *Delta_B* que determinam o tamanho em blocos da janela de transmissão durante as ações interativas dos clientes. Esta janela pode ser visualizada através da Figura 5.2.

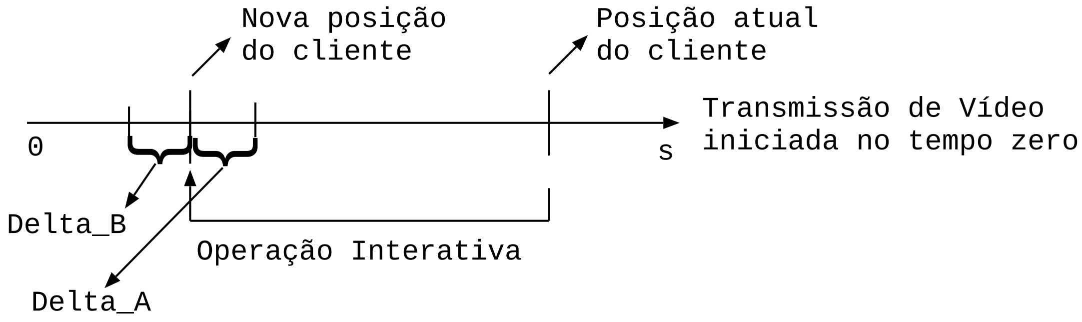

Figura 5.2: Ilustração do DeltaBefore e DeltaAfter

Quando o cliente realiza uma movimentação, é verificado se existe algum fluxo do objeto requisitado que se encaixe dentro da janela *Delta_B*. Caso não exista, a janela *Delta_A* é verificada. Se existir algum fluxo que se encaixe em uma das janelas do cliente, ele passa a compartilhar a transmissão existente. Todos os dados recebidos do fluxo *multicast* são armazenados pelo cliente.

5.2.1 Arquitetura Cliente-Servidor e o Módulo de *Patching* Interativo

A implementação do mecanismo de compartilhamento de recursos no servidor não modificou sua arquitetura, apenas adicionou uma nova funcionalidade através da criação de um módulo no servidor RIO. Este módulo é denominado *Patching* Interativo (PI). Assim, os clientes podem optar pela conexão direta ao servidor ou através do módulo PI. A Figura 5.3 ilustra a conexão de dois clientes, onde o cliente A se conecta ao servidor através do módulo PI,e o cliente B se conecta diretamente ao servidor.

Na conexão do cliente B nada foi alterado. Ele se comunica com o servidor através de uma conexão TCP, para envio de mensagens de controle e requisições.

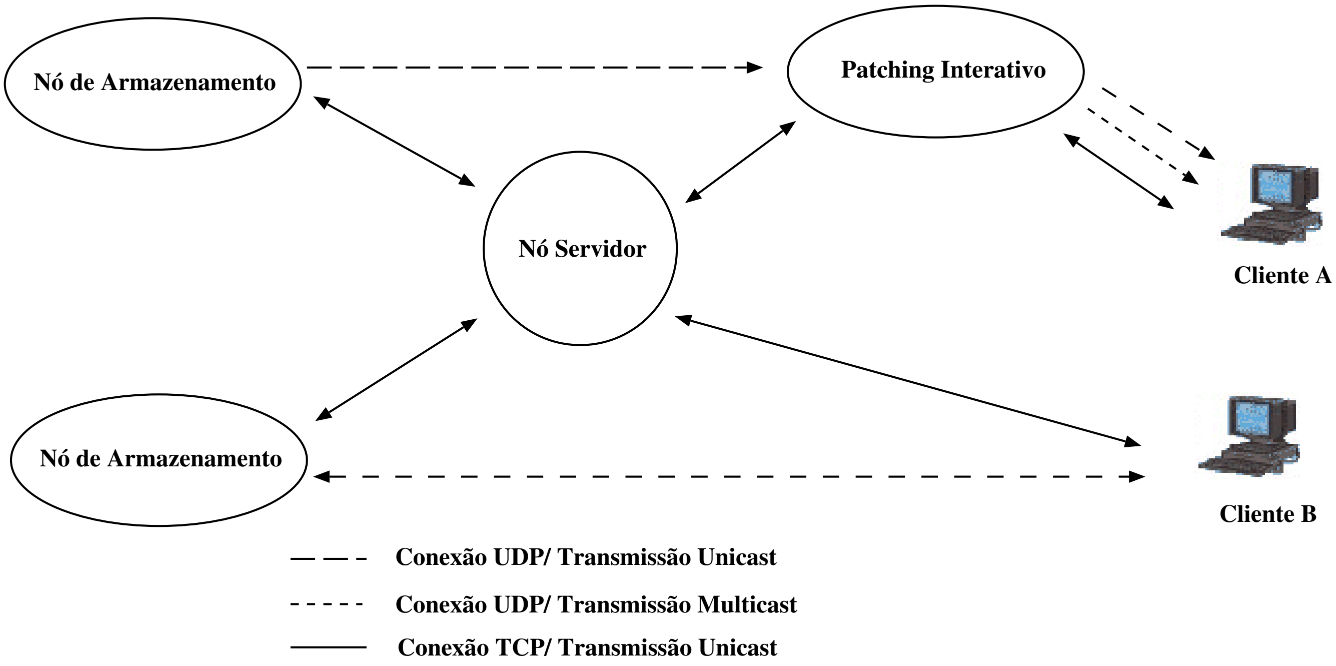

Figura 5.3: Arquitetura Cliente-Servidor e o Módulo de *Patching* Interativo

O servidor repassa os pedidos para os nós de armazenamento, que envia os dados diretamente ao cliente através de transmissões *unicast*, utilizando o protocolo UDP.

A comunicação do cliente A se diferencia da anterior pela presença do módulo PI. Este cliente estabelece uma conexão TCP com o módulo PI, para envio de mensagens e pedidos. O módulo PI se conecta ao servidor, como um novo cliente, e envia os pedidos de dados. Esses pedidos são repassados aos nós de armazenamento, que enviam os dados através de transmissões *unicast* utilizando o protoloco UDP. Os dados recebidos pelo módulo PI são enviados ao cliente através de uma transmissão *multicast*. Caso o *patching* seja necessário, os dados são enviados através de uma transmissão *unicast*, utilizando o protocolo UDP.

5.3 Arquitetura do Módulo de *Patching* Interativo

Os clientes "enxergam" o módulo PI como se ele fosse o servidor, porém ele não fornece as mesmas funcionalidades do RIO, como criar e listar diretórios e copiar arquivos para o servidor. O módulo PI foi projetado para enviar os dados que são requisitados pelos clientes, gerenciar os grupos *multicast* criados e as transmissões de *patching*. Existem quatro componentes que fazem parte deste módulo e serão descritos nesta seção: Gerenciador de Sessão (GS), Gerenciador de Fluxo (GF), Gerenciador de Objeto (GO) e a Interface de Comunicação *Multicast* (*Commulticast*). Em seguida, é decrito o funcionamento do módulo PI.

5.3.1 Visão Geral

O módulo PI e o servidor RIO podem ser executados em máquinas diferentes. Durante sua inicialização alguns parâmetros são conhecidos, como:

- *RioServer*: Máquina onde o servidor RIO está executando;

- *BuffersClient*: Tamanho do *playout buffer* de cada cliente conectado ao módulo;

- *TotalPorts*: Número total de portas *multicast* que serão usadas;

- *InicialMulticastPort*: Porta inicial da qual será somado "*TotalPorts*";

- *DeltaWindow*: Tamanho em blocos da janela do *patching*, descrita na Seção 5.4, para a união dos clientes aos fluxos existentes;

- *DeltaBefore* e *DeltaAfter*: Tamanho em blocos da janela durante as ações interativas dos clientes;

- *MulticastIP*: endereço IP que será utilizado para as transmissões *multicast*.

A cada cliente que se conecta ao módulo, é criada uma *thread* para atender seus pedidos. A Figura 5.4 ilustra o funcionamento do módulo PI.

O módulo cria um *socket* e fica aguardando a chegada de clientes. Quando um cliente se conecta é criada uma conexão TCP. Caso o cliente não consiga estabelecer uma conexão TCP com o módulo, este envia uma mensagem de aviso e finaliza a execução do cliente. Se a conexão for estabelecida, a *thread* fica aguardando a chegada de requisições do cliente. Ao receber um pedido, a *thread* verifica se pode atendê-lo. Se o pedido for inválido, a *thread* e a conexão com o cliente são finalizadas. Caso contrário, uma rotina de tratamento de requisições é executada. Esta rotina

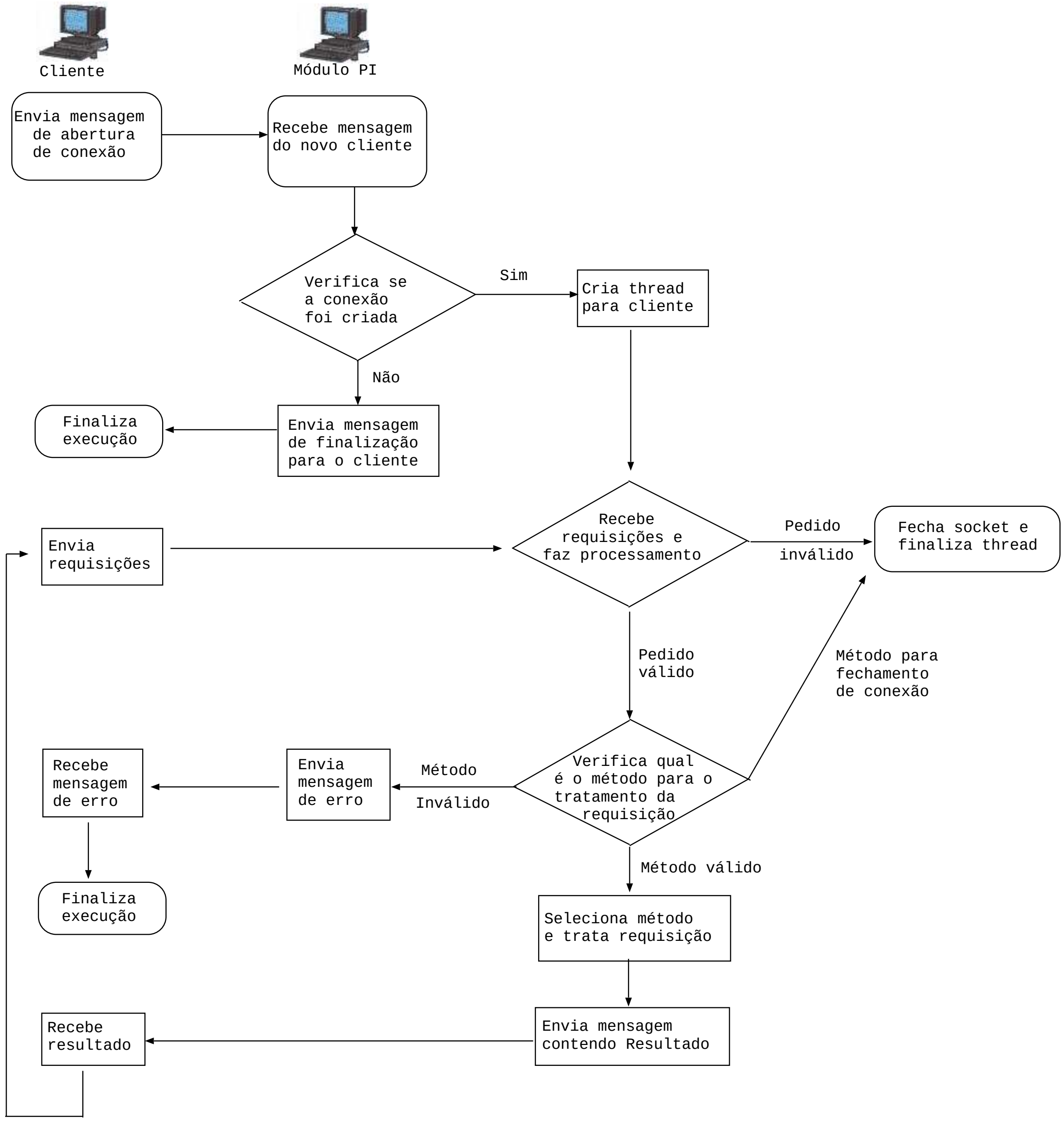

Figura 5.4: Diagrama de Execução do Módulo PI

seleciona o método correto para tratar cada pedido. Se o pedido não puder ser atendido pelo módulo PI, uma mensagem de erro é enviada ao cliente, que finaliza sua execução. Se o método selecionado for válido, o módulo PI executa a ação correspondente e envia o resultado da operação realizada. Após a execução de cada requisição, a *thread* volta ao estado de espera por novos pedidos. Quando o cliente envia uma mensagem para finalizar a conexão, o módulo PI fecha todos os objetos abertos e encerra sua conexão.

5.3.2 Rotina de Tratamento de Requisições

Os componentes do módulo PI, descritos na Seção 5.3, são responsáveis pela execução das requisições dos clientes. Os pedidos recebidos pela *thread* são repassados à rotina de tratamento de requisições. Ela faz a verificação do tipo de pedido feito pelo cliente, e o encaminha para o componente adequado. Os tipos de pedidos podem ser: abertura de sessão, fluxo, objeto e pedido de blocos de dados. Em seguida são apresentados os respectivos componentes.

Gerenciador de Sessão (GS)

Este é o componente principal do módulo PI. É ele que executa a rotina de tratamento de requisições e se comunica com os outros componentes repassando os pedidos recebidos. Ele é responsável pela abertura e fechamento de sessão com o servidor RIO, abertura de fluxo e processamento dos pedidos de bloco dos clientes. O tamanho do bloco de dados utilizado pelo servidor é enviado ao cliente na resposta de abertura de sessão. A cada pedido atendido, ele envia o resultado da operação ao cliente.

Gerenciador de Fluxo (GF)

O Gerenciador de Fluxo é responsável por atender os pedidos de fechamento de fluxo, abertura de objeto, e por verificar se o *buffer* do servidor e do módulo PI já foram preenchidos, para iniciar a transmissão dos dados para os clientes. Durante a abertura de objeto, o módulo PI possui conhecimento do objeto requisitado pelo cliente. Assim, este componente faz uma busca antecipada (*prefetching*) dos blocos iniciais desse objeto (tamanho do *playout buffer* do cliente), que serão posteriormente enviados ao cliente. Ele também é responsável por inserir o novo cliente na lista_de_clientes_conectados (que será vista na Seção 5.5), e por enviar ao cliente o endereço e a porta da transmissão *multicast*.

Gerenciador de Objeto (GO)

Este componente é responsável por enviar os pedidos de fechamento do objeto ao servidor RIO e encaminhar o resultado da operação ao componente GS.

Interface de Comunicação *Multicast*

Esta interface foi criada para troca de mensagens de controle entre o módulo PI e os clientes. As ações de interatividade realizadas pelos clientes são transmitidas ao módulo PI através desta interface, que será expandida na Seção 5.7.

5.4 Janela Ótima de Transmissão do *Patching*

A técnica de *patching* permite que os fluxos *multicast* ativos sejam compartilhados pelos clientes. Com isso, os usuários que solicitam um vídeo que já está sendo transmitido pelo servidor, podem se unir ao fluxo *multicast*, armazenando os dados enviados ao grupo, e solicitando a parte inicial da transmissão que foi perdida, chamada de *patching*. O período de tempo da transmissão *multicast* no qual podem ser iniciadas transmissões de *patching* é determinado através da janela do *patching*. O tamanho desta janela deve ser suficiente para aumentar a eficiência da técnica, minimizando o uso da banda de transmissão. A janela não pode ser muito grande senão a maioria dos canais disponíveis serão alocados para o *patching*. Porém, não pode ser muito pequena, senão o compartilhamento dos fluxos será ineficiente.

Para minimizar o uso dos recursos do servidor, deve-se determinar o tamanho ótimo da janela do *patching*. Em [22] a seguinte equação foi obtida:

$$W = \frac{\sqrt{2L_i\lambda_i + 1} - 1}{\lambda_i} \tag{5.1}$$

onde:

- L_i é o tamanho do vídeo i;

- λ_i é a taxa de chegada dos clientes.

Assume-se que a chegada dos clientes segue a distribuição Poisson, que o número de canais de transmissão é infinito, e que o vídeo será transmitido desde o início até o fim. Devido a essas restrições, a janela ótima do *patching* não pode ser utilizada durante as operações de interatividade, quando um cliente passa a compartilhar um fluxo diferente de seu atual.

5.5 Gerenciamento dos Grupos *Multicast*

Como foi visto na Seção 5.3.1, a cada cliente que se conecta ao módulo PI é criada uma *thread* para gerenciar suas requisições. Quando o cliente faz um pedido por um vídeo válido, ele é inserido em uma estrutura, chamada InfoClient. Todos os clientes que fazem parte desta estrutura são denominados por membro da *InfoClient*, e armazenam as seguintes informações:

- *ClientAddress*: identifica o endereço IP e a porta do cliente;

- *GroupId*: identificação do grupo *multicast* do cliente;

- *NextMember*: membro do mesmo grupo do cliente e sucessor a ele;

- *NextGroup*: próximo grupo de um membro da lista;

- *VideoName*: nome do vídeo requisitado pelo grupo do cliente;

- *PIDClientThread*: identificação da *thread* do cliente;

- *First*: este campo habilita o cliente a fazer pedidos se for igual a 1, e desabilita se for igual a zero;

- *MulticastPort*: cada grupo possui uma porta específica para "escutar" a transmissão *multicast*;

- *MulticastAddr*: endereço *multicast* do grupo;

- *CurrentBlock*: bloco que está sendo transmitido ao cliente, inicialmente igual a

-1.

A Figura 5.5 apresenta a estrutura *InfoClient*, e destaca algumas informações do membro A.

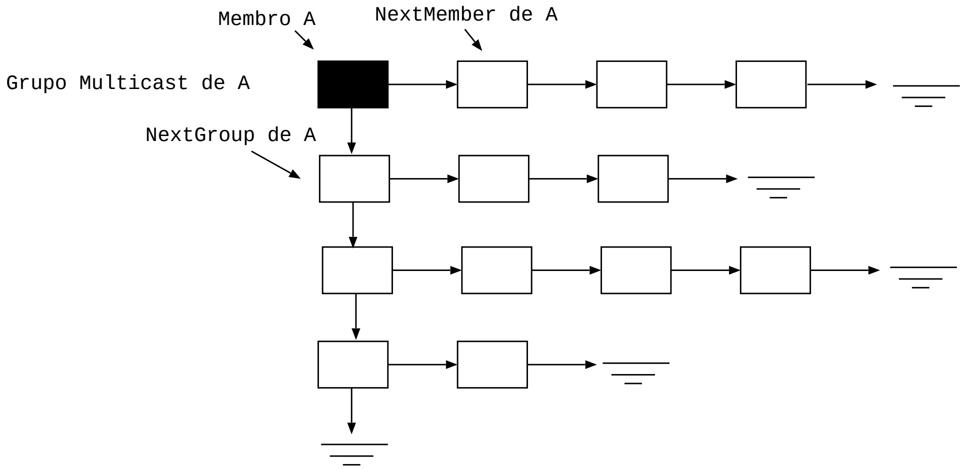

Figura 5.5: Estrutura *InfoClient*

As operações que podem ser realizadas na *InfoClient* são descritas nos tópicos seguintes. A estrutura *InfoClient* é composta por grupos. Para um membro pertencer a um grupo ele deve estar dentro da janela de transmissão do vídeo requisitado. Um grupo pode ter um ou mais membros, porém os membros devem estar sempre presentes em algum grupo. Os grupos criados são relativos ao vídeo que está sendo transmitido, ou seja, cada grupo recebe a transmissão de um único vídeo. Grupos diferentes podem estar recebendo a transmissão do mesmo vídeo. Cada vídeo sendo transmitido possui uma janela de transmissão. O módulo PI foi projetado para enviar apenas os dados que são solicitados pelos clientes, ou seja, ele é orientado a pedidos. Como todos os membros de um mesmo grupo estão recebendo os blocos através de uma transmissão *multicast*, apenas um membro deve fazer os pedidos de bloco. Por isso, cada grupo possui um membro que é denominado o líder do grupo ($First = 1$) e é responsável por enviar os pedidos ao módulo PI. Inicialmente o primeiro membro é eleito líder. As operações realizadas na *InfoClient* são: inserção, remoção e troca, e são descritas na Seção 5.6. Para realizar estas operações, o módulo PI precisa conhecer as seguintes informações: quais os grupos existentes na lista, quem é o líder de cada um deles, o tamanho da janela de transmissão dos vídeos existentes, a qual grupo pertence o membro que requisita uma operação e o endereço *multicast* das transmissões existentes. O endereço *multicast* e a porta são determinados pelo módulo PI.

5.6 Algoritmo do *Patching* Interativo

O módulo PI é responsável pelo gerenciamento de todos os clientes que se conectam ao servidor RIO através dele. Ele executa a transmissão *unicast* e a transmissão *multicast*. Quando os clientes se conectam ao módulo, suas informações são armazenadas na estrutura apresentada na Seção 5.5, que permite o gerenciamento de todos os fluxos existentes. Os clientes inseridos na *InfoClient* possuem um estado, denominado *First*, que indica se eles são líderes de algum grupo. Caso o líder do grupo se mova para outra posição da lista, um outro membro do grupo deve ser eleito líder. Quando os clientes executam uma ação como parar, pausar ou estão visualizando os dados armazenados no *buffer*, eles passam a fazer parte de uma lista de clientes inativos. O algoritmo utilizado no módulo de *Patching* Interativo é apresentado a seguir em Português Estruturado.

```
Patching_Interativo( pedido )

        se( pedido == abertura de sessão )
            Envia pedido de abertura de sessão ao servidor;
            Envia resultado ao cliente;

        se( pedido == abertura de stream )
            Envia pedido de abertura de stream ao servidor;
            Envia resultado ao cliente;

        se( pedido == abertura de objeto )
            Envia pedido de abertura de objeto ao servidor;

            //Insere cliente na estrutura InfoClient
            InsereCliente( pid, video_name, block, status );
            novo_cliente = LocalizarCliente( pthread_self() );

            SolicitaBlocos();

        se( pedido de bloco )
```

```
            cliente = LocalizaCliente();
            cliente.CurrentBlock = bloco;

            se( ( client.block != bloco )  ou ( client.BufferStatus == 0 ) )
                  SolicitaBlocos();

        se( cliente.First == 1 )
            trafego = unicast;
        senão
            trafego = multicast;

        //Envia bloco para cliente
        EnviaBloco( bloco, trafego );

Fim-Patching_Interativo
```

Este algoritmo funciona da seguinte forma: Quando um novo cliente se conecta, ele deve fazer a abertura de sessão, fluxo e objeto. O módulo PI se conecta ao servidor RIO e envia os pedidos de abertura de cada uma dessas etapas, enviando a resposta ao cliente. Durante a abertura de objeto, o cliente é inserido na *InfoClient* de acordo com o algoritmo abaixo:

```
Função Move_Cliente( cliente, janela_ativa )

       se( ( janela_ativa ) e ( posição_grupo[ janela_ativa ] > posição_de_movimento ) )
             Insere Cliente no grupo com a janela_ativa;
       senão
             // Verifica se o cliente pode entrar em algum outro grupo
             i = 0;
             grupo_delta_before     = -1;
             grupo_delta_after      = -1;
             distancia_delta_before = INFINITO;
             distancia_delta_after  = INFINITO;

             enquanto existir algum fluxo para o mesmo vídeo do cliente

                se( ( ( posição_de_movimento - DELTA_BEFORE ) <= posição_grupo[ i ] ) e
                    ( posição_grupo[ i ] <= posição_de_movimento ) )

                        se( ( posição_de_movimento - posição_grupo[ i ] ) < distancia_delta_before )
                              distancia_delta_before = posição_de_movimento - posição_grupo[ i ];
                              grupo_delta_before = i;
                senão
                        se( ( posição_de_movimento <= posição_grupo[ i ] ) e
                            ( posição_grupo[ i ] <= ( posição_de_movimento + DELTA_AFTER ) ) )
```

```
                        se( ( posição_grupo[ i ] - posição_de_movimento ) < distancia_delta_after )
                            distancia_delta_after = posição_grupo[ i ] - posição_de_movimento;
                            grupo_delta_after = i;
                incrementa valor de i;

        // Verificando se foi encontrado algum grupo no DeltaBefore
        se( grupo_delta_before != -1 )
            Insere Cliente no grupo com grupo_delta_before;
        senão
            // Verificando se foi encontrado algum grupo no DeltaAfter
            se( grupo_delta_after != -1 )
                Insere Cliente no grupo com grupo_delta_after;
            senão
                // Não achou nem um grupo para o cliente
                Cria novo grupo para o cliente;

Fim-Move_Cliente
```

Para determinar se o cliente fará parte de algum grupo, deve-se verificar se o cliente está dentro da janela de transmissão. Cada transmissão de um vídeo possui uma janela que habilita um novo cliente a compartilhar o fluxo. A janela é calculada de acordo com a Fórmula 5.1. Se o cliente estiver dentro da janela, ele é inserido no grupo existente, senão verifica-se todos os grupos existentes que transmitem o vídeo solicitado pelo cliente. Para habilitar o cliente a "escutar" a transmissão de algum grupo, deve-se comparar a posição do cliente em relação a posição do grupo. Os valores de delta_before e delta_after são utilizados para determinar a janela de cada grupo.

Após a inserção do cliente na *InfoClient*, o módulo PI inicia os pedidos dos blocos iniciais do objeto ao servidor. Os blocos são armazenados em um *buffer* para posterior envio ao cliente. As informações da transmissão *multicast* (endereço, porta e informação de líder) são enviadas ao cliente através da interface de comunicação *multicast*, que será apresentada na Seção 5.7.

Quando o cliente é inserido em um fluxo já existente, ele passa a receber os dados que estão sendo enviados para o grupo. Isso significa que ele deverá soliciar ao módulo PI os dados iniciais da transmissão que foram perdidos. Ele inicia o pedido de blocos do *patching* até alcançar os dados já recebidos do fluxo *multicast*, ou seja, os blocos armazenados em seu *buffer*. O *patching* é enviado através de uma transmissão *unicast*.

Quando o módulo PI recebe um pedido de bloco, ele verifica qual o cliente que enviou o pedido e se o bloco requisitado já está presente em seu *buffer*. Se não estiver, o pedido do bloco é enviado ao servidor. Se o cliente for líder do grupo, o módulo PI envia o bloco para o endereço *multicast* do grupo, senão ele envia diretamente para o cliente, através de uma transmissão *unicast*. Todos os dados recebidos pelos clientes são armazenados em disco. Os detalhes dessa necessidade serão vistos na Seção 5.7, que trata a questão da interatividade.

A operação de remoção de um cliente da lista será apresentada em 5.6.1, e a operação de alteração está descrita na Seção 5.7.

5.6.1 Remoção da lista

Quando um cliente finaliza a conexão, ele é removido da *InfoClient*, que é atualizada. O módulo PI verifica se o cliente é o líder de seu grupo. Se for, e existir outro membro no grupo, esse será eleito líder e será enviada uma mensagem para avisá-lo de sua condição. Caso o cliente não seja líder, então o cliente é removido e a lista é atualizada. Se o cliente for o único membro do grupo, ele é removido e o grupo é finalizado. A Figura 5.6 ilustra este comportamento.

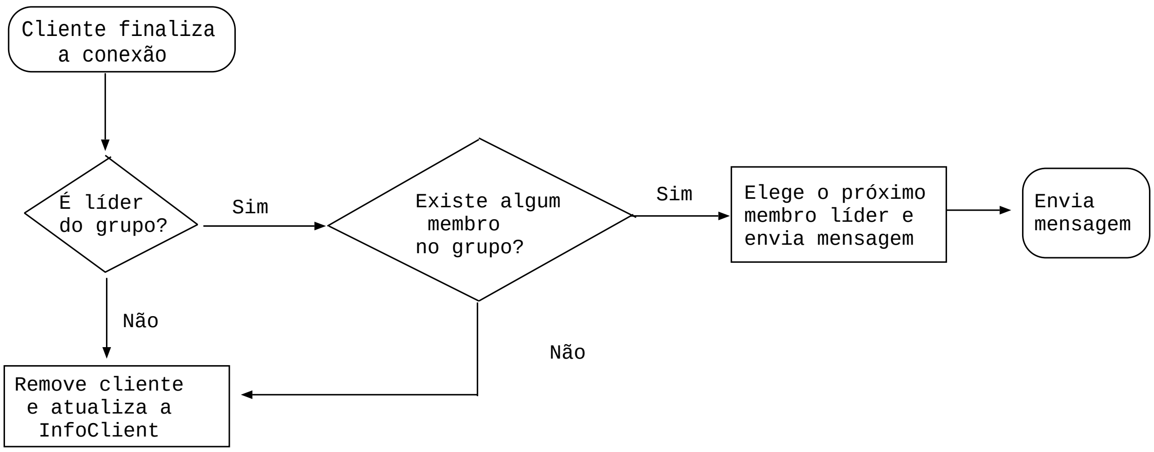

Figura 5.6: Remoção de um Cliente da Lista

5.7 Tratamento das Operações de Interatividade

Como este trabalho foi desenvolvido dando enfoque às aplicações de ensino a distância, as ações de interatividade dos clientes foram tratadas para garantir a qualidade de serviço da aplicação. Para isto, o componente Interface de Comunicação *Multicast* (*Commulticast*) foi criado para troca de mensagens de controle entre os clientes a o módulo PI durante as ações interativas. Estas mensagens são transmitidas utilizando-se o protocolo TCP, e são definidas da seguinte forma:

- msgPID: esta mensagem é enviada pelo cliente e indica o seu PID. Através dela, o módulo PI verifica as informações do campo *first*, do endereço e da porta *multicast* da transmissão do cliente, e repassa essas informações para que ele possa "escutar" o fluxo. Os parâmetros desta mensagem são:

  - *code*: identifica o tipo da mensagem (msgPID);

  - *pid*: identifica o *pid* do cliente.

- msgIP: esta mensagem é enviada pelo módulo PI indicando as informações do fluxo *multicast*. Ela possui os seguintes parâmetros:

  - *code*: identifica o tipo da mensagem (msgIP);

  - *address*: identifica o endereço IP da transmissão *multicast*;

  - *port*: identifica a porta que o cliente deverá "escutar" a transmissão;

  - *first*: identifica se o cliente deve fazer pedidos de bloco;

  - *block*: indica o bloco que o cliente deverá tocar. Este parâmetro é válido apenas quando o cliente se movimenta e é inserido em um grupo existente.

- msgMove: determina o bloco solicitado pelo cliente ao módulo PI. Este pedido é enviado por um cliente, que não seja líder de algum grupo, que esteja realizando uma operação de interatividade como pausar, parar, avançar, retroceder ou tocar. O líder de um grupo não precisa enviar esta mensagem caso as operações realizadas sejam avanço ou retrocesso no vídeo. Sua operação de interatividade é detectada através do pedido de bloco. Esta mensagem é composta pelos seguintes campos:

- *code*: identifica o tipo da mensagem (msgMove);
- *block*: identifica o bloco solicitado;
- *action*: identifica a operação realizada pelo cliente.

A estrutura *InfoClient* permite ao módulo PI ter uma visão global do sistema. Através da *Commulticast*, o cliente envia ao módulo as mensagens que indicam as operações de interatividade realizadas. Dessa forma, o módulo organiza a *InfoClient* de acordo com a nova condição do cliente, mantendo a qualidade de serviço da aplicação. Como foi visto na Seção 5.3.1, o módulo PI possui um *buffer* para cada cliente, onde são armazenados os dados a serem solicitados. Quando o usuário realiza uma ação interativa, que gere um pedido de bloco que não está presente em seu *buffer* no módulo PI, ele realiza um *refresh* no *buffer*. A operação de *refresh* verifica se existe algum bloco no *buffer* que será utilizado no futuro próximo, e envia ao servidor as requisições dos blocos necessários. A janela de transmissão utilizada durante a movimentação dos clientes é determinada pelos deltas da Função Move_Cliente descrita na Seção 5.6. Existem dois deltas a serem calculados: *DeltaBefore* e *DeltaAfter*. O *DeltaBefore* determina uma janela de transmissão anterior a nova posição do cliente medida em um número de blocos fixos. O *DeltaAfter* determina a janela posterior a novo posição do cliente. Assim, quando o cliente se move para uma nova posição do vídeo, e não existe nenhuma janela ativa para compartilhar o fluxo, deve-se determinar se existe algum outro fluxo do mesmo vídeo que se encaixe dentro da janela do *DeltaBefore*. Se não existir, a outra janela a ser utilizada é o *DeltaAfter*. Caso o cliente não se encaixe dentro da janela de nenhum fluxo, um novo grupo é criado para o cliente. A Figura 5.2 ilustra os deltas citados anteriormente.

As operações de interatividade que serão apresentadas nessa seção são: pausar, parar, tocar, avançar e retroceder. Todas as operações geram uma procura na *InfoClient* para modificar a situação do cliente e manter o desempenho do sistema.

5.7.1 Operações de Pausar e Parar

Apesar da operação de pausa ser diferente da operação de parada, o novo estado do cliente é o mesmo pois ele não estará visualizando os dados. Quando um cliente realiza uma destas operações, ele envia uma mensagem ao módulo PI indicando a ação realizada. Com isso, o cliente é removido do grupo ao qual pertence, e é inserido em uma lista que contém todos os clientes inativos, chamada de Lista de Clientes Inativos. A Figura 5.7 apresenta este esquema.

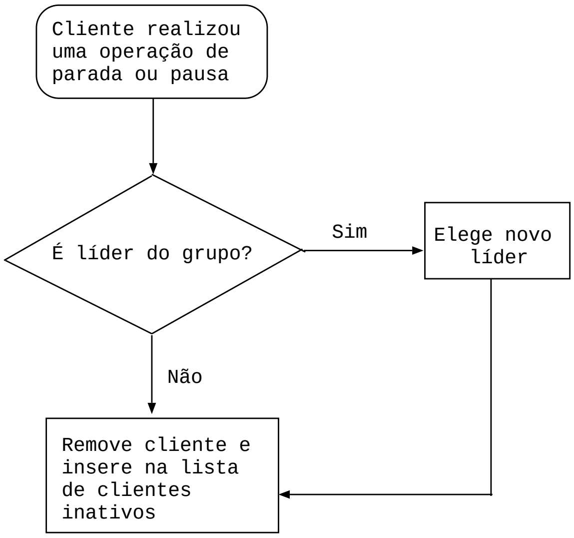

Figura 5.7: Operação de Parada e Pausa

5.7.2 Operação de Tocar - *Play*

Uma operação de tocar é realizada para iniciar a visualização do vídeo, ou após as operações de parada e pausa. No primeiro caso, será realizada a operação de inserção descrita na Seção 5.6. Todos os dados recebidos pelo cliente são armazenados em disco.

Já no segundo caso, o cliente sempre verifica se já possui os dados armazenados. Se possuir, ele irá "consumir" os dados do *buffer* e continuará na lista de clientes inativos. Caso não possua os dados armazenados, ele deve enviar uma mensagem ao módulo PI inciando que sua nova situação, ou seja, uma mensagem do tipo msgMove. Ao receber a mensagem do cliente, o módulo PI executa a função Move_Cliente para determinar qual a nova posição do cliente na *InfoClient*. Assim, o endereço *multicast* da transmissão é enviado ao cliente.

O cálculo da janela de transmissão dos fluxos durante as operações de interatividade é determinado através dos parâmetros *Delta_B* e *Delta_A*. Sempre que um cliente for movido de grupo, o endereço *multicast* do novo grupo que ele passará a "escutar" será enviando pelo módulo PI. A operação de tocar é descrita na Figura 5.8.

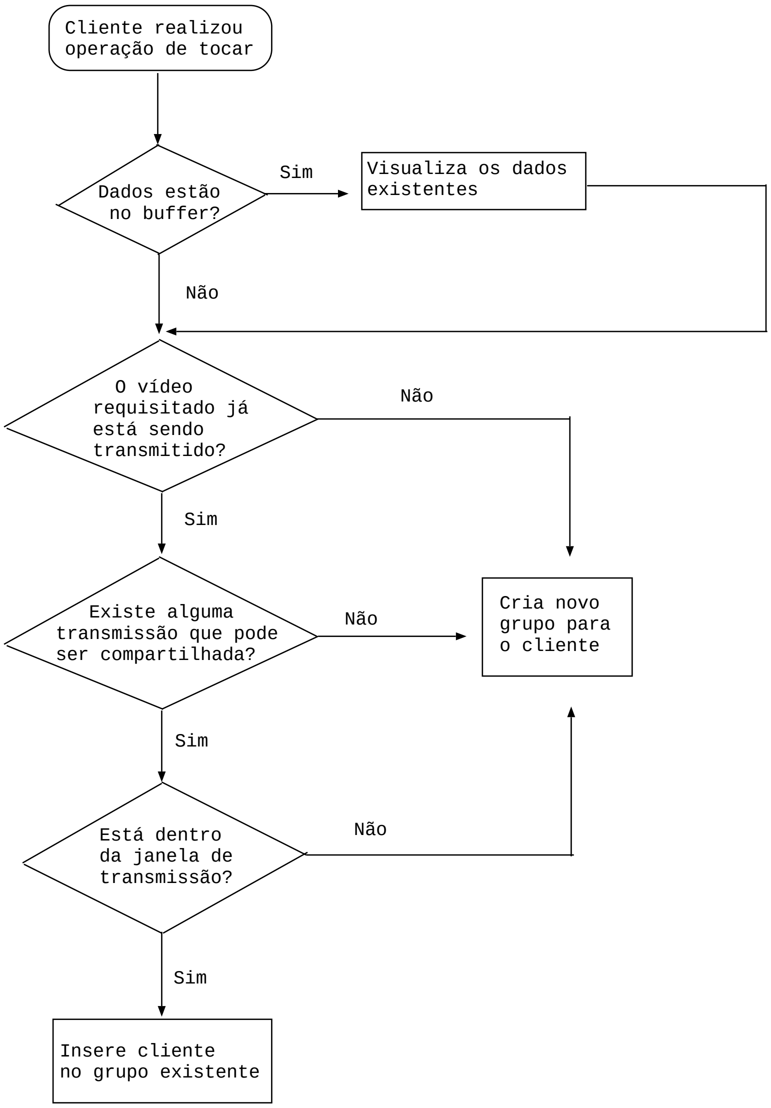

Figura 5.8: Operação de Tocar

5.7.3 Operações de Avançar e Retroceder

Quando o cliente realiza essas operações, os dados a serem visualizados podem estar presentes em seu *buffer*. Como o cliente não estará mais visualizando os dados recebidos pelo seu grupo, ele deve informar a ação realizada ao módulo PI. A mensagem enviada, msgMove, irá conter a operação realizada e o bloco que o cliente necessita, caso os dados não estejam presentes em seu *buffer*. Se o cliente for "consumir" os dados armazenados, ele é retirado do grupo ao qual pertence e é inserido na lista de clientes inativos.

Ao receber a mensagem do cliente, o módulo PI executa a função Move_Cliente, descrita na Seção 5.6 para determinar seu novo grupo. Caso seja criado um grupo para o cliente, o módulo PI realiza um *refresh* em seu *buffer*, e solicita os dados necessários ao servidor RIO. Porém, se o cliente estiver dentro da janela de transmissão de algum grupo, ele passará a fazer parte dele. Através da mensagem enviada pelo módulo PI, msgIP, o cliente é informado do novo endereço *multicast* da transmissão, juntamente com a informação de líder.

Se o cliente estiver "escutando" a transmissão existente e no momento de tocar um bloco ele não estiver presente em seu *buffer*, ele deverá pedir um *patching* conforme Figura 5.9. A janela de transmissão nesta operação é determinada pelo cálculo do tamanho dos deltas.

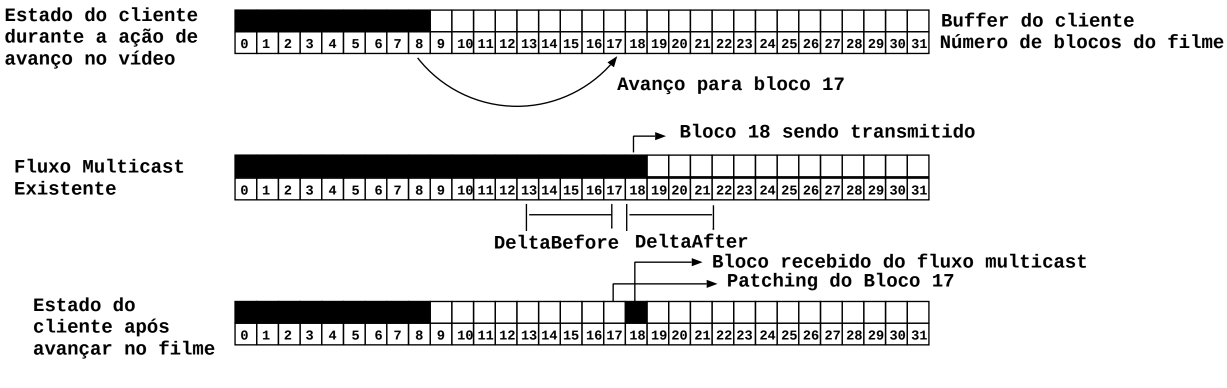

Figura 5.9: Pedido de *Patching* Durante uma Operação Interativa

5.8 Resumo

Este capítulo apresentou o algoritmo de *patching* interativo utilizado na implementação do módulo PI para o compartilhamento da banda de transmissão do servidor RIO. A idéia geral da arquitetura do servidor com o módulo PI foi apresentada na Seção 5.1. Com a introdução do módulo de compartilhamento, algumas características do servidor RIO foram modificadas. O envio dos dados era feito através de transmissões *unicast*, onde todos os clientes deveriam enviar os pedidos de blocos. Os blocos passaram a ser enviados através de transmissões *multicast*, diminuindo a quantidade de mensagens enviadas pelos clientes para pedidos de dados. Não existia a infra-estrutura que gerencia os fluxos sendo transmitidos (InfoClient), e permite o compartilhamento das transmissões, mesmo durante operações de interatividade dos clientes. Os clientes passaram a armazenar todos os dados recebidos, o que permite manter a continuidade na visualização do vídeo e diminuir a quantidade de dados enviados pelo servidor durante as ações interativas. Todas as operações realizadas pelos usuários são tratadas baseadas na estrutura *InfoClient*, que permite uma visão global do sistema.

As características adicionadas ao RIO permitem a diminuição da carga gerada pelos clientes no servidor, e mantêm a qualidade de serviço do sistema. A Figura 5.3 ilustra o módulo desenvolvido e a flexibilidade de sua utilização. A eficiência desta implementação é analisada no Capítulo 6.

Capítulo 6

Medidas e Resultados

NESTE capítulo serão apresentados os resultados obtidos dos testes realizados com o emulador e com o modelo. O emulador simula o comportamento de vários clientes durante a transmissão de um vídeo. O modelo representa o módulo de *patching* interativo adicionado ao servidor RIO. Eles foram criados para análise de desempenho da técnica implementada para aumentar as funcionalidade do servidor multimídia.

6.1 Experimentos

Este trabalho foi desenvolvido visando as aplicações de ensino a distância, onde os clientes que estão assistindo ao vídeo de uma aula possuem alta interatividade com o sistema. Este comportamento diferenciado, pois o vídeo não é assistido de forma seqüencial, foi tratado durante a implementação do módulo de *patching* interativo, explicado no Capítulo 5. Para analisar o desempenho do módulo implementado, foi desenvolvido um emulador que executa as ações interativas do cliente durante a transmissão de um filme.

Através dos testes realizados pode-se obter o número de fluxos ativos e inativos no sistema e a quantidade de fluxos *multicast* e de *patching* solicitados, variando-se a quantidade de clientes, e os valores de Delta_Before e Delta_After, apresentados na Figura 5.2 do capítulo anterior, e mantendo-se o tamanho da janela ótima do *patching* vista na Seção 5.4 do Capítulo 5.

6.2 Ambiente de Teste

Para realização dos experimentos, foram utilizadas duas máquinas servidoras de discos e uma máquina para execução do nó servidor. Os clientes foram disparados em três outras máquinas. A escolha da máquina a ser utilizada para execução de cada cliente foi aleatória. A Figura 6.1 apresenta o ambiente de testes. As máquinas utilizadas possuem sistema operacional Linux *Red Hat* 7.2, (Kernel versão 2.4.7) e estão conectadas a um *switch* de 100 Mbps em uma rede de 100 Mbps.

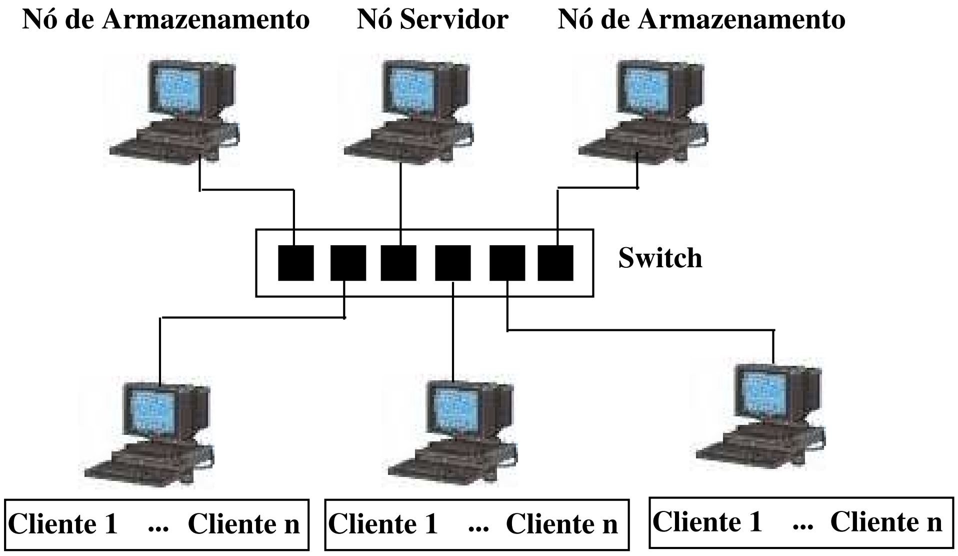

Figura 6.1: Ambiente de Testes

6.3 Obtenção do Logs

Os arquivos de *log* utilizados nos experimentos foram obtidos através do modelo que descreve o comportamento dos clientes durante uma aula de TCP/IP. Este modelo foi desenvolvido no ambiente de modelagem da ferramenta TANGRAM-II e é composto por dez estados, onde oito representam alguma transparência da aula e os outros dois descrevem ações de *pause*, a probabilidade dos clientes estarem em cada um deles é variável. O arquivo de saída do modelo representa o comportamento de cada cliente da simulação, ou seja, os estados visitados pelo cliente. A duração da aula foi de uma hora e quarenta minutos. Para gerar o *log* a ser utilizado pelo emulador, cada estado foi representado por um bloco do filme, contendo o tempo de permanência do cliente no respectivo estado. O filme utilizado para o teste foi o Imagens_Mapas_Menor.mpg e possui tempo de duração de uma hora e dezoito minutos. Este filme é uma aula do CEDERJ, e foi escolhido por ter o tempo de duração mais próximo da duração do modelo.

6.3.1 Extração do Tempo de Duração dos Blocos do Filme

Para obter a duração de cada bloco foi utilizado um programa em linguagem C que lê os blocos de dados do filme (tamanho do bloco igual a 128K) e envia ao aplicativo *mplayer* responsável pela exibição da mídia. Desta forma, a cada bloco visualizado seu tempo de exibição é coletado e gravado em um arquivo de saída. Ao final, obtem-se o tempo de duração de todos os blocos do filme.

6.3.2 Formato do *Logs*

O *log* obtido do modelo representa o comportamento de todos os clientes da simulação, ordenados pelo tempo da requisição. Foram realizados experimentos com 100 e 200 clientes. Para o emulador utilizar o *log* do modelo, foi implementado um programa em linguagem awk **??** que gera um arquivo para cada cliente. Assim, quando o cliente é iniciado, o emulador executa as ações contidas no *log* deste cliente. Os campos que compõe o arquivo de *log* são:

- identificação da sessão: cada cliente da simulação é identificado por um valor que corresponde a sua sessão;
- tempo de disparo da ação: este campo é gerado em segundos e especifica o tempo que a ação do cliente é disparada;

- ação a ser realizada: identifica uma ação interativa realizada pelo cliente;
- bloco: este campo identifica o bloco que será solicitado pelo cliente. Nas operações de abrir e parar este campo possui valor igual a -1.

6.4 Integração do Emulador com o Cliente

Para utilizar o emulador, o cliente *RIOMMClient* passou a inicializar com um novo parâmetro (–e <*filename*>), que indica que a sua execução não é interativa, ou seja, as requisições enviadas ao servidor serão obtidas do *log* gerado.

Para disparar vários clientes foi gerado um programa, implementado em linguagem awk, que lê o tempo de início de cada cliente, escolhe aleatoriamente uma máquina, e dispara o cliente no tempo lido do arquivo. A espera entre a inicialização dos clientes é realizada através da chamada de sistema *sleep*. O cliente iniciado, estabelece uma conexão com o módulo PI, e faz o pedido dos blocos iniciais. O número de blocos solicitados é igual ao tamanho do *buffer* do cliente. Após o preenchimento de todo o *buffer*, o cliente detecta que a sessão será simulada, e executa o emulador. Este emulador foi denominado *RioMMEmul*, e utiliza a interface gráfica do *RIOMMClient* para execução das operações a serem realizadas. A cada nova operação, o cliente atualiza o bloco, e envia uma mensagem ao módulo PI indicando a ação realizada. Após receber as informações de endereço e porta do fluxo *multicast*, o cliente atualiza seus dados e continua sua execução. O intervalo de tempo entre as operações VCR é calculado através da subtração entre o tempo da próxima operação e o tempo da operação atual.

A cada operação realizada pelos clientes, o módulo PI atualiza a estrutura *InfoClient*, apresentada no Capítulo 5, e grava as informações dos fluxos. O tamanho do *buffer* de cada cliente no módulo é de cinco posições.

Para a realização dos testes, o cliente *RIOMMClient* não faz a visualização dos dados recebidos do módulo PI, ou seja, o tocador *mplayer* não é iniciado devido ao alto consumo de processamento. Para simular a exibição dos blocos recebidos do módulo PI, o cliente executa a chamada de sistema *sleep* e passa como parâmetro o tempo de duração do bloco que é obtido do arquivo gerado visto na Seção 6.3.1.

6.5 Resultados dos experimentos

Os resultados obtidos apresentam o número de canais alocados para os fluxos *multicast* durante a transmissão da mídia. Foram feitas variações do valor do Delta_After, mantendo-se constante o Delta_Before pois seu valor não deve ser muito grande. Para realizar os experimentos foram considerados dois cenários, o primeiro com 100 clientes e o segundo com 200 clientes. Os valores de Delta_Before, Delta_After e da janela ótima foram calculados em blocos.

A Tabela 6.1 apresenta os valores utilizados.

Delta_Before	Delta_After	Janela Ótima	Taxa de Chegada
56	2828	662	1 cliente/minuto
56	744	662	1 cliente/minuto
56	339	662	1 cliente/minuto
56	56	662	1 cliente/minuto

Tabela 6.1: Parâmetros utilizados nos experimentos

6.5.1 Resultado dos experimentos com 100 Clientes

Nesta seção serão apresentados os resultados obtidos da simulação de 100 clientes. Os parâmetros utilizados nos experimentos estão descritos na Tabela 6.1. Antes de sua realização, foram feitas algumas validações do módulo PI com um número menor de clientes. Os testes realizados para validar a aplicação descrevem as operações de VCR e o compartilhamento dos fluxos através do Delta_After, Delta_Before e da janela ótima do *patching*. Durante sua realização, os problemas apresentados foram corrigidos. Os experimentos foram realizados no laboratório LAND **??**. O ambiente de testes foi apresentado na Seção 6.2.

6.5.2 Resultado dos experimentos com 200 Clientes

Capítulo 7

Conclusão

Como o uso das aplicações de mídia contínua vêm crescendo ao longo do tempo, é necessário que os recursos da rede sejam utilizados de forma eficiente para prover QoS satisfatória aos usuários. Devido às restrições de tempo dessas aplicações, a qualidade da apresentação pode ser comprometida se houver atraso na entrega dos dados. Outro fator importante das aplicações de tempo real é a alta interatividade dos clientes. Este comportamente deve ser tratado para não comprometer a QoS oferecida. O atraso na entrega dos dados pode ser inserido pelo processamento da requisição no servidor multimídia ou durante a transmissão na rede. Como os servidores multimídia utilizam transmissões *unicast* para envio dos dados, a largura de banda disponível da rede não é utilizada de forma eficaz. Para minimizar o uso da rede, os dados transmitidos para os clientes que requisitam a mesma mídia, podem ser enviados através de transmissões *multicast*.

Como foi apresentado no Capítulo 3, várias técnicas foram estudadas para compartilhar os recursos da rede utilizando transmissões *multicast*. A técnica de *patching* foi escolhida por oferecer serviço imediato aos clientes, ser menos complexa que as outras e possuir desempenho semelhante a algumas delas.

Neste trabalho, a técnica de *patching* foi implementada para minimizar o uso da banda do servidor, e estendida para atender as funcionalidades de interação dos usuários durante a transmissão da mídia. Foram definidos métodos para o tratamento deste comportamento, de forma que fosse mantida a QoS oferecida aos clientes. Foi implementado também um cliente para o servidor RIO que executa em sistema operacional Windows, como objetivo de expandir a utilização do servidor para outro sistema operacional.

Esta funcionalidade foi adicionada ao servidor RIO, mantendo a interface e os módulo existentes. Para isso, foi desenvolvida uma camada, a nível de aplicação, que implementa o *Patching* Interativo (PI). Com isso, os clientes possuem a flexibilidade de conexão direta com o servidor, ou através do módulo PI compartilhando o fluxo de dados.

Também foram desenvolvidos um emulador, que foi incorporado ao cliente, e um modelo da técnica implementada. O emulador visa a simulação de cargas reais de servidores de vídeo operacionais no servidor RIO, para testar a eficácia da técnica implementada. O modelo descreve o comportamento da técnica de *patching* durante a transmissão de vídeos para vários clientes diante as ações de interatividade destes. Os resultados obtidos do emulador e do modelo foram analisadas e comparados no Capítulo 6.

Como este trabalho faz parte de um projeto de ensino a distância, o servidor RIO será utilizado para armazenamento e distribuição das aulas gravadas dos cursos, que serão oferecidos pelo Centro de Educação a Distância do Estado do Rio de Janeiro (CEDERJ). Durante a exibição do vídeo das aulas, o material didático é visualizado através de *slides*. Os alunos poderão interagir com o vídeo através da interface do cliente RIO. O vídeo e os *slides* são sincronizados garantindo a qualidade da apresentação.

7.1 Trabalhos Fututos

Algumas linhas de atuação podem ser acrescentadas a este trabalho, como:

- Transmissão *multicast* dos fluxos de *patching*, para que estes também possam ser compartilhados;

- Determinação da janela ótima durante as operações de interatividade dos clientes, sem admitir que o vídeo será assistido desde o início até o final;

- Um melhoria que pode ser implementada no servidor RIO é o tratamento de perdas de dados. Em caso de muitos blocos sucessivos serem perdidos, o cliente RIO pode paralisar a exibição dos dados. Devido a esse problema, faz-se necessária a implementação de um algoritmo de recuperação dos dados na interface de recebimento do cliente.

Referências Bibliográficas

[1] Centro de Educação de Ensino Superior a Distância do Estado do Rio de Janeiro. URL http://www.cederj.rj.gov.br.

[2] Free Network protocol analyser for Unix and Windows. URL http://www.ethereal.com.

[3] Free Software and Open Source video streaming solution for every OS. URL http://www.videolan.org.

[4] Microsoft Visual Studio C++. URL http://msdn.microsoft.com/visual C.

[5] MPEGTV mtvp-MPEG Movie Player and VCP Player for Linux. URL http://www.mpegtv.com.

[6] MPlayer for Linux. URL http://www.mplayerhq.hu/homepage/design 6/news.html.

[7] Site do LAND. URL http://www.land.ufrj.br.

[8] Site do Quicktime. URL http://www.apple.com/quicktime.

[9] Site do RealPlayer. URL http://www.real.com.

[10] Windows Media Player. URL http://windowsmedia.com/download.

[11] Aggarwal, C.,C., Wolf, J., L., and Yu., P., S. A permutation-based pyramid broadcasting scheme for video-on-demand systems. In *Proc. of IEEE International Conference on Multimedia Computing and Systems* (Jun 1996).

[12] Almeroth, K. C., e Ammar, M. H. The use of multicast delivery to provide a scalable and interactive video-on-demand service. *IEEE Journal of Selected Areas in Communications 14*, 6 (1996), 1110–1122.

[13] Cai, Y., Hua, K., e Vu, K. Optimizing patching performance.

[14] Cardoso. A. Q. Mecanismos para garantir qualidade de serviço de aplicações de vídeo sob demanda, 2002. Tese de Mestrado, UFRJ.

[15] Carter, S., e Long, D. Improving video-on-demand server efficiency through stream tapping, 1997.

[16] Dan, A., Sitaram, D., e Shahabuddin, P. Scheduling policies for an on-demand video server with batching. In *ACM Multimedia* (1994), pp. 15–23.

[17] Eager, D., Vernon, M., e Zahorjan, J. Bandwidth skimming: A technique for cost-effective video-on-demand. Relatório Técnico CS-TR-1999-1408, 1999.

[18] Eager, D. L., e Vernon, M. K. Dynamic skyscraper broadcasts for video-on-demand. *Lecture Notes in Computer Science 1508* (1998), 18–??

[19] Eager, D. L., Vernon, M. K., e Zahorjan, J. Optimal and efficient merging schedules for video-on-demand servers. In *ACM Multimedia (1)* (1999), pp. 199–202.

[20] Eager, D. L., Vernon, M. K., e Zahorjan, J. Minimizing bandwidth requirements for on-demand data delivery. *Knowledge and Data Engineering 13*, 5 (2001), 742–757.

[21] Edmundo de Souza e Silva, Rosa M. M. Leão, Berthier Ribeiro-Neto and Sérgio Campos. Performance Issues of Multimedia Applications, 2002.

[22] Gao, L., e Towsley, D. F. Supplying instantaneous video-on-demand services using controlled multicast. In *ICMCS, Vol. 2* (1999), pp. 117–121.

[23] Gemmell, J., Vin, H. M., Kandlur, D. D., Rangan, P. V., e Rowe, L. A. Multimedia storage servers: A tutorial. *IEEE Computer 28*, 5 (1995), 40–49.

[24] Golubchik, L., Lui, J. C. S., e Muntz, R. R. Reducing i/o demand in video-on-demand storage servers. In *Measurement and Modeling of Computer Systems* (1995), pp. 25–36.

[25] Hua, K. A., Cai, Y., e Sheu, S. Patching : A multicast technique for true video-on-demand services. In *ACM Multimedia* (1998), pp. 191–200.

[26] Hua, K. A., e Sheu, S. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. In *SIGCOMM* (1997), pp. 89–100.

[27] James F. Kurose and Keith W. Ross. *Computer Networking - A Top-Down Approach Featuring the Internet*. Addison-Wesley, 2000.

[28] Ji, P., Kurose, J., e Woolf, B. Student behavioral model based prefetching in online tutoring system.

[29] Lau, S.-W., Lui, J. C. S., e Golubchik, L. Merging video streams in multimedia storage server: Complexity and heuristics. *Multimedia Systems 6*, 1 (1998), 29–42.

[30] Muntz, R., Santos, J., e Berson, S. A parallel disk storage system for realtime multimedia applications, 1998, pp1137-1174. In International Journal of Intelligent Systems.

[31] Netto. B. Mecanismos para garantir qualidade de serviço de aplicações de vídeo sob demanda, 2004. Tese de Mestrado, UFRJ.

[32] Sahu, S., Zhang, Z.-L., Kurose, J. F., e Towsley, D. F. On the efficient retrieval of VBR video in a multimedia server. In *International Conference on Multimedia Computing and Systems* (1997), pp. 46–53.

[33] Santos, J. R., Muntz, R. R., e Ribeiro-Neto, B. A. Comparing random data allocation and data striping in multimedia servers. In *Measurement and Modeling of Computer Systems* (2000), pp. 44–55.

[34] Santos, J. R. G. Rio: A universal multimedia storage system based on random data allocation and block replication, 1998.

[35] Sen, S., Rexford, J., e Towsley, D. F. Proxy prefix caching for multimedia streams. In *INFOCOM (3)* (1999), pp. 1310–1319.

[36] Viswanathan, S., e Imilelinski, T. Pyramid broadcasting for video on demand service. Relatório técnico, 1994.

[37] Wang, B., Sen, S., Adler, M., e Towsley, D. Optimal proxy cache allocation for efficient streaming media distribution, 2002.