

PATCHING INTERATIVO: UM NOVO MÉTODO DE COMPARTILHAMENTO  
DE RECURSOS PARA TRANSMISSÃO DE VÍDEO COM ALTA  
INTERATIVIDADE

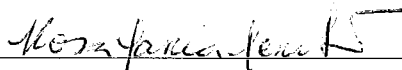
Bernardo Calil Machado Netto

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS  
PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA  
UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS  
REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE  
MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E  
COMPUTAÇÃO.

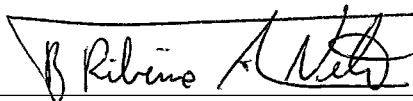
Aprovada por:



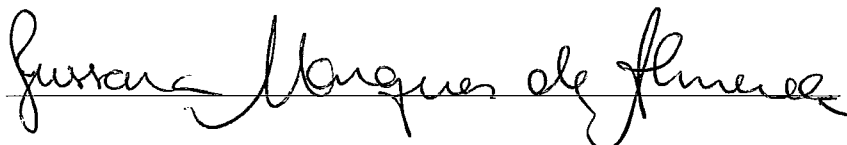
Prof. Edmundo Albuquerque de Souza e Silva, Ph.D.



Prof. Rosa Maria Meri Leão, Dr.



Prof. Berthier Ribeiro de Araujo Neto, Ph.D.



Prof. Jussara Marques de Almeida, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

FEVEREIRO DE 2004

NETTO, BERNARDO CALIL MACHADO

Patching Interativo: Um Novo Método De  
Compartilhamento De Recursos Para Trans-  
missão De Vídeo Com Alta Interatividade  
[Rio de Janeiro] 2004

XIV, 89 p. 29,7 cm (COPPE/UFRJ,  
M.Sc., Engenharia de Sistemas e Computa-  
ção, 2004)

Tese - Universidade Federal do Rio de Ja-  
neiro, COPPE

1. Vídeo sob Demanda
2. Compartilhamento de banda
3. Qualidade de Serviço
4. Transmissão de Mídia Contínua

I. COPPE/UFRJ      II. Título (Série)

*“tudo é dor e toda dor vem do desejo de não sentirmos dor”*

*(Legião Urbana)*

# Agradecimentos

Inicialmente, gostaria de agradecer aos meus familiares que sempre me deram força para continuar minha vida acadêmica, principalmente nos momentos difíceis. Sem o apoio de minha mãe Marlene e meus irmãos Marcelo e Fernanda, esta tarefa teria sido muito mais difícil. À minha mãe, gostaria de agradecer pela ajuda dispensada em momentos cruciais; à minha irmã, pela força e pelo incentivo e ao meu irmão pelas inúmeras piadas enviadas, por email, durante a execução da dissertação, porém algumas (a maioria) de qualidade ruim. Outra pessoa que me ajudou muito neste trabalho foi minha namorada Gabi, aguentando os vários momentos de sufoco e partilhando as alegrias, nunca se abatendo e sempre me dando força e incentivo a continuar.

Gostaria de agradecer, em especial, ao meu pai Wadson, que não está mais aqui comigo, mas sempre me incentivou a crescer e a lutar pelos meus ideais. Infelizmente, ele não está presente neste momento tão importante, mas sei que ficaria muito feliz com esta “nossa” conquista.

Agradecer também aos meus orientadores Edmundo e Rosa pelos ensinamentos recebidos e a oportunidade de trabalhar com pessoas tão dedicadas, que não só ajudam no desenvolvimento dos vários alunos do LAND como no crescimento da qualidade da pesquisa no Brasil.

Por fim, gostaria de agradecer ao pessoal do LAND e aos colegas de mestrado, que me ajudaram durante esse tempo, em especial: Guto, Beto, Drika, Carol, Felipe, Melba, Allyson, GD, Bruno, Ana, Flávio, Carlo Kleber, Daniel, Isabela, Fernando, Hugo, Ed., Carolina, Kelvin, Aline, Kleber, Isaac e Denilson.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PATCHING INTERATIVO: UM NOVO MÉTODO DE COMPARTILHAMENTO  
DE RECURSOS PARA TRANSMISSÃO DE VÍDEO COM ALTA  
INTERATIVIDADE

Bernardo Calil Machado Netto

Fevereiro / 2004

Orientadores: Edmundo Albuquerque de Souza e Silva

Rosa Maria Meri Leão

Programa: Engenharia de Sistemas e Computação

As técnicas para transmissão de vídeo sob demanda vêm se aprimorando para suprir as necessidades dos usuários, sendo que um dos principais problemas para esta transmissão é a alta demanda por recursos da rede. Diversas técnicas tem sido propostas na literatura onde vários clientes compartilham um mesmo fluxo, reduzindo, desta forma, a demanda por capacidade de transmissão. A maioria dessas técnicas foi concebida supondo que o cliente faz acesso seqüencial ao vídeo. O fato de os clientes executarem operações de VCR (*Video Cassete Recorder*), torna mais complexo o compartilhamento dos fluxos.

No âmbito de educação a distância, as operações de VCR são bastante freqüentes, pois os alunos têm a opção de escolha da parte da aula que lhes interessa, ou seja, de um conteúdo, que tenha gerado dúvida, ou de uma seção da aula que ainda não tenham assistido.

A proposta deste trabalho é desenvolver e implementar um método chamado *Patching* Interativo, que visa o compartilhamento dos fluxos de dados transmitidos aos clientes. Quando os usuários interagem com o vídeo, executam comandos, tais como: avançar, retroceder, pausar, entre outros. Foi desenvolvido, também, um modelo para simulação e validação do método proposto.

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

INTERACTIVE PATCHING: A NEW METHOD FOR PROVIDING  
HIGH-INTERACTIVITY VOD SERVICE

Bernardo Calil Machado Netto

February / 2004

Advisors: Edmundo Albuquerque de Souza e Silva

Rosa Maria Meri Leão

Department: Engenharia de Sistemas e Computação

In the past few years several techniques have been proposed to transmit data generated by video on demand (VoD) applications. One of the main challenges of these techniques is to reduce the bandwidth requirements of these applications. A common approach for providing scalability to VoD applications is to allow a single stream to be shared among several clients. Most of the techniques for stream sharing assume sequential streaming access. A sequential access model, however, is inappropriate to represent the behavior of all clients. This is because there are VoD clients that execute interactive operations such as fast-forward, fast-rewind and pause.

In the scope of educational media servers, VCR (Video Cassette Recorder) operations might occur often because the students may frequently rewind the video to review parts of the material that were not clearly understood in a first pass.

The proposal of this work is to develop and to implement a method, denoted as Interactive Patching, to share the data streams transmitted to the clients who execute VCR operations, namely fast-forward, fast-rewind, pause and others. We have also developed a model for simulation and validation of the above method.

# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Motivações e contribuições . . . . .	2
1.2	Organização do texto . . . . .	3
<b>2</b>	<b>Servidor Multimídia RIO</b>	<b>4</b>
2.1	Introdução . . . . .	4
2.2	Políticas de armazenamento dos dados . . . . .	6
2.3	RIO <i>Multimedia Storage Server</i> . . . . .	8
2.3.1	Visão geral . . . . .	8
2.3.2	Componentes do servidor RIO . . . . .	10
2.3.2.1	Nó servidor . . . . .	10
2.3.2.2	Nó de armazenamento . . . . .	12
2.3.2.3	Cliente . . . . .	14
2.3.3	Controle de admissão . . . . .	15
2.3.4	Policimento de pedidos . . . . .	16
<b>3</b>	<b>Técnicas de compartilhamento de recursos</b>	<b>18</b>
3.1	Técnicas orientadas às requisições dos clientes . . . . .	18

3.2	Mecanismos de difusão periódica . . . . .	23
3.3	Estratégias baseadas no uso de proxy . . . . .	25
3.4	Interatividade . . . . .	29
<b>4</b>	<b>O novo cliente do servidor RIO e a proposta do <i>Patching</i> Interativo</b>	<b>33</b>
4.1	Visão geral do <i>Patching</i> Interativo . . . . .	33
4.2	<i>RioMMClient</i> inicial . . . . .	37
4.3	Características adicionadas ao <i>RioMMClient</i> . . . . .	40
4.3.1	Arquitetura do novo cliente . . . . .	41
4.3.2	Protocolo para envio de mensagens de controle entre o cliente e o servidor . . . . .	44
4.3.2.1	Cenários . . . . .	44
4.3.3	Protocolo para envio de dados entre o cliente e o servidor e interface gráfica do riosh . . . . .	47
4.4	Detalhes da arquitetura e módulo PI . . . . .	48
4.5	Visão geral . . . . .	49
4.6	Estruturas de controle . . . . .	53
4.7	Mensagens de controle . . . . .	55
<b>5</b>	<b>Modelo do <i>Patching</i> Interativo</b>	<b>61</b>
5.1	Introdução . . . . .	61
5.2	Modelo <i>Patching</i> Interativo . . . . .	63
5.3	Resultados . . . . .	69
<b>6</b>	<b>Conclusões</b>	<b>83</b>



6.1 Trabalhos Futuros . . . . . 84

# Lista de Figuras

2.1	Arquitetura básica . . . . .	5
2.2	Técnica <i>striping</i> . . . . .	7
2.3	Técnica de alocação aleatória . . . . .	7
2.4	Arquitetura básica do RIO . . . . .	9
2.5	Componentes do RIO . . . . .	10
2.6	Componente <i>Router</i> . . . . .	12
2.7	Nó de armazenamento . . . . .	13
2.8	Protocolo de conexão com o servidor RIO . . . . .	15
3.1	<i>Batching</i> . . . . .	19
3.2	<i>Patching</i> . . . . .	21
3.3	<i>Hierarchical stream merge</i> . . . . .	23
3.4	<i>Skyscraper</i> . . . . .	24
3.5	<i>UPatch</i> . . . . .	28
3.6	<i>MPatch</i> . . . . .	28
4.1	Janelas definidas para que o cliente se junte a um grupo . . . . .	35
4.2	Algoritmo <i>Patching_Interativo</i> . . . . .	36
4.3	Componentes do <i>RioMMClient</i> . . . . .	37

4.4	Interface gráfica <i>RioMMClient</i> . . . . .	38
4.5	Utilização de um <i>pipe</i> . . . . .	39
4.6	Etapas iniciais do <i>RioMMClient</i> . . . . .	40
4.7	Arquitetura do cliente . . . . .	41
4.8	Algoritmo <i>ArmazenamentoBuffer</i> . . . . .	43
4.9	Pedidos dos blocos de dados . . . . .	43
4.10	Algoritmo Cliente . . . . .	43
4.11	Algoritmo Recebimento de Mensagens . . . . .	45
4.12	Protocolo de comunicação . . . . .	46
4.13	Movimentação do cliente . . . . .	47
4.14	Interface gráfica <i>riosh</i> . . . . .	49
4.15	Visão geral <i>patching</i> interativo . . . . .	50
4.16	Algoritmo de tratamento de pedidos . . . . .	51
4.17	Componentes PI . . . . .	51
4.18	Lista dos tamanhos das janelas de cada vídeo . . . . .	54
4.19	Lista dos clientes inativos . . . . .	54
4.20	Estrutura das informações dos clientes . . . . .	56
4.21	Estrutura de dados <i>CommMulticast</i> . . . . .	56
4.22	Troca de mensagens entre o cliente e o PI . . . . .	58
5.1	Interface gráfica inicial do <i>Tangram-II</i> . . . . .	62
5.2	Ambiente de modelagem . . . . .	62
5.3	Modelo do sistema . . . . .	63

5.4	Modelo do comportamento do cliente . . . . .	66
5.5	Exemplo do arquivo com instantes de movimento do cliente . . . . .	67
5.6	Exemplo do arquivo com a posição do vídeo para a qual o cliente se moveu . . . . .	67
5.7	Exemplo do arquivo de pausa . . . . .	68
5.8	Total de fluxos no cenário 1 . . . . .	71
5.9	Fração de tempo no cenário 1 . . . . .	71
5.10	Total de fluxos no cenário 2 . . . . .	72
5.11	Gráfico fração de tempo no cenário 2 . . . . .	73
5.12	Total de fluxos no cenário 3 . . . . .	74
5.13	Gráfico fração de tempo no cenário 3 . . . . .	75
5.14	Total de fluxos no cenário 4 . . . . .	76
5.15	Gráfico fração de tempo no cenário 4 . . . . .	76
5.16	Diagrama de estados com novas probabilidades . . . . .	77
5.17	Total de fluxos no cenário 5 . . . . .	77
5.18	Gráfico fração de tempo no cenário 5 . . . . .	78
5.19	Modelo do comportamento do cliente no cenário 6 . . . . .	79
5.20	Total de fluxos no cenário 6 . . . . .	80
5.21	Gráfico fração de tempo no cenário 6 . . . . .	81

# Lista de Tabelas

5.1	Seções utilizadas no diagrama . . . . .	65
5.2	Número máximo de fluxos ativos das técnicas <i>Patching</i> e <i>Patching</i> Interativo . . . . .	81
5.3	Economia de banda em relação aos picos em porcentagem . . . . .	82

# Palavras-chave

1. Vídeo sob Demanda.
2. Compartilhamento de banda.
3. Transmissão de Mídia Contínua.
4. Qualidade de Serviço.

# Capítulo 1

## Introdução

COM o avanço e o desenvolvimento do acesso residencial à Internet em banda larga, é possível atualmente, a transmissão de vídeo através da Internet. Outros fatores que auxiliam nesta transmissão são a maior compactação dos dados e a criação de técnicas de compartilhamento de banda, onde o fluxo de um cliente pode ser compartilhado por vários usuários, não saturando a rede de dados.

O projeto de aplicações que envolvem a transmissão de vídeo apresenta diversos desafios. O primeiro deles é o fato da Internet não garantir a qualidade de serviço (QoS) necessária a este tipo de aplicação. Apesar de existirem propostas para a Internet que visam fornecer uma QoS diferenciada para aplicações de vídeo na Internet, estas propostas envolvem mudanças em toda a estrutura da rede. O segundo desafio diz respeito ao servidor multimídia, pois este precisa ser capaz de armazenar e manipular uma grande quantidade de dados, obedecendo requisitos estreitos de tempo e garantindo um bom desempenho na rede.

Aplicações de transmissão de vídeo estão se popularizando com a chegada de canais de maior largura de banda e com projetos de educação a distância, onde os alunos assistem às aulas gravadas e armazenadas previamente.

Diante destes fatos, tornam-se necessários o estudo e o aprimoramento dessas técnicas de armazenamento de dados e compartilhamento de banda, visando a oferecer um melhor desempenho na rede para um número de clientes expressivo.

## 1.1 Motivações e contribuições

O principal objetivo desta dissertação é a proposta e implementação de uma extensão da técnica *Patching* visando dar suporte à interatividade dos clientes, mantendo o compartilhamento dos fluxos de dados melhorando assim o desempenho do sistema. A técnica de *Patching* tem sido usada em diversos trabalhos recentes da literatura [38] e tem como principais vantagens a sua simplicidade e eficiência em termos de banda média requerida, quando comparada com outras técnicas mais sofisticadas da literatura. A implementação foi feita no servidor Multimídia RIO (*Randomized I/O Multimedia Storage Server*) [35, 14]. Este servidor será usado no projeto de educação a distância do CEDERJ (Centro de Ensino Superior a Distância do Estado do Rio de Janeiro) [10], onde os alunos terão acesso às aulas através do cliente *RioMMClient* (cliente de visualização de vídeos do RIO), que será descrito neste trabalho. Neste projeto, estão engajadas as universidades UFRJ (Universidade Federal do Rio de Janeiro), UFF (Universidade Federal Fluminense) e UFMG (Universidade Federal de Minas Gerais).

Outro projeto relacionado a este trabalho é o MAPPED (Mecanismos de Adaptação e Controle para Recuperação e transmissão Multimídia com Aplicação ao Centro de Educação Superior a Distância do Estado do Rio de Janeiro)[31], financiado pelo CNPq, que inclui dentro de seus objetivos, o desenvolvimento e a implementação de algoritmos de adaptação no servidor multimídia RIO, para o uso no projeto CEDERJ.

As principais contribuições deste trabalho são a proposta e a implementação de um módulo, chamado de *Patching* Interativo (PI) para o servidor RIO, que provê o compartilhamento dos fluxos de dados enviados aos clientes, no caso em que estes acessam o vídeo de forma sequencial ou não, bem como a inclusão de novas características no cliente *RioMMClient*, viabilizando tal compartilhamento. Além disso, o protocolo de transmissão de dados do servidor foi alterado para melhorar o desempenho desta transmissão.

Foi desenvolvido, também, no âmbito desta dissertação, um módulo no cliente



que armazena os dados recebidos do servidor, para que futuramente estes dados não sejam solicitados. O cliente foi alterado para suportar o recebimento de dados, através de *multicast*, e também para ter a capacidade de receber dois fluxos de dados simultâneos.

Para os testes e a validação da proposta, foi desenvolvido um modelo, com o uso da ferramenta TANGRAM-II [8, 29, 15], para a simulação dos clientes e do módulo PI, onde os usuários chegam ao sistema e executam algumas operações, tais como: avançar, retroceder, pausar.

## 1.2 Organização do texto

No capítulo 2, é feita uma descrição de sistemas multimídia em geral e são detalhadas as características do servidor multimídia RIO, que será usado como base da implementação deste trabalho.

São descritas, no capítulo 3, algumas técnicas de compartilhamento de banda, apresentadas na literatura, que minimizam o uso da rede e são apresentados, também, alguns problemas desses mecanismos.

No capítulo 4, é feita uma descrição detalhada do cliente *RioMMClient*, onde são apresentadas todas as características e módulos adicionados a este cliente. É feita, também, uma enumeração das mensagens criadas para a comunicação entre o cliente e o servidor.

O módulo do *Patching* Interativo, descrito no capítulo 5, foi implementado no servidor RIO para dar suporte ao compartilhamento dos fluxos enviados aos clientes. É feita, também, a apresentação das mensagens e estruturas criadas para auxiliar neste compartilhamento.

No capítulo 6, é apresentado o modelo desenvolvido para simular os clientes acessando o servidor com o módulo PI. São também descritos os resultados gerados com as simulações. No capítulo 7, são apresentadas as conclusões dessa dissertação assim como sugestões para trabalhos futuros.

# Capítulo 2

## Servidor Multimídia RIO

### 2.1 Introdução

**N**OS últimos anos o aumento da largura de banda, o aperfeiçoamento dos algoritmos de compressão e a maior capacidade de processamento dos computadores propiciaram a transmissão de vídeos através da Internet. Em paralelo, a alta demanda de banda necessária a transmissão de vídeo necessitaram o estudo de técnicas para o uso mais inteligente dos recursos da rede, já que o meio de transmissão tem uma capacidade, limitada. Baseado em [27], podemos classificar as aplicações multimídia em três tipos:

#### Áudio e vídeo pré-armazenados

Este tipo de transmissão se dá em um ambiente em que a mídia foi previamente codificada e armazenada em um servidor. Os clientes, quando conectados ao servidor, podem escolher a mídia desejada para visualização, o que pode propiciar algumas funcionalidades como: pausar, avançar e retroceder. Como consequência, o usuário tem a sua disposição um ambiente interativo, onde ele pode enviar comandos ao servidor que permitem a visualização da parte do vídeo/áudio desejada.

## Áudio e vídeo ao vivo

Neste tipo de aplicação não existe o armazenamento prévio das informações, os dados são transmitidos logo após a sua geração. Existe apenas uma ponte de informação, portanto os dados são transmitidos somente no sentido da aplicação para os clientes. Podemos citar como exemplo, as rádios pela Internet.

## Áudio e vídeo interativo em tempo real

Nestas aplicações o vídeo e o áudio são gerados e imediatamente transmitidos e é permitida interatividade entre os diversos clientes. Um exemplo para este tipo de transmissão é uma vídeo conferência.

A Figura 2.1 mostra um conjunto de clientes para um servidor multimídia, onde cada cliente decide qual mídia irá visualizar e tem a possibilidade de executar comandos de avançar, retroceder, entre outros. O servidor multimídia é o responsável pelo armazenamento dos objetos (vídeo, áudio, entre outros) nos discos do sistema, pela transmissão dos dados para o cliente e pela leitura dos dados de cada objeto solicitado por algum cliente. A rede é responsável pela comunicação física entre o cliente e o servidor.

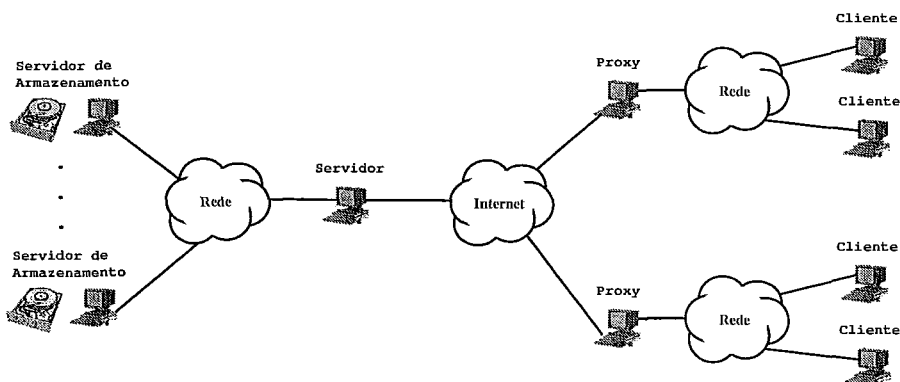


Figura 2.1: Arquitetura básica

Para a comunicação entre o cliente e o servidor podem ser usados protocolos de domínio público ou protocolos proprietários. Os protocolos de domínio público RTSP (*Real Time Streaming Protocol*), RTP (*Real Time Protocol*), RTCP (*Real*

*Time Control Protocol*), são alguns dos padrões definidos em [27].

## 2.2 Políticas de armazenamento dos dados

A forma mais utilizada de armazenamento de dados é a divisão do objeto em blocos e distribuídos nos discos de acordo com alguma política de armazenamento. Os blocos são lidos dos discos, armazenados em um *buffer* do servidor para enviar posteriormente ao cliente.

A organização dos dados nos discos é muito importante para o bom funcionamento do sistema multimídia. Esses dados podem ser armazenados com uma política muito simples, ou seja, todos os blocos de dados de um objeto são armazenados no mesmo disco. Porém, desta forma, poderíamos ter uma sobrecarga nos discos que contiverem os objetos mais populares. Para minimizar esta sobrecarga existe outra técnica onde os blocos de dados de um objeto são distribuídos por todos os discos, fazendo assim uma distribuição da carga.

Baseado nesta última técnica, foram desenvolvidas várias abordagens para distribuir os blocos de dados nos discos e elas consideram o padrão de acesso dos clientes para determinar esta alocação. A forma mais utilizada é a técnica de *Striping* [34], na qual o objeto é dividido em blocos de dados de tamanho fixo, chamados de *stripe units*, e distribuídos nos discos, usando a técnica *Round Robin*, ou seja, os blocos consecutivos serão armazenados em discos consecutivos. Este esquema, mostrado na Figura 2.2, considera que os clientes acessam os dados com taxa constante e de forma seqüencial. Em cada ciclo, o servidor faz a leitura de um bloco de dados de um dos discos para cada cliente.

Outra técnica para o armazenamento dos blocos de dados nos discos dos sistemas multimídia é a alocação aleatória dos blocos, *Random Data Allocation* [5, 35, 34]. Nesta técnica, é escolhido um disco aleatório e uma posição aleatória dentro do disco para o armazenamento, como mostra a Figura 2.3.

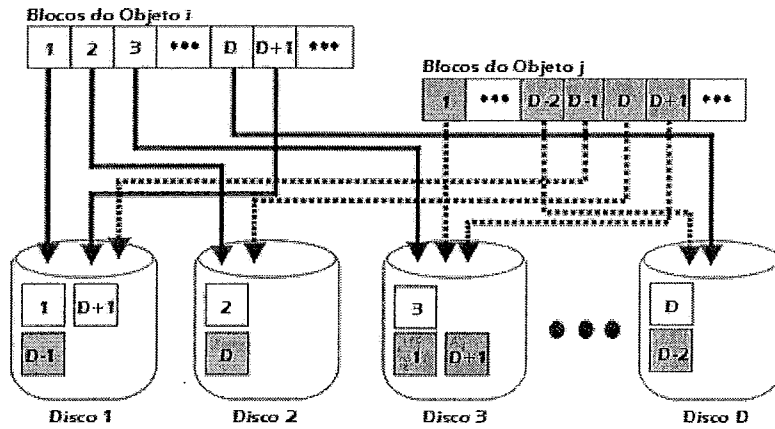


Figura 2.2: Técnica *striping*

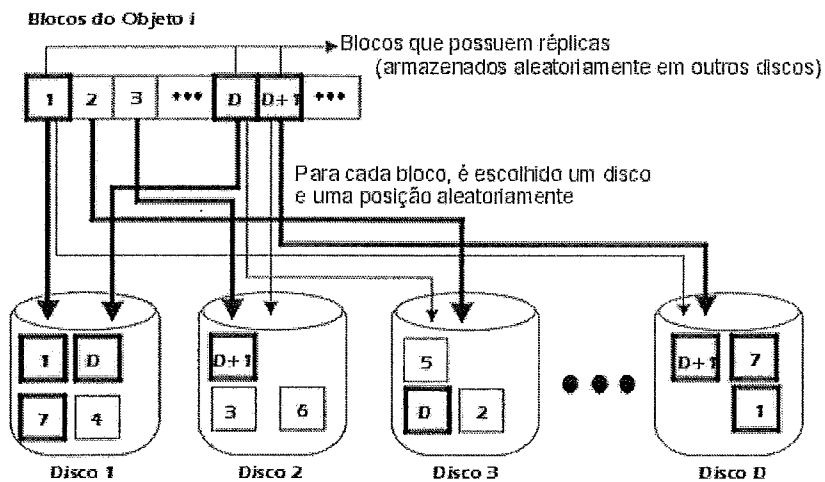


Figura 2.3: Técnica de alocação aleatória

## 2.3 RIO *Multimedia Storage Server*

O objetivo desta seção é apresentar as principais funcionalidades do servidor RIO e uma descrição de seus componentes. O servidor foi originalmente desenvolvido na *University of California at Los Angeles* (UCLA) e uma série de novas funcionalidades foram adicionadas conforme descrito em [14].

### 2.3.1 Visão geral

O Servidor RIO é um sistema de armazenamento multimídia universal que usa alocação aleatória e replicação de blocos. Sendo um servidor universal, o RIO suporta vários tipos de mídias: vídeo, áudio, texto, imagem, além de ser capaz de gerenciar aplicações com ou sem restrição de tempo. Aplicações como visualização de imagens e textos são exemplos de aplicações sem restrição de tempo. Por outro lado a exibição de um vídeo sob demanda é um exemplo de aplicação de tempo real.

Para o servidor, todos os tipos de mídias são chamados de objetos e são armazenados da mesma forma. Os objetos são divididos em blocos de dados e estes são armazenados, aleatoriamente, como descrito na seção 2.2. O tamanho do bloco de dados é definido na configuração do servidor.

O RIO foi desenvolvido pelo laboratório de multimídia da UCLA, sendo que o primeiro protótipo foi implementado para SUN E4000 e utilizado para exibição de vídeos MPEG e simulação 3D. Logo após, foi implementada uma versão para um *cluster* de PCs com sistema operacional Linux. Esta versão foi desenvolvida usando a linguagem C++ e disponibilizou um cliente de visualização de vídeos MPEG (*ri-omtv*) e um cliente interpretador de comandos para executar a administração dos objetos do servidor (*riosh*). Posteriormente, o grupo de pesquisas LAND (Laboratório de ANálise, modelagem e Desenvolvimento de redes e sistemas de computação) [28] desenvolveu um outro cliente para visualização de vídeos, o *RioMMClient*, com mais funcionalidades. O *RioMMClient* possui uma interface gráfica desenvolvida em C++, para interação do usuário com o vídeo, ou seja, o cliente pode executar os comandos de avançar, retroceder e pausar. Além desta interação com o usuário,

o *RioMMClient* também possui um módulo de sincronização com transparências. Este módulo é utilizado, por exemplo, em uma aula, onde o vídeo do professor é exibido acompanhado de uma transparência. Caso o usuário avance ou retroceda, a transparência acompanha este movimento.

Em [14], foram desenvolvidas diversas novas funcionalidades para o RIO, quais sejam: foi implementado no servidor *buffers* para permitir o gerenciamento dos dados solicitados pelo cliente de forma a minimizar o *jiter* decorrente da leitura nos discos e da transmissão pela rede. Outra característica adicionada foi um novo algoritmo detalhado para a admissão de novos usuários no sistema, além de um módulo para retirada de diversas medidas de desempenho, como tempo médio de leitura de um bloco de dados nos discos.

A Figura 2.4 [14] mostra que a arquitetura do RIO é composta por um único nó servidor, um ou mais nós de armazenamento e os clientes. A comunicação entre o cliente e o servidor é feita usando os protocolos TCP (*Transmission Control Protocol*) e UDP (*User Datagram Protocol*), sendo o primeiro para comunicação de controle, como pedidos de blocos de dados, e o segundo para a transmissão dos blocos. Como pode ser observado na Figura 2.4 [14], os blocos são transmitidos diretamente dos nós de armazenamento para os clientes.

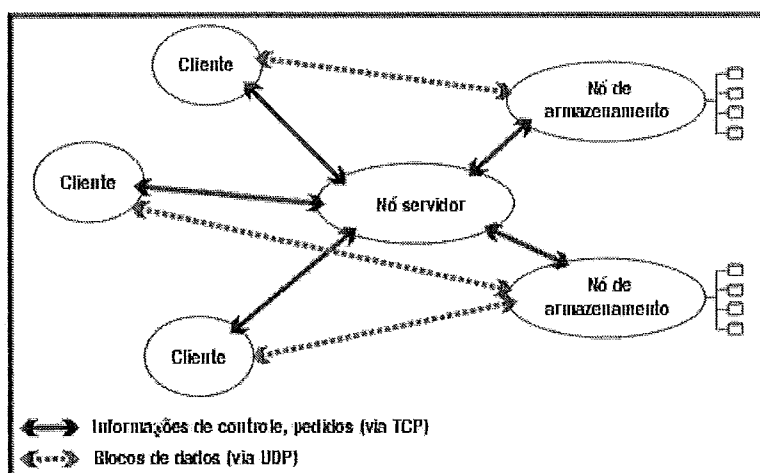


Figura 2.4: Arquitetura básica do RIO

### 2.3.2 Componentes do servidor RIO

Nesta seção apresentaremos os componentes da arquitetura do RIO. Estes componentes podem ser visualizados na Figura 2.5 [14].

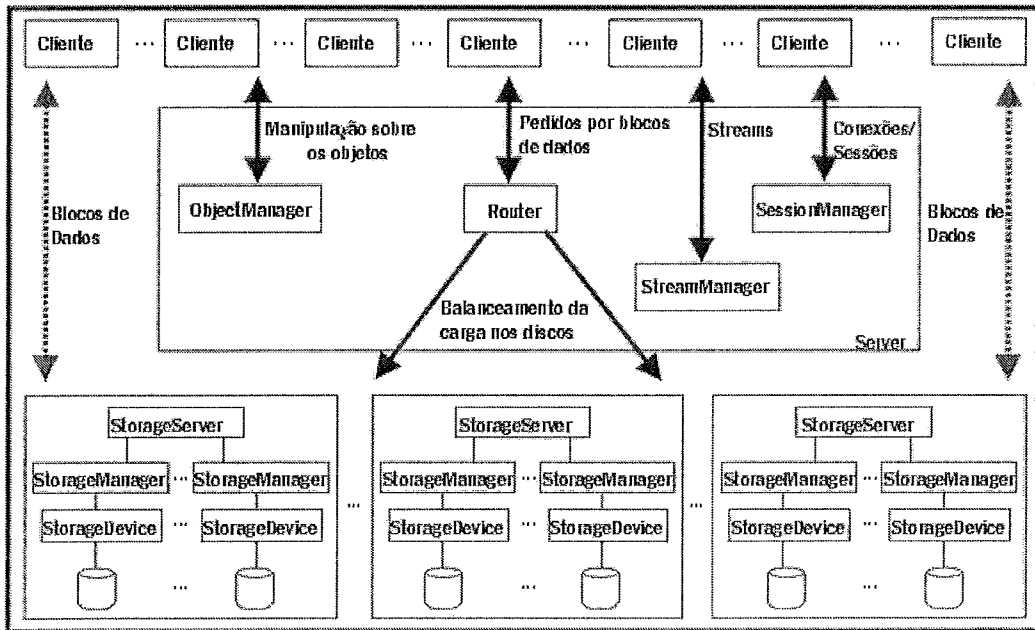


Figura 2.5: Componentes do RIO

#### 2.3.2.1 Nó servidor

Na seção 2.3.1 foi descrito que, na arquitetura do RIO, existe apenas um nó servidor. Serão detalhados agora os principais componentes desse nó, que são o *SessionManager*, o *StreamManager*, o *ObjectManager* e o *Router*.

#### Gerenciador de Sessão (*SessionManager*)

O *SessionManager* é o ponto de conexão do cliente com o servidor. Ele é responsável por todos os pedidos do cliente e a cada nova sessão é iniciada uma nova *thread* para o tratamento dos pedidos daquele cliente. Como mostra a Figura 2.5, cada pedido do cliente pode executar um método do próprio *SessionManager*, do *StreamManager*, do *ObjectManager* ou do *Router*.



### Gerenciador de Fluxos (*StreamManager*)

O *StreamManager* é responsável pelo gerenciamento dos fluxos abertos no servidor e pela abertura de novos fluxos. É responsável, também, pela execução do controle de admissão que será descrito na seção 2.3.3.

Os pedidos de cada fluxo são encaminhados ao *Router* para serem atendidos, porém, nem sempre, são processados no momento de sua chegada pois, como será descrito na seção 2.3.4, os pedidos entram em uma fila antes de sua execução. Cada fluxo possui uma fila, na qual o pedido fica aguardando o seu momento de ser atendido. Esta fila é verificada periodicamente pelo *StreamManager*, fazendo com que os pedidos pendentes sejam enviados para o *Router* e neste envio é priorizado o tráfego de tempo real.

### Gerenciador de Objetos (*ObjectManager*)

O *ObjectManager* é responsável pelos metadados dos objetos armazenados no servidor. O gerenciamento das operações sobre os objetos, tais como: criação, exclusão, abertura e fechamento é outra função do *ObjectManager*.

Quando algum cliente se conecta ao servidor para a criação de um novo objeto, o *ObjectManager* deve alocar os blocos e sua(s) réplica(s) e atualizar os metadados dos objetos e discos, o mesmo ocorre para a exclusão de objetos.

### Roteador (*Router*)

O componente *Router* possui uma fila para cada disco do sistema, como mostra a Figura 2.6 [14], uma fila para os pedidos de tempo real (Fila de RT) e outra para os pedidos sem restrição de tempo (Fila de NRT). O envio de pedidos para o disco começa pela fila de tempo real e o atendimento de cada fila é FIFO.

O *Router* é responsável por receber os pedidos de leitura/escrita, gerados pelos fluxos, escolher o(s) dispositivo(s) para atender aos pedidos, enviar os comandos para os nós de armazenamento responsáveis e receber as mensagens de controle

enviadas por estes nós.

À medida que os pedidos vindos dos fluxos chegam ao *Router* é feito o mapeamento do bloco, usando o *ObjectManager*. Caso o pedido seja de leitura, ele é redirecionado para o disco que contém aquele bloco e que possua a menor fila. Se o pedido for de escrita, é necessário que todas as réplicas do bloco também sejam atualizadas; logo é gerada uma requisição para todos os discos que contenham cópia daquele bloco.

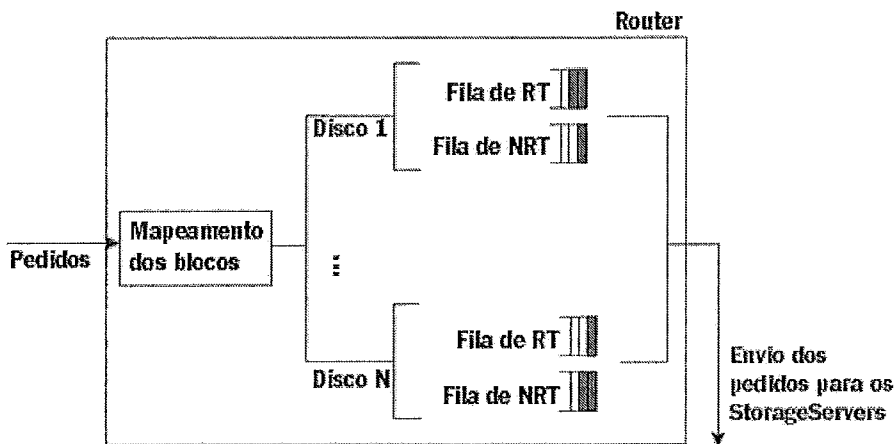


Figura 2.6: Componente *Router*

### 2.3.2.2 Nó de armazenamento

Cada servidor de armazenamento, como mostra a Figura 2.7 [14], tem os seguintes componentes: *Router Interface*, *StorageDevice*, *StorageManager* e *ClientInterface*.

#### Interface com o Roteador (*RouterInterface*)

O *RouterInterface* é responsável pelo recebimento e pelo envio de informações de controle entre o nó de armazenamento e o nó servidor. Esta conexão é feita usando o protocolo TCP. As informações de controle podem ser, por exemplo, pedido de envio de bloco de dados para um cliente e neste caso, o pedido é imediatamente repassado para o *StorageManager*, caso contrário, é enviado ao *ClientInterface*.

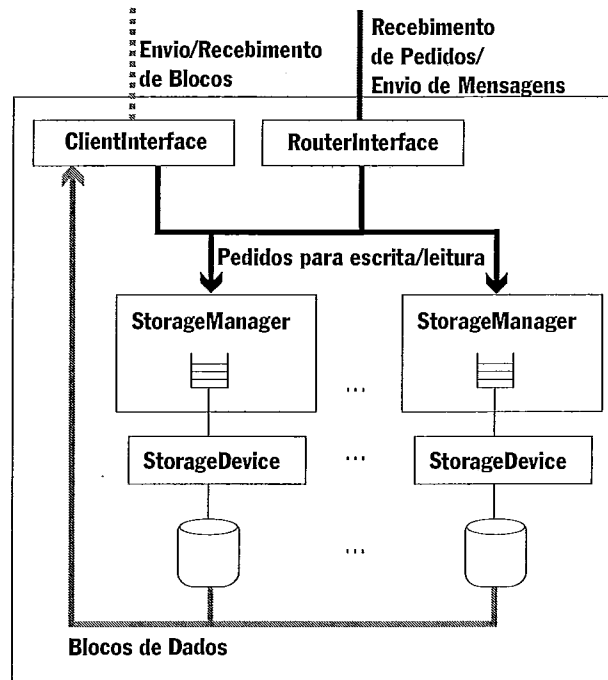


Figura 2.7: Nó de armazenamento

### Dispositivo de Armazenamento (*StorageDevice*)

Para cada disco do sistema existe uma instância do *StorageDevice* que é responsável pela realização das operações de entrada e saída. Estas operações são feitas com chamadas do sistema operacional.

### Gerenciador de Armazenamento (*StorageManager*)

Para cada *StorageDevice* existe um *StorageManager* para gerenciá-lo. O *StorageManager* é responsável pelo escalonamento dos pedidos dos dados armazenados neste dispositivo. Para executar tal tarefa, o *StorageManager* possui uma fila com os pedidos a serem atendidos, como mostra a Figura 2.7 [14]. Para cada pedido, é associado um *buffer* para o armazenamento do bloco solicitado.

No caso dos pedidos de leitura, após a sua execução, o *StorageManager* solicita ao *ClientInterface* o envio do bloco ao cliente. Para pedidos de escrita, o *StorageManager* é responsável pelo armazenamento do bloco.

### Interface com o Cliente (*ClientInterface*)

O componente *ClientInterface* é responsável pelo envio/recebimento dos blocos de dados a serem transmitidos/armazenados. É utilizado o protocolo UDP para essa transmissão. Cada bloco a ser enviado é dividido em vários pacotes e o transmissor controla, através de algumas variáveis, a quantidade de pacotes recebida pelo receptor. Alguns exemplos dessas variáveis são: quantidade de pacotes que compõe o bloco e o endereço do receptor (IP, porta, identificação da requisição). Como pode ser observado, temos um fluxo *unicast* para cada cliente o que pode gerar o esgotamento da banda passante quando o número de clientes aumenta. Existem técnicas para a melhoria desta transmissão que utilizam o *multicast* como base e estas, serão apresentadas no capítulo 3. No capítulo 4.4 será apresentada uma proposta para a melhoria de um destes métodos.

#### 2.3.2.3 Cliente

##### Cliente riosh

É o interpretador de comandos do servidor RIO. Cada usuário possui um nome de usuário e senha para que possa efetuar sua identificação. Com o riosh, o cliente pode executar várias operações como, copiar um vídeo para o servidor, criar/apagar diretórios, listar os arquivos do servidor, dentre outras funções.

##### Cliente *RioMMClient*

O *RioMMClient* é o cliente de visualização de vídeos do servidor RIO. Possui funcionalidades como avançar, retroceder, pausar, além de sincronização com transparências. É utilizado como tocador de mídia o *MPlayer* [21] e a interface gráfica é feita usando C++. Não serão apresentados agora os detalhes do cliente *RioMMClient*, pois este será explicado no capítulo 4.

Os clientes, para se conectarem ao servidor RIO, devem seguir um protocolo, que pode ser visualizado na Figura 2.8. Nesta figura supões-se que o *buffer* do cliente

tem duas posições. Em primeiro lugar é solicitada a abertura de sessão e o cliente recebe como resposta o tamanho do bloco de dados utilizado pelo sistema. Após este recebimento, é enviado ao servidor o pedido de abertura de fluxo, que tem como resposta o número máximo de pedidos ativos deste cliente. Neste momento o usuário solicita a abertura do objeto que ele deseja visualizar e, na resposta, o servidor informa o tamanho em *bytes* deste objeto. A partir deste instante o cliente solicita os blocos de dados iniciais para o preenchimento do seu *buffer* e após a chegada destes envia a mensagem *CanStart*, para saber se o *buffer* do servidor já está cheio. Após o recebimento desta resposta o cliente inicia a exibição do primeiro bloco de dados e em seu término solicita o bloco 3 para o preenchimento da posição vazia de seu *buffer*.

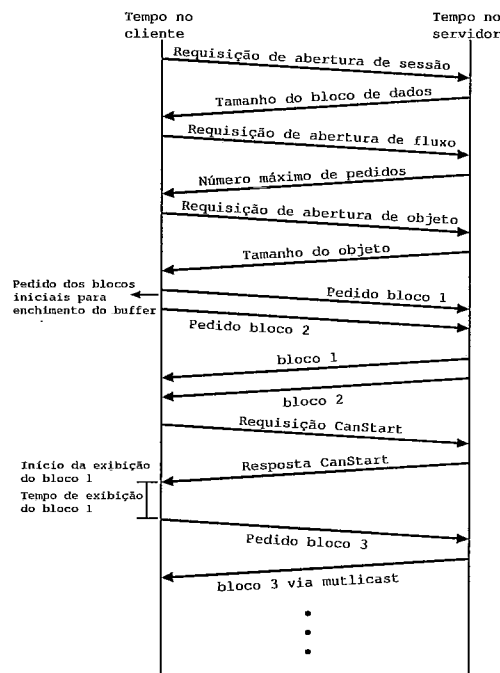


Figura 2.8: Protocolo de conexão com o servidor RIO

### 2.3.3 Controle de admissão

O controle de admissão de usuários, usado pelo servidor RIO, é executado no módulo *StreamManager* (Gerenciador de Fluxo) no momento em que o cliente abre o fluxo de dados.

Em [14], foi implementado um outro controle de admissão, mais detalhado, porém com um custo computacional elevado. Entretanto descreveremos abaixo um controle mais simples baseado nas taxas solicitadas por cada cliente. Estão disponíveis no servidor RIO as duas implementações e usaremos o controle mais simples neste trabalho, pois o custo computacional é bem mais baixo em relação ao outro.

Durante a inicialização do servidor, é lido um arquivo de configuração, no qual é informada a taxa total (*TaxaTotal*) aceita pelo servidor e a taxa reservada (*TaxaReservadaNTR*) para fluxos que não possuem restrições de tempo. Esta reserva de banda garante que tais aplicações serão atendidas pelos discos, mesmo com a existência de fluxos com restrição de tempo. Outra variável do sistema é a *TaxaAlocada*, que é inicializada com o valor 0 e, de acordo com os requisitos de cada cliente, essa variável é incrementada. Para a admissão de um cliente, este envia ao servidor o tipo de tráfego (tempo real ou não), direção do tráfego (escrita ou leitura) e a *TaxaSolicitada*. O controle considera somente os fluxos de tempo real e um novo usuário é aceito se a seguinte condição for verdadeira:

$$(TaxaAlocada + TaxaSolicitada) \leq (TaxaTotal - TaxaReservadaNTR)$$

Caso esta condição seja verdadeira, a *TaxaAlocada* é acrescida da *TaxaSolicitada* e esta também será usada no policiamento do tráfego que será descrito na seção 2.3.4.

### 2.3.4 Policiamento de pedidos

O objetivo deste policiamento é não deixar que um determinado cliente solicite muitos dados em um curto intervalo de tempo, fazendo com que os pedidos dos demais clientes possam atrasar. Este policiamento é feito pelo Gerenciador de Fluxos através de um *leaky bucket* que possui dois parâmetros: a capacidade de fichas que cada cliente pode armazenar (representada por  $F$ ) e a taxa de geração de fichas (representada por  $r$ ). Quando se envia um pedido para o *Router*, deve-se consumir um ficha. Desta forma, a fila de pedidos armazena os pedidos caso a sua taxa de chegada seja maior que a taxa de geração de fichas e o número de fichas seja 0. A quantidade de fichas vai sendo incrementada de acordo com a taxa de geração e tem

como limite o valor  $F$ . Este valor é configurado na inicialização do servidor e a taxa de geração de fichas é configurada a partir da taxa solicitada pelo cliente. A quantidade máxima de pedidos que serão repassados em qualquer intervalo de tempo  $t$  é  $rt + F$ . Este mecanismo foi implementado em [14]. O mecanismo anterior, implementado na primeira versão do RIO em [35], policiava os pedidos dos clientes da seguinte forma: baseado na taxa solicitada pelo cliente o servidor calculava o intervalo médio entre pedidos e com isso não deixava o cliente solicitar os blocos em intervalos menores que a media calculada. O mecanismo usando *leaky bucket* possui algumas vantagens em relação ao mecanismo anterior, pois o cliente pode acumular fichas durante a sua execução, como por exemplo, quando a exibição do vídeo é pausada. Quando o cliente executa operações que alteram o padrão de acesso, como as operações de VCR, onde os *buffers*, tanto do cliente, quanto do servidor são esvaziados, através deste mecanismo com *leaky bucket* é possível o preenchimento mais rápido destes *buffers*.

## Capítulo 3

# Técnicas de compartilhamento de recursos

VISANDO permitir o acesso ao servidor multimídia de um grande número de clientes, isto é, aumentar a escalabilidade para esta aplicação, várias técnicas de compartilhamento de banda tem sido propostas. Estas técnicas podem ser divididas em 2 tipos [16]: técnicas orientadas às requisições dos clientes e mecanismos de difusão periódica. A primeira técnica se baseia na transmissão, por parte do servidor, de um fluxo de mídia contínua em resposta a pedidos dos cliente. A segunda se baseia na difusão periódica de segmentos da mídia. Além disso, essas técnicas podem ser combinadas com o uso de servidores *proxy*.

### 3.1 Técnicas orientadas às requisições dos clientes

Nesta seção serão abordadas as técnicas de compartilhamento de recursos baseadas em requisições dos clientes, quais sejam: *batching*, *stream tapping*, *patching*, *controlled multicast*, *piggybacking*, *hierarchical stream merging* e *bandwidth skimming*. A técnica *patching* será apresentada em maior detalhe pois é o objeto de estudo deste trabalho.



### *Batching*

Nesta técnica [2, 1, 13], quando o primeiro cliente solicita um determinado fluxo, é iniciada uma janela, e este cliente é colocado em uma fila de espera. Os demais usuários que solicitarem este mesmo fluxo, entrarão nesta mesma fila até que a janela expire. Neste momento, é iniciada uma transmissão *multicast* do fluxo para o grupo de clientes que está na fila de espera. Esta técnica causa uma latência inicial para os cliente proporcional ao tamanho da janela. Como pode ser observado na Figura 3.1, o primeiro pedido  $r_0$  para o fluxo  $s$  inicia a janela do *batching*. As requisições  $r_1$  e  $r_2$ , que também são para o fluxo  $s$ , chegam dentro da janela e são agrupadas no mesmo fluxo. A transmissão para este grupo se inicia assim que a janela expira.

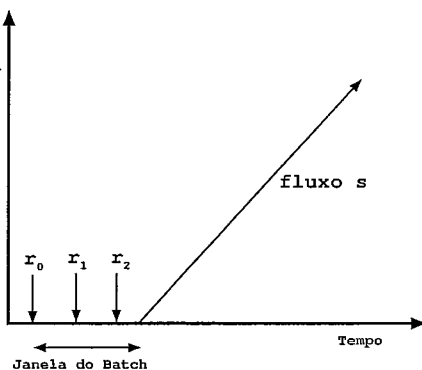


Figura 3.1: *Batching*

### *Stream tapping, patching e controlled multicast*

Estas técnicas são bastante similares e provêem serviço imediato para o cliente, diferentemente do *batching*. No esquema básico do *patching* [24], o servidor mantém uma fila com todos os pedidos pendentes. Quando um canal se torna disponível o servidor inicia uma transmissão *multicast* para todos os clientes da fila que solicitaram o fluxo  $S$  (grupo  $A$  de transmissão). Considere que o fluxo  $S$  já esteja sendo transmitido para um outro grupo  $B$  de clientes. Então, o grupo  $A$  irá receber dois fluxos: um que foi iniciado pelos clientes do grupo  $B$  (estes dados serão armazenados, para exibição futura) e outro que contém a parte inicial do fluxo  $S$  denominada *patch* (que será imediatamente exibida). Nesta técnica o cliente deve ter capacidade

para receber dois fluxos ao mesmo tempo e espaço em *buffer* para armazenar parte do fluxo que será exibido no futuro.

As técnicas *stream tapping* [9], *optimal patching* [7] e *controlled multicast* [20] se diferenciam do *patching* básico, uma vez que definem uma janela para o *patching*. Esta janela é o intervalo mínimo entre o instante inicial de duas transmissões completas do mesmo fluxo.

O modelo definido para cálculo da janela é baseado na hipótese de que as chegadas dos pedidos dos clientes obedecem a um Processo de *Poisson* com média  $\lambda$  para o fluxo  $S$ . Logo pode ser mostrado que a capacidade do servidor é dada por [20]:

$$c = T + \lambda w^2/2/w + 1/\lambda \quad (3.1)$$

onde  $T$  é a duração do fluxo  $S$  e  $w$  é o tamanho da janela.

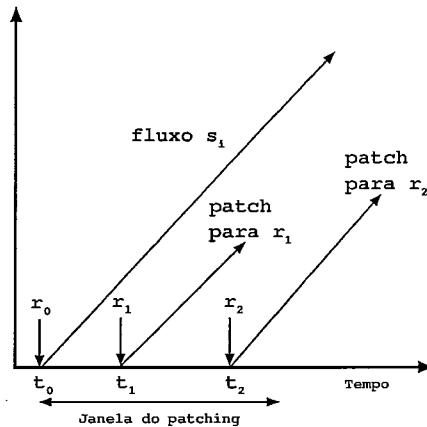
A janela ótima, ou seja, aquela que minimiza os requisitos de banda, pode ser obtida diferenciando-se a expressão 3.1 [20]:

$$w_{ot} = \sqrt{2N + 1} - 1/\lambda \quad (3.2)$$

onde  $N = \lambda T$ . Substituindo 3.2 em 3.1 temos a capacidade média para a janela ótima dada por:

$$C_{ot} = \sqrt{2N + 1} - 1 \quad (3.3)$$

A Figura 3.2 apresenta um exemplo das técnicas citadas acima. A transmissão  $s_i$  é iniciada a partir da chegada do pedido  $r_0$ . Os pedidos  $r_1$  e  $r_2$  chegam dentro da janela, assim estes clientes recebem os dados vindos do fluxo  $s_i$  e  $r_1$  recebe o *patch* do intervalo  $(t_1 - t_0)$  e  $r_2$  recebe o *patch* do intervalo  $(t_2 - t_0)$ . O pedido  $r_3$  que chega após o término da janela irá iniciar uma nova janela.

Figura 3.2: *Patching*

### *Piggybacking*

Esta técnica [3] consiste na mudança dinâmica da taxa de exibição para permitir que um fluxo possa alcançar e se juntar a outro fluxo. Suponhamos que o fluxo  $s_i$  esteja sendo transmitido para um cliente; se um novo cliente chega para este mesmo fluxo, então uma nova transmissão de  $s_i$  é iniciada. Neste momento, o servidor diminui a taxa de transmissão de dados para o primeiro cliente e aumenta a taxa do segundo. Desta forma, as duas transmissões irão se sincronizar em algum instante no futuro, podendo ser unidas e liberar um canal de transmissão. Uma desvantagem desta técnica é que ela requer um hardware especializado para suportar mudanças dinâmicas na velocidade do canal.

### *Hierarchical Stream Merging (HSM)*

A idéia básica desta técnica é a junção hierárquica dos clientes em grupos. O cliente recebe simultaneamente dois fluxos: um iniciado por ele e outro iniciado pelo cliente mais próximo. A grande questão desta técnica é escolher qual fluxo anterior cada cliente irá escutar, e para isto é desenvolvida uma árvore de “merges”. Existem diversas abordagens para a construção desta árvore e algumas delas são apresentados em [19]. Neste trabalho é determinado, através de programação dinâmica, a árvore ótima para a junção dos fluxos, considerando que os instantes de todas as chegadas

são conhecidos.

Como estas chegadas de clientes não são conhecidas em sistemas reais, então a busca por outras formas de determinação da árvore se faz necessária. São apresentadas 3 heurísticas para a construção da árvore, onde a primeira abordagem apresentada é a *Earliest Reachable Merge Target* (ERMT), onde um novo cliente, ou um novo grupo recém unido, escuta o próximo cliente com o qual possa se unir mais rapidamente, considerando que não haja chegadas futuras que possam ser unidas com eles. Esta decisão é feita através de uma simulação de todas as uniões a fim de determinar qual fluxo deve ser escutado para não ter desperdício de escuta. Esta simulação pode ser muito custosa computacionalmente e para minimizar este problema, foi definida a política *Simple Reachable Merge Target* (SRMT), onde se determina o fluxo alcançável mais próximo considerando que cada fluxo ativo termina em seu *target merge point*, que é o instante onde cada fluxo irá se juntar com o seu fluxo mais próximo. A técnica SRMT tem uma desvantagem, pois pode haver desperdício de escuta, uma vez que um cliente poderá estar escutando um segundo fluxo que se juntará a um terceiro antes mesmo do primeiro conseguir se unir ao segundo, levando o primeiro cliente a escutar todos os dados restantes para alcançar o terceiro.

É definido, também, uma terceira abordagem chamada *Closest Target* (CT), bem mais simples que as anteriores, onde cada cliente, ao chegar no sistema, escuta o fluxo ativo anterior a chegada dele, sem se preocupar se irá alcançá-lo em tempo hábil ou não. Este trabalho conclui que a ERMT se aproxima do resultado obtido com a árvore ótima, porém conclui, também, que a técnica CT, apesar de bem mais simples, apresenta resultados satisfatórios em termos de economia de banda.

A Figura 3.3 mostra um exemplo das uniões executadas quando utilizada a técnica de merge ótimo. O Cliente B escuta o fluxo do A e os clientes C e D escutam o fluxo iniciado por B. o cliente D escutará B pois ele não conseguirá se juntar com C antes de C se unir a B (pois a união C/B ocorre em 0.5), o que torna B o merge target alcançável mais próximo para D se chegadas futuras não forem se juntar a D.

A técnica *Bandwidth Skimming* [18] se baseia no *hierarquical stream merge* e

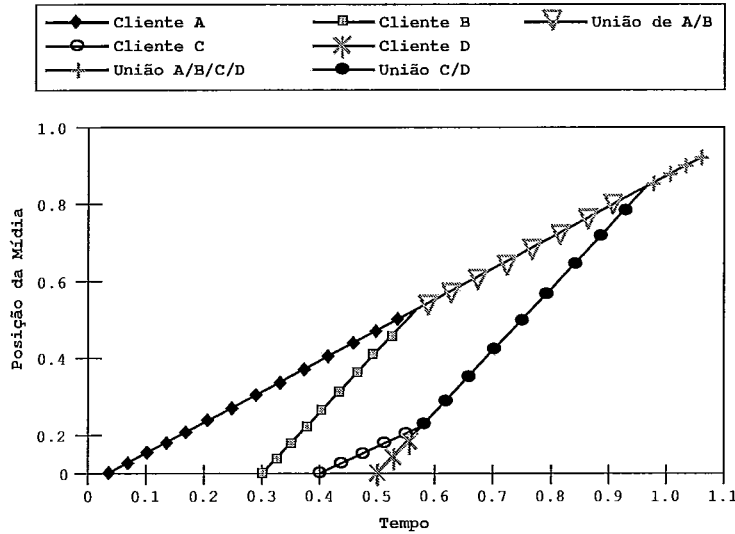


Figura 3.3: *Hierarchical stream merge*

define algumas políticas para a redução dos requisitos de banda para menos de duas vezes a taxa de exibição do fluxo.

Em [12], é obtida uma expressão para a banda média requerida no servidor para as técnicas de HSM e BS. Suponha que a taxa de transmissão requisitada pelo cliente seja igual a  $b$  unidades da taxa de exibição da mídia ( $b = 2$  para o HSM e  $b < 2$  para o BS) e as chegadas de requisições são *Poisson* com taxa média  $\lambda$  para o fluxo  $s$ . A banda requerida no servidor para a transmissão do fluxo  $s$  pode ser aproximada por:  $B_{HSM,BS} = \eta_b \ln(N_i/\eta_b + 1)$ , onde  $N = \lambda T$  e  $\eta_b$  é a constante real positiva que satisfaz a equação:  $\eta_b [1 - (\eta_b/(\eta_b + 1))^b] = 1$  [12].

## 3.2 Mecanismos de difusão periódica

Esta técnica consiste em dividir cada fluxo em segmentos que podem ser transmitidos simultaneamente em um conjunto de  $k$  canais diferentes. Esses segmentos são transmitidos periodicamente usando o mecanismo de difusão, onde o canal  $c_1$  encaminha o primeiro segmento e os outros  $k - 1$  canais encaminham o restante do fluxo. No momento da chegada de um cliente, este deve aguardar o início de uma transmissão do primeiro segmento no canal 1 e escalonar a escuta dos outros

canais, afim de receber os dados restantes do vídeo. Conforme descrição feita em [16], as técnicas de difusão periódica podem ser classificadas em 3 tipos: O primeiro grupo propõe uma técnica que divide o fluxos em canais segmentos de tamanhos incrementais e os transmite em canais de mesma largura de banda. Os segmentos menores são transmitidos mais freqüentemente que os maiores. Em [33], é definido o esquema *Pyramid Broadcasting*, onde o tamanho dos segmentos segue a distribuição geométrica e um canal é usado para transmitir fluxos diferentes. A taxa dos canais deve ser suficiente para a entrega, em tempo hábil, do fluxo de dado e o cliente deve ter banda e capacidade de armazenamento suficientes para estes fluxos. O problema principal desta técnica é a grande quantidade de recursos requeridos no cliente. Para minimizar este problema foi proposta a técnica *Permutation-Based Pyramid Broadcasting* [6], onde a idéia é multiplexar um canal em  $k$  subcanais de menor taxa.

Outra técnica, denominada de *Skyscraper Broadcasting* [25], propõe que cada segmento seja continuamente transmitido à taxa do vídeo em um canal. A série de tamanhos de segmentos é 1,2,2,5,5,12,12,25,25,52,52,... sendo o maior deles de tamanho  $W$ . A Figura 3.4 mostra a chegada de duas requisições de clientes: a primeira um pouco antes do terceiro segmento e outra antes do décimo oitavo segmento transmitido no canal 1. O escalonamento da transmissão está representado em cinza e preto e é feito de forma que o cliente recebe os dados em no máximo dois canais. O espaço máximo de *buffer* do cliente é igual ao tamanho do maior segmento.

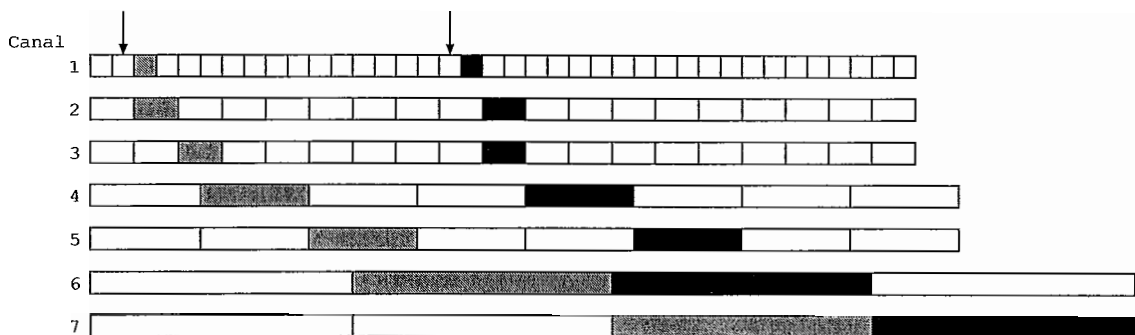


Figura 3.4: *Skyscraper*

Para as técnicas apresentadas acima, a banda requerida no servidor é igual ao

número de canais e é independente da taxa de chegada de clientes, mas são eficientes quando a taxa de chegada é alta.

Para melhorar a técnica *Skyscraper Broadcasting*, foi proposto um novo mecanismo, denominado *Dynamic Skyscraper* [17]. Essa nova proposta considera popularidades dinâmicas para os vídeos e assume taxas baixas de chegadas de clientes. Nesta técnica é alterado dinamicamente a taxa de transmissão do vídeo que está sendo enviado em um canal, onde a cada  $W$  unidades de tempo no canal 1 e de acordo com as requisições dos cliente um vídeo diferente é transmitido. As requisições dos clientes são escalonadas para o próximo grupo livre a ser transmitido, usando a disciplina FIFO (*First In First Out*) para decidir a ordem de atendimento. Em [12] foi proposto uma nova progressão de segmentos para prover serviço imediato ao cliente. A banda requerida no servidor para a transmissão do fluxo  $s_i$ , considerando chegadas *Poisson* com taxa média  $\lambda_i$ , é apresentada em [12]:  $B_{DynSky} = 2U\lambda_i + (K - 2)/(1 + 1/\lambda_i WU)$ , onde  $U$  é a duração da unidade de segmento,  $W$  é o tamanho do maior segmento e  $K$  é o número de segmentos na progressão de tamanhos de segmentos.

O segundo grupo, denominado *Harmonic Broadcast*, divide o vídeo em segmentos de tamanhos iguais e os transmite em canais de largura de banda decrescentes. O terceiro grupo propõe um método que usa em conjunto as abordagens anteriores, onde o esquema é a junção das técnicas *Pyramid Broadcast* e *Harmonic Broadcast*.

### 3.3 Estratégias baseadas no uso de proxy

O uso de servidores *proxies* em um ambiente de transmissão de mídia contínua traz algumas vantagens, como a diminuição do atraso inicial para a exibição da mídia e a diminuição da carga sobre o servidor, aumentando desta forma a escalabilidade da aplicação.

Uma das questões no uso do *proxy* é qual mídia armazenar. Para isso, existem várias técnicas desenvolvidas na literatura: a primeira delas é baseada na compressão de vídeos em camadas, onde tem-se uma camada base e camadas adicionais

para o armazenamento desse conteúdo no *proxy* [32]. Outra técnica é baseada no armazenamento de uma parte do vídeo no proxy [36]. Existem trabalhos, também, que usam o servidor *proxy* em conjunto com as técnicas de compartilhamento de banda [38, 4, 23].

Uma abordagem é apresentada em [32], onde um mecanismo de camadas de vídeo é usado em conjunto com um controle de congestionamento e uma técnica de adaptação de qualidade. Baseado na popularidade dos vídeos é determinado quais vídeos serão armazenados no *proxy*, onde os mais populares terão mais camadas armazenadas. Esta proposta tem como limitação a necessidade de implementação do controle de congestionamento e do mecanismo de adaptação da qualidade em todas as transmissões entre os clientes e *proxies* e entre *proxies* e os servidores.

Foi proposto em [36], um esquema para o armazenamento dos quadros iniciais do vídeo no proxy, chamado de *prefix*. Este trabalho se motivou na observação da queda de desempenho das aplicações de mídia contínua quando submetidas a algumas características da Internet, como perda, atraso, entre outras. No esquema proposto o cliente solicita um fluxo ao *proxy* que envia imediatamente o *prefix* ao cliente e solicita ao servidor o restante do fluxo. Esta técnica utiliza dois *buffers*, um para o *prefix* e outro temporário para o fluxo vindo do servidor.

O trabalho [23] estuda o uso de *proxy*, com o armazenamento de *prefix*, com a técnica de difusão periódica. A técnica propõe o uso do *Patching* para a transmissão do *prefix* do *proxy* para o cliente e difusão periódica para o envio do restante do vídeo. São feitas pequenas modificações no *Patching* e na difusão periódica para que o número de canais do cliente não seja maior que dois. A alocação do *buffer* é baseada em um algoritmo que visa minimizar o uso da rede entre o *proxy* e o servidor. Os resultados mostram que a alocação ótima dos *buffers* melhora o desempenho do esquema onde estes *buffers* são divididos igualmente, sem considerar o tamanho dos vídeos.

Em [38], é feito um estudo sobre as técnicas de compartilhamento orientadas a pedidos com o uso de *proxy*. Os cenários avaliados são: *Unicast suffix batching* (*SBatch*), *Unicast patching with prefix caching* (*UPatch*), *Multicast patching with*



*prefix caching* (*MPatch*) e *Multicast merging with prefix caching* (*MMerge*). Em todos os cenários o caminho servidor-*proxy* é *unicast*.

O *SBatch* é um esquema que permite ao cliente exibir o vídeo instantaneamente, pois o *prefix* armazenado no *proxy* é transmitido a pedido do cliente. Este esquema é apropriado para ambientes onde o caminho *proxy*-cliente é somente *unicast*.

Quando um cliente solicita um vídeo, o *proxy* lhe transmite imediatamente o *prefix*. Neste momento, é escalonada a transmissão do *suffix* do servidor para o *proxy*, o mais tarde possível, mas garantindo a continuidade no vídeo. O primeiro quadro do *suffix* é escalonado para chegar ao *proxy* no tempo  $v_i$ , onde  $v_i$  é a duração do *prefix*. Para todas as requisições feitas entre  $(0, v_i]$ , o *proxy* encaminha um único *suffix* para o novo cliente. Com isso, nenhuma nova transmissão do *suffix* tem que ser feita pelo servidor no intervalo  $(0, v_i]$ .

O esquema *UPatch* pode ser usado no contexto *unicast* se o *proxy* encaminhar uma cópia para cada um dos clientes. Quando um cliente chega, o *proxy* envia o *prefix* e escalona o envio do *suffix*, como no *SBatch*. Suponhamos que um novo cliente chegue no tempo  $t_2$ ,  $v_i < t_2 < L$ , onde  $L$  é a duração do vídeo, como mostra a Figura 3.5. Neste caso o *proxy* pode escalonar a transmissão do *suffix* completo para tempo  $t_2 + v_i$ , ou transmitir um *patch* de  $[v_i, t_2]$  do *suffix*, desde que o segmento  $[t_2, L]$  seja enviado imediatamente. Com o *patch*, o cliente irá receber dois fluxos simultâneos. A decisão de transmitir o *suffix* completo ou o *patch* depende do *suffix threshold*. Se o novo cliente chegar antes do *suffix threshold*, então ele recebe o *patch*, caso contrário ele recebe o *suffix* completo.

O *MPatch* é uma técnica que permite ao *proxy* usar o esquema de transmissão *multicast* para os clientes.

Na chegada do primeiro cliente, este recebe imediatamente o *prefix* em uma transmissão *multicast*. O servidor inicia a transmissão do *suffix* no tempo  $v_i$  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.

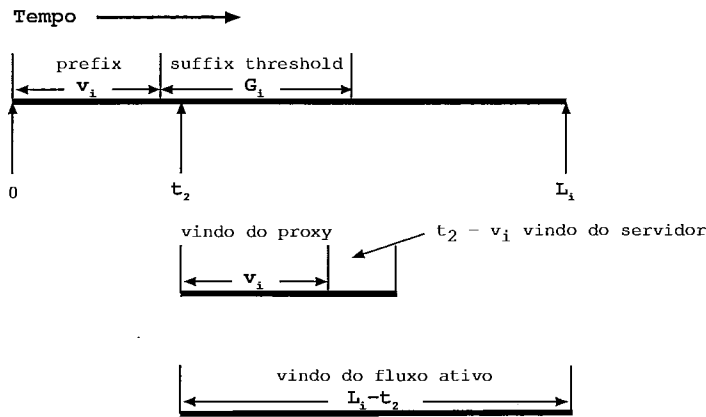


Figura 3.5: UPatch

Como mostra a Figura 3.6, suponhamos que um novo cliente chegue no intervalo  $0 < t_2 < T_i$ , onde a entrega do vídeo pode ser classificada em dois tipos: no caso 1, quando  $T_i < v_i < L_i$ , o cliente recebe do *proxy* o segmento  $[0, t_2]$ , via *unicast*, em um fluxo separado e o segmento  $(t_2, L_i]$  do último fluxo *multicast*; no caso 2, quando  $0 < v_i < T_i$ , temos duas possibilidades: a primeira, quando  $0 < t_2 < v_i$ , o mecanismo é o mesmo do caso anterior (onde  $T_i < v_i < L_i$ ), e a segunda, quando  $v_i < t_2 < T_i$ , o cliente recebe do *proxy* o segmento  $[0, v_i]$  por um canal separado via *unicast* e o segmento  $(t_2, L_i]$  do último fluxo *multicast*. O segmento  $(v_i, t_2]$  é transmitido pelo servidor, via *proxy*, usando *unicast*.

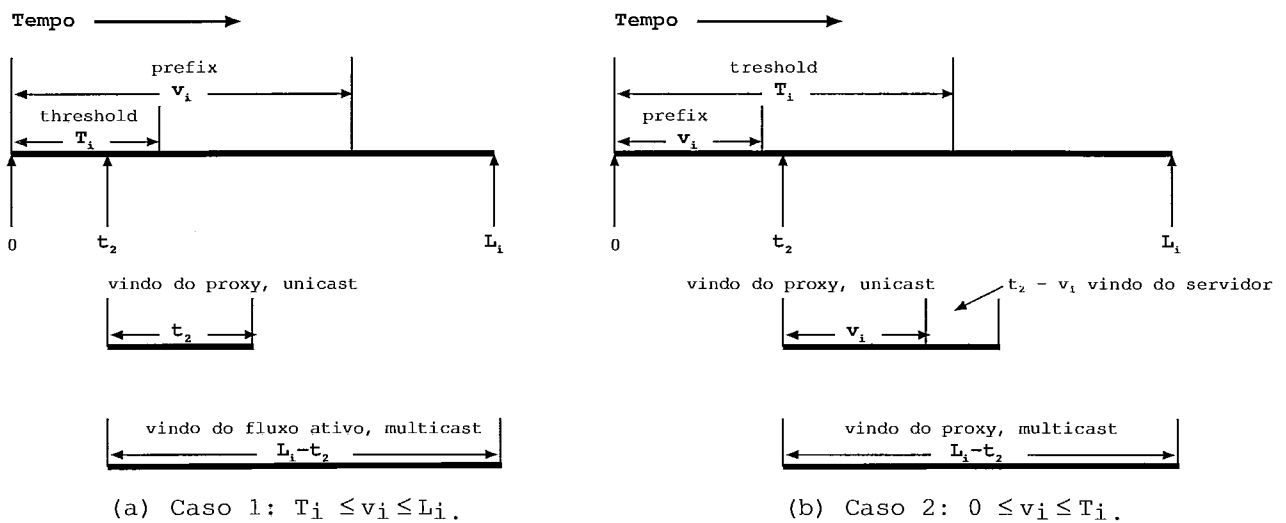


Figura 3.6: MPatch

No cenário *MMerge* foi utilizada a técnica *Closest Target* [19] no caminho *proxy*-cliente. Esta técnica escolhe o fluxo mais próximo ativo no sistema como sendo o próximo fluxo a ser alcançado. A *MMerge* integra o *proxy caching* com o *stream merge*. Para o segmento de um vídeo, requisitado pelo usuário, se o *proxy* contiver este segmento, ele é transmitido diretamente do *proxy* para o cliente. O *suffix* que não está no *proxy* é transmitido do servidor para o cliente, o mais tarde possível, para garantir a qualidade na exibição pelo cliente. Para todos os cenários são definidas expressões para o custo da transmissão. Em [38] é desenvolvido um método analítico para a determinação da alocação ótima do *prefix* no *proxy* e mostrou que essa alocação reduz o custo de transmissão do vídeo.

Em [4] é feito um estudo de vários cenários e são determinados os custos de transmissão para cada um deles. Os cenários analisados são:  $BWSkim(b)$ ,  $BWSkim/U(b)$  e  $BWSkim[/U]+Batch(b)$ . No cenário  $BWSkim(b)$  tanto o servidor principal quanto os servidores *proxy* utilizam a técnica *bandwidth skimming*, com a banda do cliente igual a  $b$ . Em  $BWSkim/U(b)$ , o servidor principal transmite para os *proxies* usando apenas *unicast* e os *proxies* usam *bandwidth skimming* para os clientes. No terceiro cenário ( $BWSkim[/U]+Batch(b)$ ), para cada um dos sistemas acima ( $BWSkim(b)$ ,  $BWSkim/U(b)$ ), se o *proxy* armazena um *prefix*, os clientes usam banda extra, ou reduzem a banda usada para escutar o fluxo do *proxy* e utiliza a técnica *batching* para compartilhar um único fluxo do *suffix* vindo do servidor. Neste trabalho, foram desenvolvidos modelos de otimização de custo total de transmissão que mostraram que o armazenamento no *proxy* deve ser do vídeo completo ou nenhuma parte, para um menor custo na transmissão.

## 3.4 Interatividade

Nas seções anteriores, foram abordadas várias técnicas para a diminuição da largura de banda necessária na transmissão de vídeo sob demanda para um conjunto de clientes, sem que haja prejuízo, ou seja, falha, lentidão ou aceleração que sejam perceptíveis para o usuário do sistema. Cada uma delas, à sua maneira, tenta reunir os usuários em grupos, para que estes possam receber o mesmo fluxo de dados. A

exibição da mídia é seqüencial, ou seja, o usuário começa a assistir o vídeo desde o início até o fim, entre outras características.

Este trabalho tem como foco principal a transmissão de vídeo sob demanda no escopo de educação à distância. Neste tipo de aplicação, os usuários não são tão “comportados” quanto pessoas assistindo a um filme, pois, em uma aula, o aluno deseja executar várias funções como: pausar, parar, retroceder e avançar. A execução dessas aplicações vai de encontro à premissa de que o cliente deve assistir ao vídeo do início até o fim. Essas funções de interatividade do cliente são essenciais para o bom entendimento da aula por parte do aluno, pois poderá ser feito o retorno de uma determinada parte que ele não entendeu.

Em [26], é determinado analiticamente um limite inferior para a banda requerida, considerando fixo o tamanho das requisições, levando em conta acesso não seqüencial para os serviços imediatos e de difusão periódica (serviço com atraso). Esse limite independe da técnica utilizada. Outras contribuições, encontradas neste trabalho, são as simulações para a determinação da banda requerida, onde são avaliados três cenários: o primeiro deles considera que os clientes possuem banda ilimitada e recebem os dados futuros de outros fluxos em andamento o máximo de vezes possível e caso algum dado não seja conseguido desta forma, este dado é transmitido o mais tarde possível, para que outros clientes possam se beneficiar desta transmissão. Outros requisitos desta simulação foram que os pedidos dos clientes são feitos por fragmentos de tamanho fixo do vídeo e que o serviço é imediato.

O segundo cenário, descrito em [26], altera a hipótese do primeiro onde este diz que o tamanho dos segmentos é fixo. O objetivo é saber se a banda requerida no servidor é afetada quando o tamanho deste segmento varia. Foram utilizadas duas distribuições para determinar o tamanho dos segmentos: uniforme e pareto. Os resultados encontrados mostraram que a diferença da banda requerida no servidor, encontrada quando o tamanho do segmento requisitado é fixo ou variável, é mínima. O terceiro cenário, analisa o serviço não imediato, onde foram geradas uma seqüência de requisições, cada uma com um atraso  $d$  e que durante este atraso, dados futuros podem ser buscados através de fluxos correntes. Os resultados obtidos mostram que

a banda requerida fica próxima ao resultado analítico.

No trabalho [37], são estudados dois casos onde o benefício da utilização dos protocolos de compartilhamento de banda pode ser prejudicado. No primeiro caso, os clientes requisitam toda a mídia e podem tolerar elevados atrasos iniciais. Neste caso, o objetivo é estabelecer a banda mínima requerida no servidor para protocolos de *download*, pois neste caso os protocolos de *streaming* já possuem banda mínima definida em outros trabalhos como [26].

No segundo caso, os clientes fazem requisições de tamanhos menores que o comprimento total da mídia, pois o fato do cliente solicitar a mídia completa favorece o compartilhamento dos fluxos de dados. Outra característica analisada é que as solicitações dos clientes podem iniciar ou terminar em qualquer ponto da mídia e são examinados quatro cenários: o primeiro considera que os intervalos de requisições começam no início da mídia e terminam em um ponto aleatório. O segundo cenário considera que esses intervalos iniciam em um ponto aleatório e terminam no fim da mídia. Já o terceiro leva em conta que os intervalos de requisições tem início e fim aleatórios e o quarto cenário divide a mídia em diversas marcas e considera que as requisições serão feitas entre marcas.

Para o primeiro caso (comparação da técnica de *streaming* com protocolos de *download*) são derivadas equações da banda mínima requerida no servidor para os protocolos de *download*. O trabalho conclui que o ganho com o uso de protocolos de *streaming* em relação aos de *download*, quando o cliente pode tolerar atrasos elevados no início da exibição, é pequeno em relação a banda requerida no servidor. Para o segundo caso, também são derivadas equações para a banda mínima requerida no servidor, onde o trabalho conclui que mesmo para cenários onde os protocolos de compartilhamento de recursos não são favorecidos (quatro cenários apresentados acima), estes protocolos geram uma economia de banda considerável em relação aos protocolos que utilizam *unicast* para envio dos dados.

No trabalho [30] é proposta a técnica de *Best Effort Patching* que tem como objetivo o tratamento das operações de interatividade utilizando como base a técnica *Patching*. De uma forma geral a idéia do algoritmo é que são transmitidos periodi-

---

camente, em diversos canais, fluxos completos da mídia. Um cliente ao chegar no sistema deve escolher aquele fluxo que mais se aproxima da posição que ele deseja assistir. Em um exemplo simples o primeiro cliente a escutar um determinado fluxo  $S$  recebe o seu *Patch* via *multicast* e os demais clientes ao invés de solicitarem um *Patch* do fluxo principal  $S$ , solicitam *Patch* do fluxo iniciado pelo primeiro cliente. Assim, temos que os próximos clientes buscam se juntar com o primeiro cliente e logo após, a união deles busca o fluxo principal  $S$ .

## Capítulo 4

# O novo cliente do servidor RIO e a proposta do *Patching* Interativo

NESTE capítulo serão descritas as características adicionadas, por este trabalho, ao cliente de visualização de vídeos do servidor RIO, o *RioMMClient*, bem como a técnica de *Patching* Interativo (PI). Será feita primeiramente, uma apresentação geral da técnica de *Patching* Interativo seguindo uma descrição das principais características do novo cliente do servidor RIO bem como uma descrição do módulo de *Patching* Interativo desenvolvido.

### 4.1 Visão geral do *Patching* Interativo

O objetivo do uso de técnicas de compartilhamento de recursos é a diminuição da banda requerida no servidor para a transmissão de vídeo. O mecanismo proposto se baseia no *Patching* que foi apresentada no capítulo 3. Esta técnica foi escolhida por sua simplicidade e por apresentar um bom desempenho para acesso seqüencial. A banda média requisitada pelo *patching* e outras técnicas mais sofisticadas, baseadas no união hierárquica, é semelhante para vídeos com popularidade baixa e média [12].

Quando do recebimento de uma mensagem de movimentação do cliente o PI executa alguns passos para buscar o melhor grupo para encaixar este cliente que se movimentou. Utilizando o conceito da janela ótima do *Patching*, descrito na

seção 3.1, o PI tentará incluir esse novo cliente em algum grupo já existente, para compartilhar o máximo de recursos possível. Dois casos devem ser considerados: (a) o cliente solicita um bloco fora da janela de *patching* e (b) o cliente solicita um bloco que pertence a janela de *patching*.

Procedimento executado para o caso (a):

É verificado se existe algum grupo ativo cujo último bloco transmitido esteja próximo ao bloco solicitado pelo cliente que se moveu. Para que o cliente se junte a um grupo ativo é necessário que o intervalo de tempo entre o bloco solicitado por ele ( $b_{move}$ ) e o último bloco transmitido para o grupo ( $b_{grupo}$ ) esteja dentro de certos limites. Definiremos  $\delta_{before}$  como sendo o intervalo máximo entre o bloco  $b_{grupo}$  e o  $b_{move}$  dado que  $b_{move} > b_{grupo}$  e  $\delta_{after}$  é o intervalo máximo entre  $b_{grupo}$  e  $b_{move}$  dado que  $b_{move} < b_{grupo}$ . A Figura 4.1 ilustra duas janelas  $\delta_{before}$  e  $\delta_{after}$  definidas para um determinado vídeo. Estas janelas foram definidas com o objetivo de tentar agrupar ao máximo os clientes para minimizar o uso da rede.

Primeiramente é verificado se existe algum grupo cujo último bloco transmitido esteja dentro da janela  $\delta_{before}$ . Se existir, o cliente se juntará a este grupo (caso 1). Caso não exista, é procurado um grupo cujo último bloco transmitido esteja dentro da janela  $\delta_{after}$  (caso 2). A pesquisa é feita inicialmente na janela  $\delta_{before}$  pois caso o cliente se junte a este grupo não será necessário o envio de blocos unicast (*patch*) como será descrito a seguir.

Procedimento executado para o caso (b):

Se existir uma janela de *patching* ativa e o cliente solicitar um bloco anterior ao próximo bloco a ser enviado para o grupo, o cliente se juntará a este grupo. O tratamento deste cliente é igual ao de um cliente novo com a diferença de que o primeiro bloco a ser enviado para ele, através do patch, pode ser diferente do primeiro bloco do vídeo. Chamaremos este de caso 3. Se existir uma janela de *patching* ativa e o cliente solicitar um bloco posterior ao próximo bloco a ser enviado ao grupo, é verificado se o bloco solicitado está dentro da janela  $\delta_{before}$ . Se existir o



cliente se juntará a este grupo. Este caso é semelhante ao caso 1 descrito acima.

Caso não seja encontrado nenhum grupo nos casos 1, 2 e 3 para este cliente se unir, é criado um novo grupo para ele (caso 4). Após qualquer um dos casos o PI envia uma mensagem `msgcode_IP` a este cliente para que ele se una ao novo grupo *multicast* e inicie a exibição dos dados. Nos casos 2 e 3, o cliente recebe os dados do grupo ao qual ele se uniu e solicita o *patch* dos blocos já transmitidos ao grupo. No caso 1, o cliente apenas “escuta” o fluxo *multicast* e exibe imediatamente os dados recebidos. Neste caso não existe a necessidade do envio de *patch* pois o bloco solicitado pelo cliente é posterior ao enviado para o grupo ao qual ele se juntou. O valor do `delta_before` deve ser pequeno de forma que o cliente não perceba que está recebendo uma parte do fluxo anterior à que ele solicitou. No caso 4 também não há a necessidade de *patch*, pois o cliente é o líder do grupo e irá solicitar os dados que ele necessita.

Quando o PI encontra mais de um grupo no qual o cliente pode entrar, a escolha do grupo é feita da forma descrita a seguir. Seja  $b_{grupo}^i$  o último bloco transmitido para o grupo  $i$  e  $b_{move}$  o bloco solicitado pelo cliente que se moveu. O grupo selecionado é aquele que satisfaz o critério:

$$\min_{\forall i} |b_{grupo}^i - b_{move}|$$

No caso de  $b_{grupo}^i$  pertencer ao intervalo `delta_after`, o grupo selecionado é aquele em que o cliente receberá o menor *patch*. No caso  $b_{grupo}^i$  pertencer ao intervalo `delta_before` é selecionado o grupo mais próximo à posição de movimentação do cliente. Na Figura 4.2 é apresentado o algoritmo *Patching*\_Interativo, que descreve as decisões tomadas pelo PI quando uma movimentação de cliente é notificada.

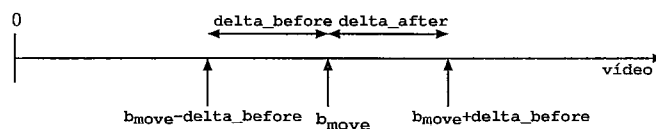


Figura 4.1: Janelas definidas para que o cliente se junte a um grupo

```

Patching_Interativo( 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;
  distância_delta_before = INFINITO;
  distância_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 ] ) < distância_delta_before )
        distância_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 ) < distância_delta_after )
            distância_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 nenhum grupo para o cliente
      Cria novo grupo para o cliente
Fim-Patching_Interativo

```

Figura 4.2: Algoritmo Patching\_Interativo

## 4.2 *RioMMClient* inicial

O *RioMMClient* é o cliente de visualização de vídeos do RIO. Além de vídeos, diversas outras funções são exercidas por este cliente, como exemplo, podemos citar a reprodução de músicas, exibição e sincronização de transparências. A Figura 4.3 mostra os componentes do cliente: comunicação interface gráfica, interface servidor RIO, interface *netscape*, interface TGIF (*Tangram Graphic Interface Facility*) [11], interface *MPlayer* [21] e um *buffer* de armazenamento de dados.

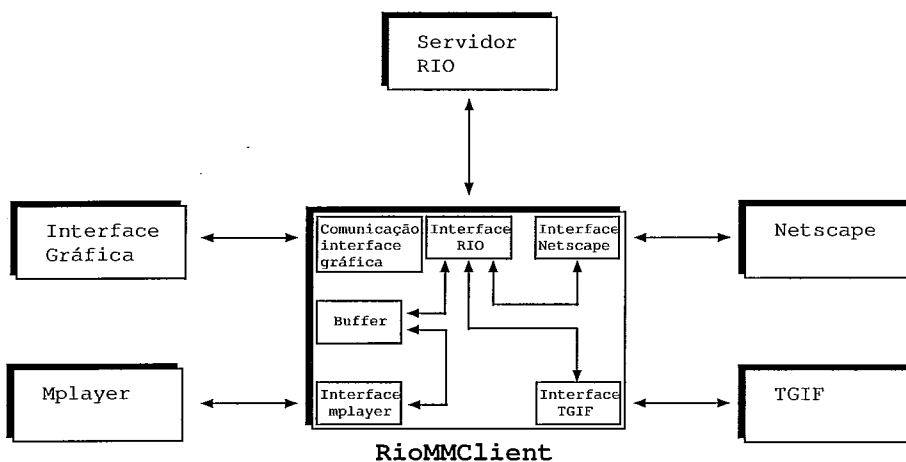
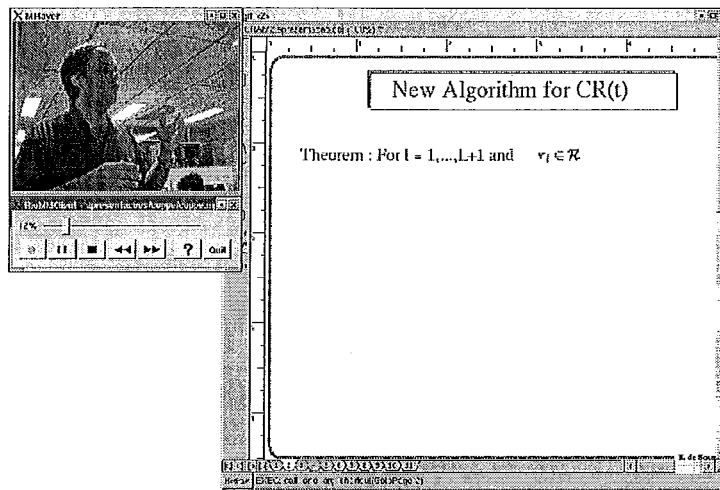


Figura 4.3: Componentes do *RioMMClient*

### Comunicação interface gráfica

Este componente é responsável pela comunicação entre o cliente e a interface gráfica. Os comandos de exibir, parar, pausar, avançar e retroceder são passados da interface gráfica para este módulo. Nele os comandos são tratados e repassados ao módulo responsável pela execução, para que este possa realizar as funções apropriadas, como, por exemplo, solicitar ao servidor dez blocos a frente do ponto atual, no caso do comando avançar.

A interface gráfica do *RioMMClient*, junto com o *MPlayer* e o TGIF, podem ser visualizadas na Figura 4.4.

Figura 4.4: Interface gráfica *RioMMClient*

## Interface servidor RIO

Este módulo faz a comunicação entre o cliente e o servidor. Os pedidos de abertura de sessão, fluxo e objeto são executados por ele. Outra função do módulo é o envio dos pedidos de blocos ao servidor e recepção dos dados para armazenamento no *buffer* do cliente.

## Interface *MPlayer*

A comunicação com o *MPlayer* é feita por este componente. Os blocos de dados são copiados para o tocador através de um *pipe* (como mostra a Figura 4.5), onde o cliente é responsável pela escrita e o *MPlayer* pela leitura.

## Interfaces *Netscape* e *TGIF*

Estas interfaces são responsáveis pela comunicação do cliente com os aplicativos *Netscape* e *TGIF*, respectivamente. Comandos são enviados aos aplicativos para que possam alterar as transparências, de acordo com o andamento do vídeo, mantendo, também, a sincronia quando o usuário executa comandos de avançar ou retroceder.

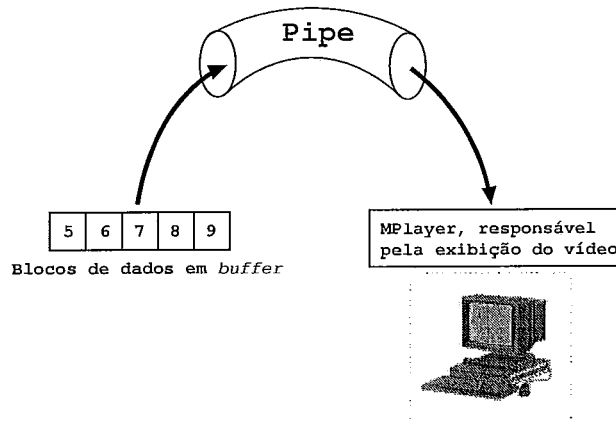


Figura 4.5: Utilização de um *pipe*

### *Buffer* de armazenamento de dados

O *RioMMClient* possui um *buffer* para compensar as variações da rede (*jitter*) e todos os blocos de dados, vindos do servidor, são armazenados neste *buffer* antes de sua exibição.

Como pode ser observado na Figura 4.6, a primeira etapa para a exibição de um vídeo é a solicitação dos blocos iniciais para o preenchimento do *buffer* (blocos 1, 2 e 3) e a espera da chegada destes dados. Após a chegada dos blocos inicia-se a exibição com o envio do primeiro bloco de dados ao *MPlayer*. Para a substituição do bloco 1, é enviado o pedido do bloco 4 ao servidor. Este processo se repete até que o usuário execute alguma ação na interface gráfica, como pausar, avançar, retroceder e parar.

O botão de pausar (*pause*) faz a imagem congelar e o cliente não solicita blocos ao servidor até que seja dado o comando de visualizar (*play*), fazendo com que a exibição continue a partir do momento do congelamento. A ação do botão de parar (*stop*) se assemelha a ação de pausar, porém no parar, o vídeo é exibido desde o início (bloco de dados inicial). Quando o cliente executa a função avançar (*forward*) ou retroceder (*rewind*) é solicitado ao servidor os blocos correspondentes a posição que cliente se moveu e no momento da chegada dos dados é feito o envio ao *MPlayer*, para o prosseguimento da exibição. O cliente *RioMMClient* não possui função de *FastForward* ou *FastRewind* nas quais o vídeo continua sendo exibido durante a

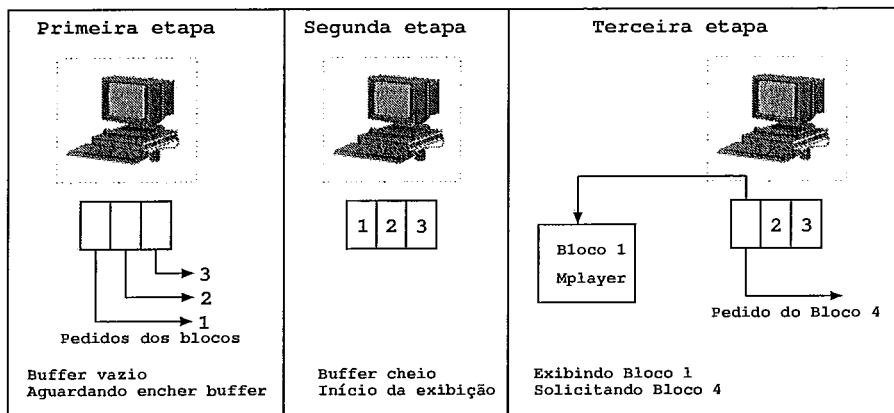


Figura 4.6: Etapas iniciais do *RioMMClient*

interação do usuário.

O cliente também possui exibição e sincronização com transparências, como pode ser visto na Figura 4.4. Para a exibição das transparências, o *RioMMClient* solicita ao servidor, quando disponível, um objeto que contém as transparências e outro com informações sobre a sincronização com o vídeo.

### 4.3 Características adicionadas ao *RioMMClient*

Foram implementadas algumas funcionalidades no *RioMMClient* para diminuir a utilização da largura de banda do servidor e reduzir o tráfego na rede. Essas funcionalidades estão diretamente ligadas ao *Patching* Interativo implementado no servidor RIO, que será descrito no capítulo 4.4. O PI é o módulo do RIO que provê um compartilhamento dos fluxos de dados enviados aos clientes. Serão apresentadas a seguir a arquitetura do novo cliente, o seu algoritmo de funcionamento e o protocolo de comunicação com o servidor. O usuário, através de um parâmetro do *RioMMClient*, decide se o cliente irá se conectar diretamente ao servidor RIO ou se a conexão será feita através do PI.

### 4.3.1 Arquitetura do novo cliente

Pode ser visto na Figura 4.7 os módulos *CommMulticast*, *BufferStream* que foram adicionados ao cliente. Além destes módulos, a interface de comunicação com o servidor foi alterada para propiciar o recebimento de dois fluxos de dados simultâneos, um *unicast* e outro *multicast*.

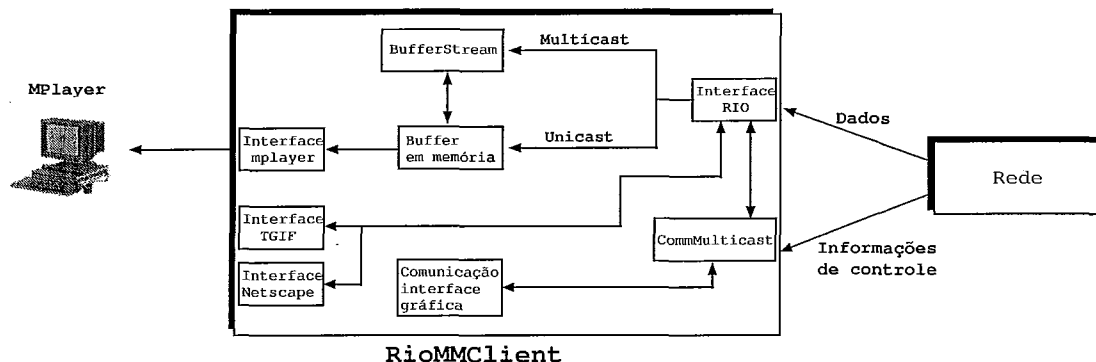


Figura 4.7: Arquitetura do cliente

A interface do cliente com o servidor foi modificada, pois previa um único fluxo *unicast* para cada cliente. Para a utilização do PI, como também do *Patching* original, é necessário que o cliente tenha a capacidade de receber dois fluxos de dados simultaneamente, um para o *patch* e outro para o fluxo *multicast*. O cliente deve fazer os pedidos dos blocos de dados que compõe o *patch* para que o servidor possa enviá-los, via *unicast*. Este fluxo terá duração a duração do *patch*. No outro fluxo, o cliente receberá, via *multicast*, os blocos de dados que estão sendo enviados para o grupo do qual ele faz parte. A nova interface de comunicação suporta também a mudança de grupo *multicast*, para que o cliente, ao fazer um movimento (avanço, retrocesso, pausa ou parada), possa escutar a transmissão de outro grupo *multicast*. Os procedimentos relativos a escolha do grupo *multicast* e mudança de grupo serão detalhados no capítulo 4.4, onde serão descritas as características inseridas no servidor RIO neste trabalho.

Como mencionado anteriormente, o cliente recebe dois fluxos de dados simultaneamente, que devem ser tratados pelo cliente para que não haja sobreposição de dados que venha a ocasionar a perda de uma determinada parte do vídeo. Foram

definidos então, dois *buffers*: *buffer* no disco e *buffer* em memória. O *buffer* no disco armazena os dados que serão exibidos no futuro e o *buffer* em memória os dados que serão exibidos imediatamente. Antes de fazer o pedido de qualquer bloco ao servidor, o cliente verifica se aquele dado já está armazenado no *buffer* em disco; em caso positivo, ele copia deste para o *buffer* em memória. Todos os dados recebidos, via *multicast*, são diretamente copiados para o *buffer* em disco e os dados que chegam através de *unicast* (*patch*), são copiados diretamente para o *buffer* em memória, pois precisam ser exibidos imediatamente e logo após copiados para o disco.

O procedimento para armazenamento dos dados é definido no algoritmo *ArmazenamentoBuffer*, visualizado na Figura 4.8, que tem os seguintes parâmetros de entrada:

- *dados* é a variável que contém o bloco de dados vindos do servidor que serão armazenados no cliente;
- *tráfego* é a variável que indica se os dados foram recebidos via *unicast* ou *multicast*;
- *bufferstream* é o vetor que contém os dados de todos os blocos já recebidos pelo cliente e armazenados em forma de arquivo em disco;
- *membuffer* é o *buffer* de comunicação do cliente com o *player*;
- *bloco* é a variável que mostra qual bloco foi recebido;
- *numero\_membuffers* é o total de posições no *membuffer*;
- *numero\_blocos\_recebidos* é o número total de blocos recebidos até o momento presente.

Foi definido, também, que todo grupo de clientes possui um líder que é responsável pelos pedidos dos blocos de dados que serão enviados, via *multicast*, para o grupo, como mostra a Figura 4.9. O algoritmo geral de funcionamento do novo cliente está definido na Figura 4.10, onde o cliente, enquanto houver blocos no *buffer* em memória, exibe os dados e solicita ao PI os novos blocos, de acordo com sua condição, líder ou não.



```

ArmazenamentoBuffer( dados, tráfego, bloco )
se( tráfego == unicast )
    membuffer[ bloco%numero_membuffer ] = dados;
fim-se
bufferstream[ numero_blocos_recebidos ] = dados;
numero_blocos_recebidos++;
fim-ArmazenamentoBuffer

```

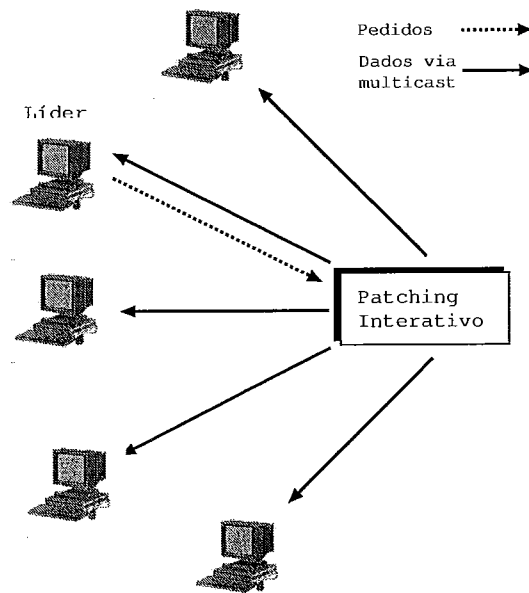
Figura 4.8: Algoritmo *ArmazenamentoBuffer*

Figura 4.9: Pedidos dos blocos de dados

```

Cliente
LendoDoBufferStream = FALSO;
enquanto houver blocos no Buffer em memória
    ExibirBloco( BlocoASerExibido );
    BlocoASerSolicitado = BlocoASerExibido + NúmeroBuffers;
    se( BlocoASerSolicitado está no BufferStream )
        Copia para o Buffer em memória;
        LendoDoBufferStream = VERDADEIRO;
        se( Líder == VERDADEIRO )
            // Entrei no estado lendo do buffer local em disco
            Enviar ao PI msgcode_MOVE( action_play, -1 );
            LendoDoBufferStream = FALSO;
        senão
            se( LendoDoBufferStream == VERDADEIRO )
                // Sai do estado lendo do buffer local em disco
                Enviar ao PI msgcode_Move( action_play, BlocoASerSolicitado );
            senão
                SolicitaBloco( locoASerSolicitado );
        fim-enquanto
fim-Cliente;

```

Figura 4.10: Algoritmo Cliente

### 4.3.2 Protocolo para envio de mensagens de controle entre o cliente e o servidor

O objetivo das mensagens de controle é manter a consistência entre as informações do cliente e do servidor e para a solicitação de uma determinada função. Para executar tal procedimento foi definido um módulo denominado *CommMulticast* que é usado pelo cliente e pelo PI, com a finalidade de prover o serviço de envio e recebimento das mensagens de controle. Estas mensagens são enviadas usando o protocolo TCP.

A *CommMulticast* mantém uma estrutura que contém as informações sobre a conexão. Os dados armazenados nesta estrutura são: PID, *socket* e *first*, onde PID é o valor do PID do cliente na PI, *socket* é o descritor do *socket* do cliente e o *first* indica se o cliente é ou não o líder do grupo. Além desta estrutura foram definidas algumas mensagens para esta comunicação, que são: *msgcode\_IP*, *msgcode\_PID* e *msgcode\_MOVE*. A mensagem *msgcode\_IP* tem como parâmetros o IP *multicast*, a porta que o cliente deve “escutar”, se este cliente é ou não líder do grupo e o bloco que será enviado ao *player*. A mensagem *msgcode\_PID* possui como parâmetro o PID do cliente no PI e a mensagem *msgcode\_MOVE* possui os seguintes parâmetros: bloco e ação, onde bloco é o bloco para o qual ele se moveu e ação indica qual o tipo de movimento o cliente executou, como parar o vídeo (*action\_stop*), tocar (*action\_play*), pausar (*action\_pause*), avançar (*action\_forward*) e retroceder (*action\_rewind*).

#### 4.3.2.1 Cenários

Um esquema de troca de mensagens entre o cliente e o servidor pode ser visto na Figura 4.12 onde o *buffer* do cliente tem tamanho 2 blocos. A troca de mensagens inicial é igual para os casos 1 e 2, onde o cliente inicia fazendo um pedido de abertura de sessão, recebendo como resposta o tamanho do bloco de dados que o servidor está utilizando. Após o recebimento desta mensagem, o cliente solicita a abertura de um fluxo e o servidor retorna o número máximo de pedidos que podem estar ativos no servidor para este cliente. Após este recebimento, é enviado ao servidor

uma requisição de abertura de objetos e o usuário recebe como resposta o tamanho do objeto (em *bytes*) e o PID do cliente no PI. Neste ponto, considera-se que o cliente conectou-se no PI, pois, caso contrário, se o cliente se conectar diretamente ao servidor, receberia apenas o tamanho do objeto. Quando o PID é recebido, o cliente se conecta ao PI, através da mensagem *msgcode\_PID*, informando o PID e recebe como resposta uma mensagem do tipo *msgcode\_IP*, que contém o IP *multicast*, a porta e se ele é ou não o líder do grupo. Neste momento ele entra no grupo *multicast* indicado pelo servidor e faz os pedidos para encher o seu *buffer*. Os procedimentos executados após o recebimento das mensagens podem vistos no algoritmo *RecebimentodeMensagens*, na Figura 4.11.

```

RecebimentodeMensagens( Tipo )
se( Tipo == msgcode_PID )
    Conectar-se ao PI enviando PID;
se( Tipo == msgcode_IP )
    AtualizaLider( Líder );
se( EstaEmAlgunGrupo() )
    se( ( IP_atual != IP ) || ( Porta_atual != Porta ) )
        SairGrupoMulticast();
    EntrarGrupoMulticast( IP, Porta );
se( Bloco != -1 )
    ExibirBloco = Bloco;
    SolicitarBlocos( Líder );
fim-RecebimentodeMensagens;

```

Figura 4.11: Algoritmo *RecebimentodeMensagens*

A partir do momento que o cliente entra no grupo *multicast* e dependendo dele ser ou não o líder do grupo, dois casos podem ocorrer. No caso 1, os blocos iniciais solicitados serão enviados pelo servidor, via *multicast*, para todos os membros do grupo, pois neste caso, o cliente é o líder do grupo. Quando ocorrer o preenchimento do *buffer* inicial, o cliente envia uma mensagem de *CanStart* ao servidor para se certificar que o *buffer* do servidor já está preenchido. Com a confirmação do *CanStart* o cliente inicia a exibição do vídeo e no momento do término da exibição do bloco 1, faz a requisição do bloco 3, que também será enviado para todo o grupo. No caso 2, onde o cliente não é o líder do grupo, os pedidos dos blocos iniciais fazem parte do *patch* e serão enviados via *unicast*. Os blocos restantes serão pedidos a medida em que os blocos forem sendo consumidos pelo *player*. Durante este processo o cliente recebe os dados, via *multicast* (blocos 4, 5 e 6), solicitados pelo líder do grupo e que serão armazenados em disco, conforme descrição feita na seção 4.3.1. Após a



bidos do servidor. Portanto, quando ele executa uma ação que altera a seqüência de exibição, serão solicitados somente os blocos que não estiverem armazenados no cliente. A Figura 4.13 exemplifica um cliente que executa alguns movimentos durante a exibição do vídeo. Inicialmente o cliente assistiu do bloco 1 ao 5 e neste ponto, se moveu para o bloco 15. No instante desta movimentação, foram solicitados do bloco 15 ao 20, supondo que o *buffer* do cliente é igual a cinco posições. Ele assistiu até o 22 quando se moveu para o bloco 30. O vídeo foi exibido até o bloco 34, quando o usuário retornou para o bloco 19. Neste momento, o cliente, verifica que já possui este bloco e solicita ao servidor o bloco 23, pois os blocos que ele possui em seu *buffer* no disco não são suficientes para o preenchimento de seu *buffer* em memória. O servidor tentará encontrar um grupo que esteja próximo ao bloco 23, para que este cliente possa se juntar. Com isso, ele não receberá dados que já tem em *buffer* no disco e estará recebendo dados que assistirá em um futuro breve.

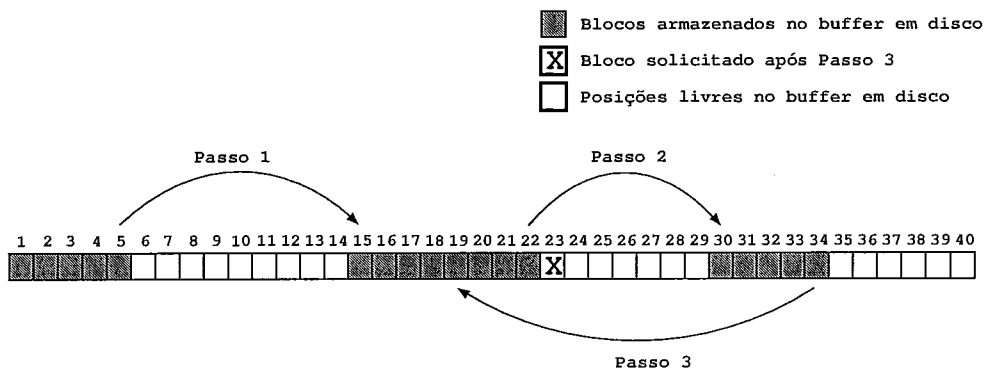


Figura 4.13: Movimentação do cliente

### 4.3.3 Protocolo para envio de dados entre o cliente e o servidor e interface gráfica do riosh

O protocolo para envio de dados da primeira versão do servidor RIO [35] foi implementado usando o protocolo UDP, porém com diversas características do protocolo TCP. A transmissão de dados, seja de vídeo em tempo real (*streaming*) ou para transferência de arquivos (nos caminhos servidor-cliente e cliente-servidor), era controlada por ACK's, ou seja, para cada fragmento enviado, ou conjunto de fragmentos, era esperado pela origem um pacote de ACK para confirmação de re-

cebimento. Caso essa confirmação não chegasse, a origem retransmitia os dados não confirmados. Esse procedimento onera muito o servidor, pois ele deve aguardar ACK's de todos os fluxos ativos. Para melhoria dessa transmissão, foi feita a alteração deste protocolo para que, na transmissão de fluxos de tempo real, o servidor não aguarde ACK's do cliente. O envio dos dados ao cliente é feito da seguinte forma: o bloco de dados é dividido em fragmentos de 1500 bytes e eles são transmitidos ao cliente sem que seja necessário o envio, por parte do cliente, de ACK's de confirmação. Para a transferência de arquivos o protocolo permanece inalterado.

Outra contribuição deste trabalho foi o desenvolvimento de uma interface gráfica para o *riosh*. Esta interface possui as mesmas funções da interface texto porém, com algumas vantagens: é possível executar o cliente de visualização de vídeo diretamente através dela, maior facilidade para o gerenciamento dos objetos do servidor entre outras. O processo acontece da seguinte forma: o usuário abre o *riosh*, loga no servidor RIO, seleciona o vídeo que ele deseja assistir, clica em play e o *riosh* chama o *RioMMClient* para visualizar o vídeo. A interface gráfica do *riosh* pode ser vista na Figura 4.14.

## 4.4 Detalhes da arquitetura e módulo PI

NESTE capítulo será apresentada a proposta do *Patching* Interativo. Esta técnica foi desenvolvida em conjunto com a dissertação de mestrado de Melba Lima Gorza [22]. Este mecanismo visa proporcionar o compartilhamento de fluxo entre os clientes, mesmo quando estes executam operações de VCR que resultam em acesso não seqüencial ao vídeo. Nas seções a seguir será apresentada a abordagem proposta e sua implementação dentro do Servidor RIO (descrito no capítulo 2). A análise de desempenho do algoritmo PI será objeto do capítulo 5.

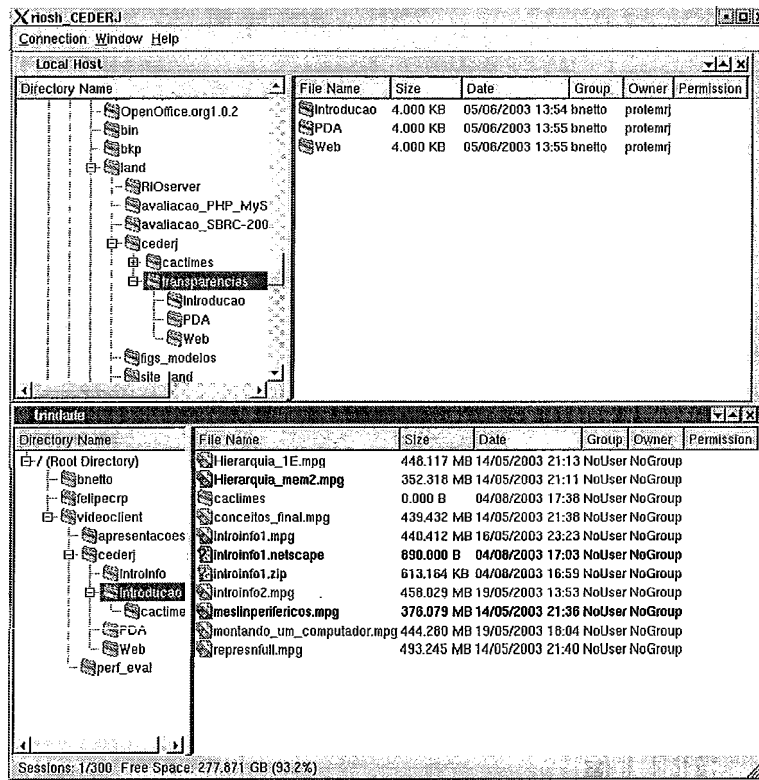


Figura 4.14: Interface gráfica riosh

## 4.5 Visão geral

Como mostra a Figura 4.15, foi implementado um módulo para atuar em conjunto com o servidor RIO, o *Patching* Interativo. Pode-se observar que o cliente escolhe onde deve se conectar: se for diretamente ao servidor, ele receberá todos os dados do vídeo via *unicast* (método definido para a primeira versão do RIO descrito no capítulo 2). Se for através do módulo, ele receberá os dados via *multicast* e apenas o *patch* via *unicast*. O PI recebe os dados do servidor usando *unicast*, pois para o servidor ele é apenas mais um cliente conectado. Com esta arquitetura, o PI pode ser executado em qualquer máquina, não necessariamente na mesma que o servidor.

Para se conectar ao PI, o cliente deve seguir os mesmos passos que executa quando se conecta diretamente ao servidor (procedimento descrito na seção 4.3.2.1). Com isso mantemos a compatibilidade do cliente com o modo de operação anterior. A única diferença é que, na abertura de um objeto, o cliente recebe uma informação adicional, que é o seu identificador no PI. Com esse identificador a *CommMulti-*

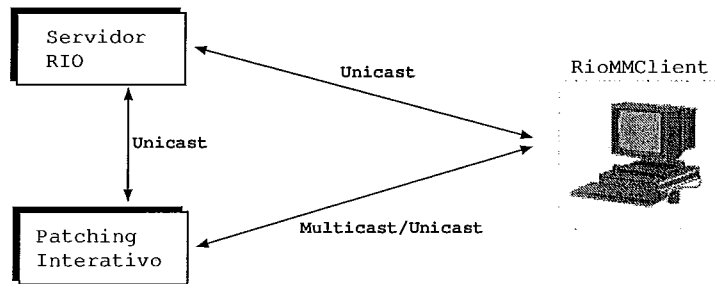


Figura 4.15: Visão geral *patching* interativo

*cast* do cliente se conecta a *CommMulticast* do PI e como resposta recebe o grupo *multicast* no qual ele deverá se unir.

O algoritmo *TratamentoDePedidos*, que pode ser visualizado na Figura 4.16, mostra o funcionamento geral do módulo PI e tem como entrada um pedido do cliente. Os tipos de pedidos que podem ser recebidos pelo PI são cinco: nos casos de abertura de sessão e fluxo, os pedidos são encaminhados ao servidor e a resposta é enviada ao cliente. Para a abertura de objeto o PI envia o pedido de abertura de objeto ao servidor, insere o cliente em suas estruturas ( que serão explicadas na seção 4.6) e solicita os blocos para preenchimento de seu *buffer*. No caso do pedido de *CanStart* o PI verifica se o *buffer* local dele está cheio e logo após esta confirmação, envia pedido de *canstart* ao servidor. Recebendo a resposta o PI envia o resultado ao cliente. No caso das mensagens de pedido de bloco o PI localiza o cliente em sua estrutura, verifica se o bloco já está em *buffer*, se não estiver solicita ao servidor, verifica se aquele cliente é ou não líder do grupo e envia o bloco de dados. A função *SolicitaBloco* é responsável por solicitar os dados para preenchimento do *buffer* do PI e a função *LocalizaCliente* procura o cliente dentro da estrutura *InfoClient* que será descrita posteriormente. A função *EnviaBloco* transmite os dados para todo o grupo, via *multicast*, ou transmite apenas para aquele cliente, no caso do *patch*, via *unicast*.

O módulo PI é dividido nos seguintes componentes: *Buffer*, *SessionManager*, *StreamManager*, *ObjectManager*, *CommMulticast* e *RIOInterface*. O esquema do módulo pode ser visto na Figura 4.17. Estes módulos fazem parte da comunicação tanto com o cliente quanto com o servidor.



```

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

  se( pedido == abertura de fluxo )
    Envia pedido de abertura de fluxo 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 );
    SolicitaBlocos();

  se( pedido == canstart )
    enquanto buffer do PI não está cheio
      aguardar chegada dos blocos
    Envia pedido de canstart ao servidor

  se( pedido de bloco )
    cliente = LocalizaCliente();
    se( bloco solicitado não está em buffer )
      SolicitaBloco();
    //Verificando se devo mandar para o grupo
    se( cliente é o líder do grupo )
      trafego = multicast;
    senão
      trafego = unicast;

    //Envia bloco para cliente
    EnviaBloco( bloco, trafego );
Fim-TratamentodePedidos

```

Figura 4.16: Algoritmo de tratamento de pedidos

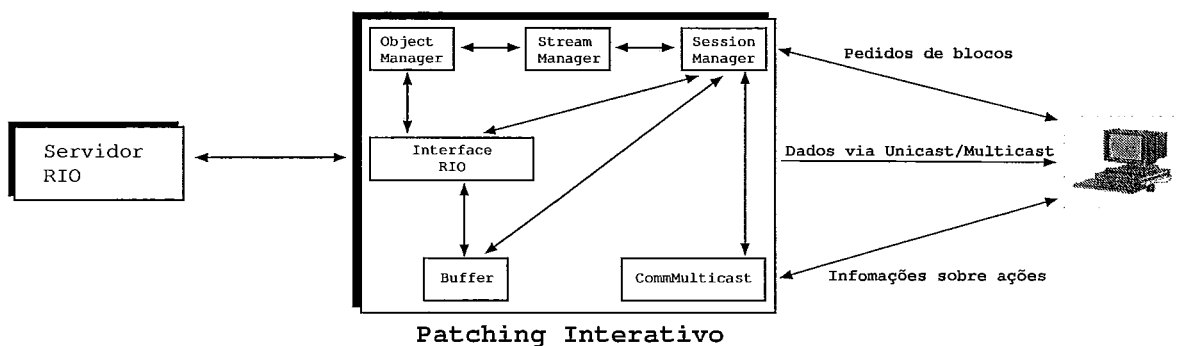


Figura 4.17: Componentes PI

### *Buffer*

O *buffer* é utilizado para o armazenamento dos blocos de dados vindos do servidor antes de seu envio ao cliente. Ele funciona como um *lookahead buffer* pois o PI faz uma antecipação dos pedidos do cliente para diminuir o *jitter* e a latência. Cada cliente possui o seu *buffer* no PI, onde este é utilizado para os dados do *patch*, no caso do cliente não ser o líder e é o *buffer* das informações que serão enviadas, via *multicast*, no caso do cliente ser o líder.

### *SessionManager*

O *SessionManager* é responsável pelo recebimento dos pedidos de blocos feitos pelo cliente e pelo gerenciamento do *buffer* do cliente no PI. Este componente também realiza a tarefa de ler os arquivos de configuração que são: *system.cfg* e *window size.cfg*. O primeiro arquivo contém as informações gerais para inicialização do módulo PI: servidor ao qual o PI deverá se conectar, tamanho do *buffer* dos clientes, além dos valores da janela de movimentação dos usuários. Esta última configuração será descrita mais a frente neste capítulo. O segundo arquivo (*window size.cfg*) contém o tamanho da janela de *patching* de cada vídeo armazenado no servidor. As informações relativas a todos os clientes ficam armazenadas e são tratadas pelo *SessionManager*.

### *StreamManager*

Este componente é responsável pelo tratamento dos pedidos de abertura de objeto e de *canstart*, conforme descrição feita no algoritmo *Patching* Iterativo. O *StreamManager*, é responsável, também, pela abertura e fechamento de fluxo com o servidor RIO, por onde os dados serão transmitidos do servidor para o PI.

### *ObjectManager*

O *ObjectManager* é responsável pelo tratamento dos pedidos do cliente, recebidos pelo *SessionManager* e pela abertura e fechamento do objeto do servidor RIO,

solicitado pelo cliente. Este componente também é responsável pelas requisições de blocos de dados feitas ao servidor.

### *CommMulticast*

Este componente é a interface de comunicação entre o cliente e o PI, no que diz respeito a movimentação do cliente. Este módulo tem como objetivo encaminhar as mensagens recebidas para o módulo responsável (*SessionManager*) e pelo envio de informações ao cliente, como por exemplo, a qual grupo *multicast* ele deve se unir.

### *RIOInterface*

Através deste componente o PI recebe os blocos de dados do servidor e também envia ao cliente. Em relação ao servidor, a interface é responsável pelo recebimento dos dados, via *unicast* e encaminhá-los ao *buffer* do PI. Na comunicação com o cliente, a interface é responsável pelo envio dos dados, *unicast* e *multicast*. Esta interface possuía apenas o envio através de *unicast* e foram necessárias mudanças e novas implementações para que ela pudesse ser capaz de enviar os dados através de *multicast*.

## 4.6 Estruturas de controle

No PI existem três estruturas de dados para o gerenciamento dos vídeos e dos clientes: a primeira estrutura (*OptimalWindowInfo*) armazena o tamanho da janela ótima do *patching* para vídeos que estão no servidor, a segunda gerencia os clientes ativos (*InfoClient*) e a terceira gerencia os clientes inativos (*InactiveClient*). A primeira, *OptimalWindowInfo*, estrutura é utilizada para determinar por quanto tempo a janela de um determinado grupo permanecerá ativa. Já a segunda estrutura, *InfoClient*, é utilizada para o armazenamento e busca de grupos para os clientes novos ou que estão se movimentando. A terceira estrutura, *InactiveClient*, armazena todos os clientes inativos no sistema, pois estes não pertencem a nenhum grupo de transmissão, logo não podem estar na estrutura *InfoClient*. Ao ser inicializado o

PI, o *SessionManager* armazena, em uma lista encadeada (*OptimalWindowInfo*), a informação sobre o tamanho da janela ótima de cada vídeo. Para o cálculo desta, foi utilizada a equação 3.2 descrita na seção 3.1, porém este parâmetro pode ser alterado. Esta lista pode ser visualizada na Figura 4.18.

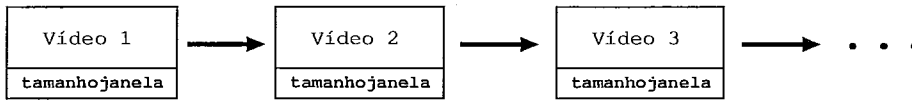


Figura 4.18: Lista dos tamanhos das janelas de cada vídeo

A estrutura de clientes inativos é uma lista encadeada com todos os clientes que estão inativos no momento, ou seja, os clientes estão em pausa (enviaram uma mensagem de pausa ao PI) e os clientes que se movimentaram (operações *forward*, *rewind*) para posições do vídeo que já possuem em seus *buffers* locais em disco, pois nos dois casos o cliente não pertence a nenhum grupo. Assim que o estado do cliente for modificado, ou seja, sair do estado de pausa ou deixar de ler de seu *buffer* local, ele informa ao PI que aloca um grupo para este cliente. Esta estrutura pode ser visualizada na Figura 4.19.

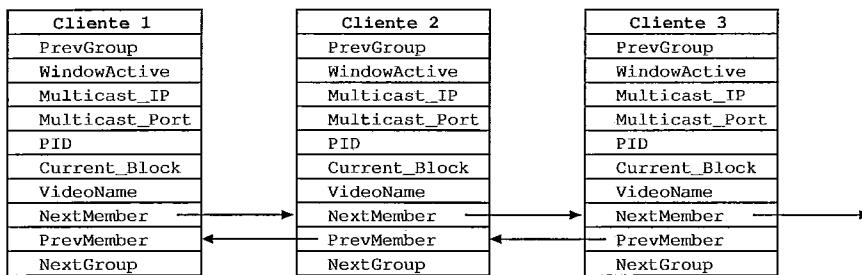


Figura 4.19: Lista dos clientes inativos

A estrutura de dados *InfoClient*, que pode ser visualizada na Figura 4.20, contém todas as informações relativas aos clientes e grupos ativos e inativos. Para cada cliente existe um conjunto de informações que são necessárias para a execução dos algoritmo proposto neste trabalho, como por exemplo a troca de um usuário de grupo devido a uma ação de avançar executada pelo cliente. Estas informações são descritas abaixo e fazem parte da estrutura que armazena os dados dos clientes:

- *WindowActive*: indica se a janela de *patching* daquele grupo está ativa ou não;
- *Multicast\_IP*: IP *multicast* que este cliente está “escutando”;
- *Multicast\_Port*: porta de recebimento dos dados;
- *PID*: identificador do cliente no PI. Este valor é exclusivo deste cliente;
- *Currente\_Block*: último bloco envia ao cliente;
- *VideoName*: nome do vídeo que este cliente está assistindo;
- *NextMember*: ponteiro para o próximo membro do grupo;
- *PrevMember*: ponteiro para o membro anterior do grupo;
- *NextGroup*: ponteiro para o próximo grupo existente;
- *PrevGroup*: ponteiro para o grupo anterior.

As quatro últimas informações da estrutura *InfoClient* são utilizadas para uma busca mais rápida em toda a estrutura, fato este que ocorre quando um cliente se movimenta e o PI deve buscar um novo grupo para ele, ou quando o líder de um grupo sai e deve ser alocado um novo líder.

A Figura 4.20 ilustra a estrutura de dados *InfoClient* para a seguinte configuração: três grupos de clientes, onde o primeiro grupo possui 3 clientes, o segundo 2 e o terceiro 4. Nesta estrutura o primeiro membro de cada grupo é o líder; caso este cliente se movimente e tenha que sair deste grupo, o líder será o próximo cliente da lista, caso exista. Não existindo outro membro no grupo este é extinto.

## 4.7 Mensagens de controle

O PI, da mesma forma que o cliente, possui o componente *CommMulticast* que provê o serviço de envio e recebimento das mensagens de controle. Serão descritos, a seguir, os procedimentos que devem ser executados quando do recebimento de cada uma das mensagens. No caso do PI, é armazenada uma lista encadeada com

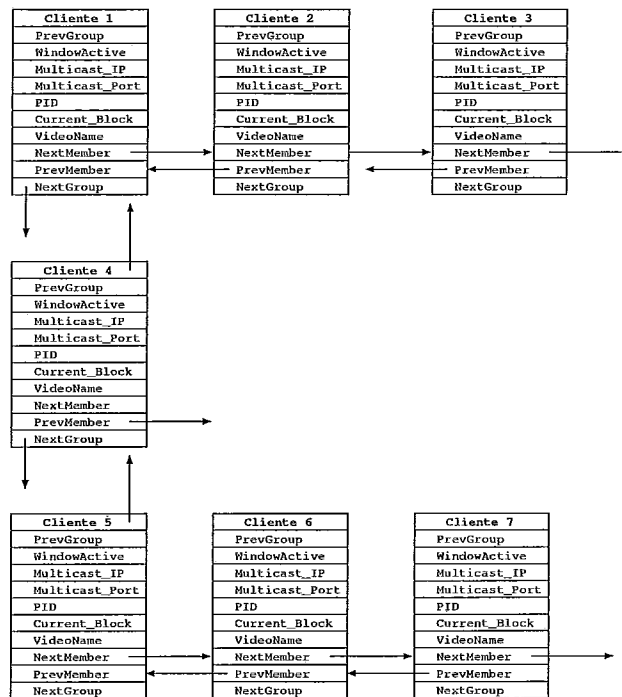


Figura 4.20: Estrutura das informações dos clientes

as informações de todos os clientes que estão conectados a ele. As informações guardadas são as mesmas do cliente: PID e *socket*. Esta estrutura pode ser vista na Figura 4.21 e é ordenada por *socket*.

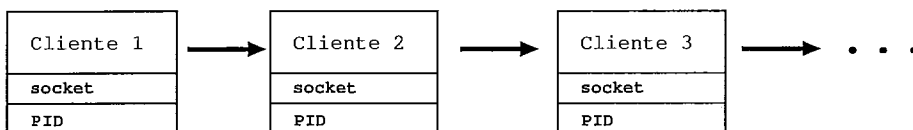


Figura 4.21: Estrutura de dados *CommMulticast*

### Requisição de abertura de objeto

Ao recebimento desta mensagem o PI irá solicitar ao servidor a abertura do objeto requisitado pelo cliente e irá alocar um grupo para este cliente. Caso exista algum grupo com a janela do *Patching* ativa, o novo cliente será incluído neste grupo, caso contrário um novo fluxo multicast será aberto para este cliente. Em ambos os casos é enviado ao cliente uma mensagem do tipo *msgcode\_IP*, informando ao cliente o grupo ao qual ele pertence.

### msgcode\_PID

Esta mensagem é gerada pelo cliente após o recebimento da resposta do pedido de abertura de objeto. Na chegada desta mensagem a *CommMulticast* cria um *socket* para a comunicação com este cliente e associa o PID do cliente com este *socket* recebido. Após a associação, a *CommMulticast* envia ao cliente uma mensagem do tipo msgcode\_IP, para que ele possa iniciar o recebimento dos dados via *multicast*.

### msgcode\_IP

Esta mensagem contém o IP, a porta e um *flag* indicando se o cliente é ou não líder do grupo. O objetivo desta mensagem é informar ao cliente a qual grupo *multicast* ele deve se unir para receber os dados deste grupo, e caso este cliente seja líder do grupo, ele deve fazer pedidos para os demais membros do grupo. As informações que são enviadas ao cliente são geradas a partir do procedimento executado quando do recebimento da mensagem msgcode\_MOVE. Esta mensagem é enviada ao cliente em resposta as mensagens msgcode\_PID e msgcode\_MOVE.

### msgcode\_MOVE

Quando do recebimento de uma mensagem do tipo msgcode\_MOVE o PI vai buscar, dentre os grupos ativos, um grupo do qual esse cliente possa receber os dados.

Para o melhor entendimento do funcionamento do módulo PI, será descrito um exemplo de troca de mensagens entre o cliente e o PI e os procedimentos executados quando este recebe alguma mensagem do cliente. Este procedimento pode ser visualizado na Figura 4.22 onde o *buffer* do cliente tem tamanho 2 blocos e o *buffer* do PI para cada cliente é de 5 blocos de dados. Serão descritos os passos executados pelo PI quando do recebimento de uma mensagem (instantes 1 a 13 da Figura 4.22, indicados na coluna “Tempo no PI”):

- Instante 1: o PI recebe a mensagem de requisição de abertura de sessão, neste

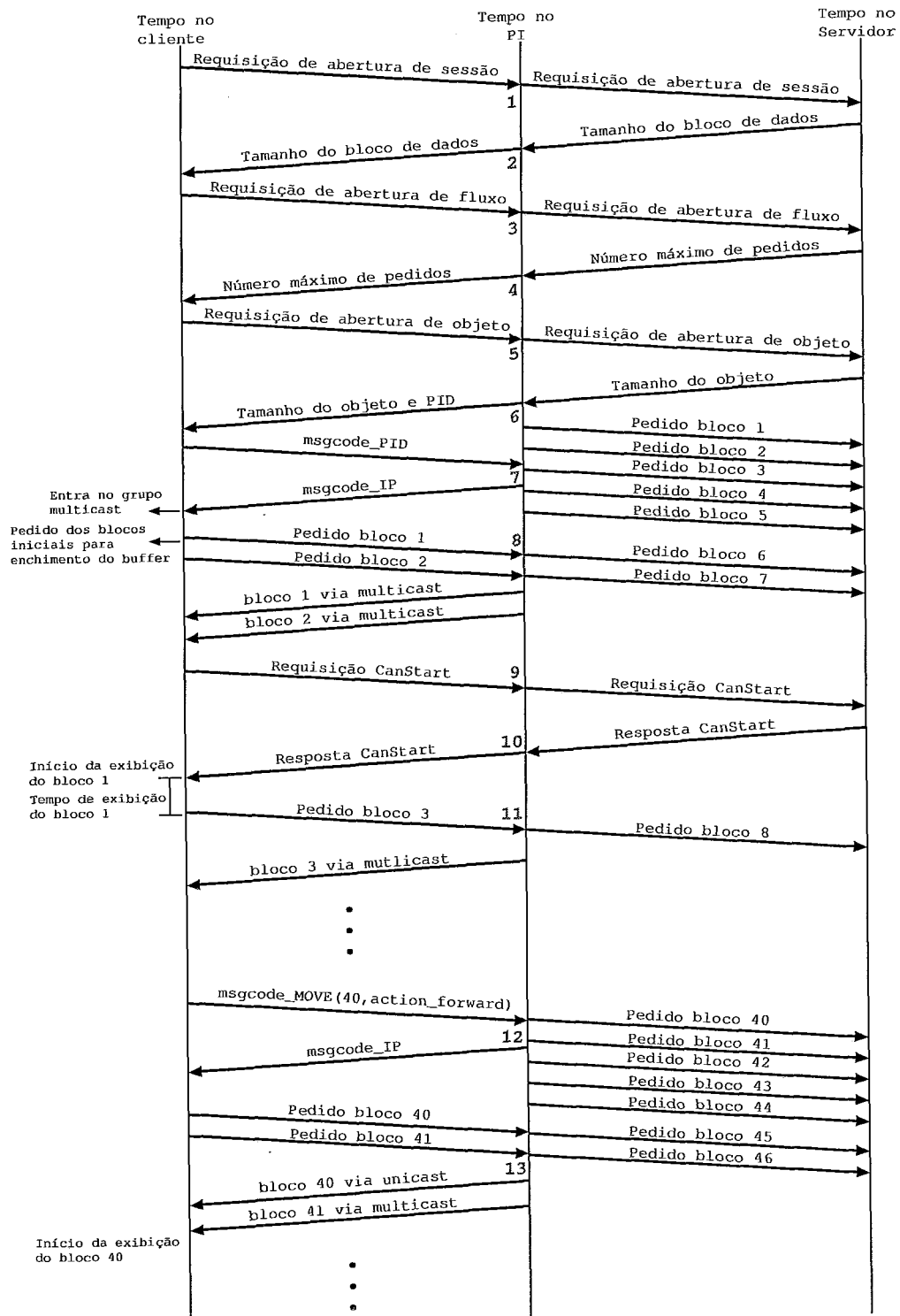


Figura 4.22: Troca de mensagens entre o cliente e o PI



momento ele solicita ao servidor a abertura de uma sessão;

- Instante 2: a resposta de abertura de sessão chega ao PI e este, de acordo com ela, envia uma mensagem ao cliente. Caso a resposta do servidor seja afirmativa, ou seja, a sessão foi aberta, o PI envia ao cliente esta mesma mensagem, caso contrário, envia uma resposta negativa ao cliente;
- Instante 3: neste momento é recebido um pedido de abertura de fluxo, o qual é repassado ao servidor para que o PI possa receber os dados a serem enviados ao cliente;
- Instante 4: quando o PI recebe a resposta do pedido de abertura de fluxo, ele executa o mesmo procedimento feito quando ele recebe a resposta do pedido de abertura de sessão;
- Instante 5: chega ao PI o pedido de abertura de objeto para que ele possa preencher a requisição de abertura do objeto no servidor RIO e executa o procedimento descrito na mensagem de requisição de abertura de objeto;
- Instante 6: neste instante chega do servidor a mensagem com a resposta da abertura de objeto. Caso a resposta seja positiva, nela está indicado o tamanho do objeto solicitado pelo cliente. Com este valor o PI preenche, juntamente com o PID (identificador do cliente no PI), a mensagem que é enviada ao cliente como resposta ao seu pedido de abertura de objeto. Em paralelo ao envio desta mensagem, o PI solicita ao servidor os blocos iniciais do vídeo para o preenchimento de seu *buffer*. Desta forma quando o cliente solicitar estes blocos, ele pode enviá-los imediatamente;
- Instante 7: o PI, baseado no PID do cliente, busca na estrutura *InfoClient* o IP, a porta, a informação de líder e envia ao cliente, através da mensagem *msgcode\_IP*, para que esse se una ao grupo *multicast* especificado;
- Instante 8: o PI recebe os pedidos para preenchimento do *buffer* do cliente. O PI imediatamente envia estes blocos, via *multicast*, para o cliente e solicita os blocos seguintes ao servidor. Neste exemplo, o cliente é o líder do grupo, portanto é ele quem solicita ao servidor os blocos;

- Instante 9: O PI recebe a mensagem *CanStart* indicando que o buffer do cliente está cheio, e solicita a autorização para iniciar a exibição. Esta autorização é feita para garantir que o *buffer* do PI estará cheio quando a exibição do vídeo iniciar. Neste momento o PI envia ao servidor a mesma mensagem para garantir que o *buffer* do servidor esteja cheio;
- Instante 10: quando do recebimento da mensagem de resposta ao *CanStart* o PI envia a mensagem autorizando o início da exibição;
- Instante 11: após a exibição do primeiro bloco de dados, o bloco 1, o cliente solicita ao PI o bloco 3 para o preenchimento de sua posição vazia no *buffer*. Quando esta mensagem chega ao PI ele imediatamente envia o bloco 3 ao servidor e solicita ao servidor o bloco 8;
- Instante 12: neste instante o cliente executou uma operação de avançar. O PI, ao receber esta mensagem, executa o procedimento descrito na mensagem `msgcode_MOVE` e retorna para o cliente o grupo *multicast* ao qual ele deve se unir. Paralelamente ao envio desta mensagem o PI solicita ao servidor o preenchimento de seu *buffer* para, assim que o cliente solicitar, enviar os blocos de dados;
- Instante 13: após o recebimento da mensagem `msgcode_IP` o cliente inicia os pedidos dos blocos, pois no exemplo ele é o líder do grupo. O PI envia os dados imediatamente ao cliente, pois estes blocos já foram solicitados ao servidor anteriormente.

# Capítulo 5

## Modelo do *Patching* Interativo

NESTE capítulo serão apresentadas as principais características do ambiente de modelagem Tangram-II, que foi utilizado para a simulação do *Patching* Interativo. Este ambiente permite a construção de modelos matemáticos de um sistema e possibilita a sua solução através de métodos analíticos ou de simulação. Será descrito, também, o modelo do *Patching* Interativo desenvolvido e os resultados obtidos com as simulações.

Além dos resultados obtidos com o modelo do mecanismo proposto, em [22] podem ser encontrados resultados de testes realizados com o servidor RIO operando em conjunto como o módulo de *Patching* Interativo, discutido no capítulo 4.4.

### 5.1 Introdução

O Tangram-II é um ambiente de modelagem desenvolvido para ensino e pesquisa que permite a construção e resolução de modelos dentro de um sistema integrado. Como pode ser visto na Figura 5.1, o Tangram-II é dividido em quatro módulos: *Modeling Environment*, *White Board*, *Traffic Generator* e *Viva Voz*.

O *Modeling Environment*, mostrado na Figura 5.2, é o ambiente onde são resolvidos os modelos desenvolvidos. Para especificar estes modelos, é usada a ferramenta gráfica TGIF. Diversos métodos de solução analíticos ou de simulação estão implementados na ferramenta e permitem o cálculo de várias medidas de interesse.

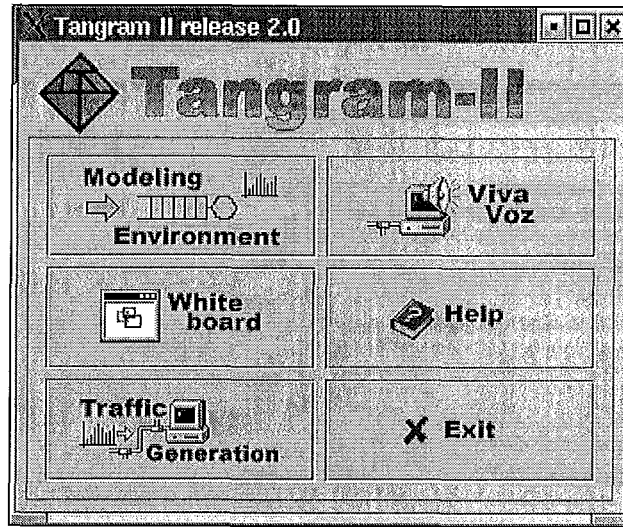


Figura 5.1: Interface gráfica inicial do Tangram-II

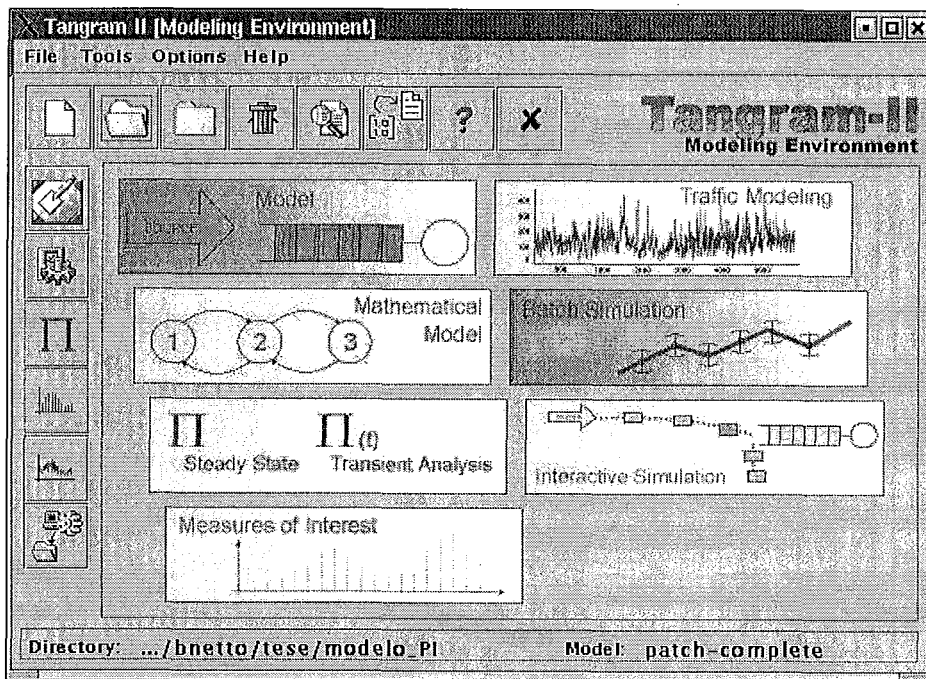


Figura 5.2: Ambiente de modelagem

O *Viva Voz* e o *White Board* são ferramentas usadas para trabalho cooperativo entre usuários. O *Viva Voz* é uma ferramenta de transmissão de voz em uma rede IP e possui mecanismos para garantia da qualidade de serviço, como recuperação de pacotes perdidos. O *White Board* foi implementado sobre a ferramenta gráfica TGIF de forma a permitir o trabalho cooperativo no desenvolvimento de um modelo, desenho ou *slides* para uma apresentação. O *White Board* é distribuído e inclui uma biblioteca de *multicast* confiável. Outro módulo do TANGRAM-II é o *Traffic Generator*, que permite a geração de tráfego IP ou ATM e coleta de diversas estatísticas da rede como, por exemplo, fração de pacotes perdidos, retardo fim-a-fim, distribuição do tamanho da rajada de perda.

## 5.2 Modelo *Patching* Interativo

O modelo desenvolvido no Tangram-II, que pode ser visualizado na Figura 5.3, possui dois componentes básicos: o objeto servidor e os objetos cliente. O objeto servidor é responsável pelo controle dos grupos de clientes e os objetos cliente são responsáveis pela movimentação de cada cliente: parar, pausar, tocar, avançar e retroceder. Neste modelo não são tratadas função de *FastForward* e *FastRewind*, onde o vídeo continua sendo exibido enquanto o usuário executa sua ação.

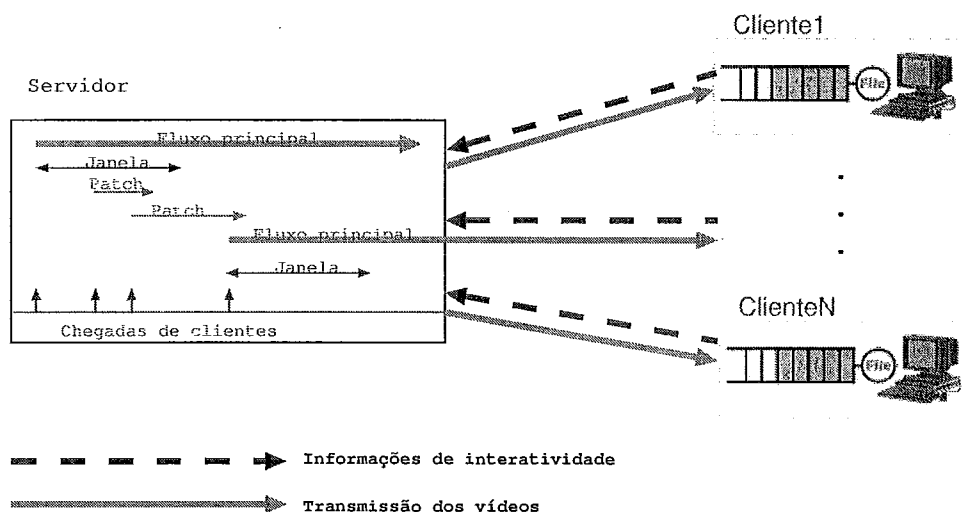


Figura 5.3: Modelo do sistema

No modelo um determinado vídeo é solicitado por múltiplos clientes e o comportamento de cada cliente é baseado em *traces* gerados a partir de um outro modelo construído, baseado nos slides do livro [27] e na experiência de professores. Até o presente momento ainda não temos disponíveis dados reais do comportamento de clientes em uma aplicação interativa. Portanto foi criado uma modelo para simular este comportamento. Entretanto, dados reais estão sendo disponibilizados pelo grupo do Professor Jim Kurose da Universidade de Massachusetts e serão utilizados no simulador desenvolvido neste trabalho. A chegada de clientes no sistema é determinada por um processo de *Poisson* com taxa igual a 1 ou 2 clientes por minuto.

## Objeto Servidor

Este objeto é responsável pelo gerenciamento dos clientes nos grupos e por receber e tratar as mensagens que são enviadas pelos clientes para informar sua interatividade. Para este controle o servidor possui 5 variáveis de estado:

- *Streaming\_Movies* representa o número de transmissões completas do vídeo;
- *Window\_Active* indica se a janela ótima do *patching* de algum grupo está ativa; caso positivo ela contém o identificador do grupo;
- *Time\_Streaming\_Movies* é um vetor onde cada posição armazena o instante do vídeo que um determinado grupo está visualizando;
- *Total\_Streaming\_Movies* é o vetor onde cada posição armazena o número de clientes que estão alocados em um determinado grupo.

A cada mensagem recebida de um cliente pelo servidor, este verifica o tipo de movimentação e tenta alocar aquele cliente em um outro grupo. Caso não exista, cria-se um novo grupo para este cliente. Este procedimento está descrito detalhadamente na seção 4.7. O servidor verifica se é um novo cliente, caso positivo verifica se existe uma janela ativa do *Patching* para unir este cliente no grupo de janela ativa. Caso o

cliente não seja novo, então é uma movimentação, o servidor procura um grupo que esteja próximo a posição para a qual o cliente se moveu, de acordo com o *delta\_after* e *delta\_before* descritos na seção 4.7. Se o cliente que está se movendo pertence ao grupo com janela ativa e não há mais nenhum outro membro no mesmo grupo, então a janela é desativada. Se houver outros membros no grupo, nada é alterado. Após a escolha do grupo, o servidor informa ao cliente, através de uma mensagem, o novo grupo ao qual pertence e o tamanho do *patch* a ser solicitado.

## Objeto Cliente

Dentro do modelo existem vários objetos cliente, onde cada um deles é responsável por simular o comportamento de um usuário, a partir de alguns *traces* de informações.

O diagrama de estados, apresentado na Figura 5.4, representa o comportamento de um aluno assistindo ao vídeo de uma aula do capítulo 3 do livro *Computer Networking* de J. Kurose e K. Ross [27]. Neste diagrama cada estado representa uma seção do capítulo 3 do livro conforme apresentado na Tabela 5.1.

Estado	Seção	Nome da seção
3.1	3.6 e 3.6.1	<i>Principles of Congestion Control e Scenario 1</i>
3.2	3.6.1	<i>Scenario 2</i>
3.3	3.6.1	<i>Scenario 3</i>
3.4	3.6.2 e 3.6.3	<i>ATM ABR Congestion Control</i>
3.5	3.7 e 3.7.1	<i>TCP Congestion Control e Congestion Window</i>
3.6	3.7.1	<i>TCP Slowstart</i>
3.7	3.7.1	<i>TCP Congestion Avoidance</i>
3.8	3.7.1	<i>Does TCP ensure Fairness?</i>

Tabela 5.1: Seções utilizadas no diagrama

As probabilidades de transição entre os estados estão representadas sob os arcos e acima do estado encontra-se o tempo em minutos de exibição do vídeo referente a aula daquela seção do livro. Os estados marcados com a letra “P” representam uma pausa realizada pelo cliente para examinar com mais detalhe uma parte do vídeo e

*slides* correspondentes. Neste modelo os clientes podem começar a assistir o vídeo desde o início, ou seja, a partir do estado 3.1 com uma certa probabilidade. Eles podem iniciar também em pontos intermediários, nos estados 3.5 e 3.6. Os estados 3.5 e 3.6 representam a parte da aula onde é descrito o protocolo TCP. Este tópico, em geral, gera bastante dúvidas e interesse por parte dos alunos. Por este motivo foi representado no modelo a possibilidade do aluno iniciar a visualização do vídeo a partir deste tópico.

As probabilidades de transição entre os estados foram definidas baseadas na experiência de ensino dos Professores Edmundo de Souza e Silva e Rosa Leão do curso de Teleprocessamento e Redes da graduação da UFRJ. Após os estados 3.7 e 3.8, o cliente pode terminar a exibição do vídeo com uma certa probabilidade indicada nas setas de saída de cada um deles. A chegada de clientes é determinada por um processo de *Poisson* com taxas iguais a 1 e 2 clientes por minuto, a depender do caso apresentado.

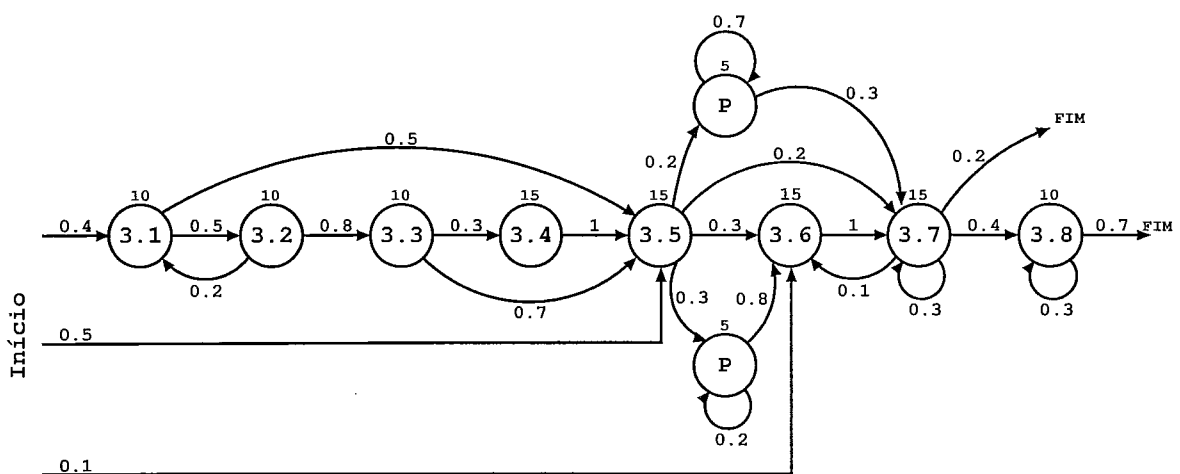


Figura 5.4: Modelo do comportamento do cliente

Cada cliente teve seu comportamento definido através de *traces* gerados a partir do modelo da Figura 5.4. As informações dos *traces* estão distribuídas em três arquivos conforme descrito a seguir. O primeiro arquivo (arquivo 1) contém informações sobre os comandos *play*, *forward* e *rewind*. A primeira linha contém o tempo de início da exibição do vídeo (*play*) e as demais linhas contém os instantes dos saltos que o cliente executará. Na Figura 5.5, temos um exemplo do arquivo 1, onde o



cliente inicia a exibição no tempo 0.0, dá um salto 530.0 segundos após o início e depois outro 420.0 segundos após o anterior. Este arquivo define o tempo entre os comandos de *play*, *rewind*, *forward* executados pelo cliente.

0.0	1
530.0	1
420.0	1

Figura 5.5: Exemplo do arquivo com instantes de movimento do cliente

O segundo arquivo (arquivo 2) contém a posição do vídeo, em unidades de tempo, para a qual o cliente se moveu no instante dos eventos do arquivo 1. Este arquivo é lido antes que a simulação se inicie e é armazenado em um vetor. Um exemplo do arquivo 2 pode ser visualizado na Figura 5.6, onde a posição que o cliente inicia o vídeo é 0.0 segundos. Após 530.0 segundos, informação coletada do arquivo 1, o cliente se movimenta para a posição 330.0. Decorridos mais 420.0 segundos, o cliente se movimenta para a posição 150.0 do vídeo. Este arquivo contém uma informação a mais em sua última linha, indicando quanto tempo após o início do vídeo o cliente irá se desconectar, ou seja, informando o término daquela sessão, no caso deste exemplo, 800.0 segundos.

0.0	1
330.0	1
150.0	1
800.0	1

Figura 5.6: Exemplo do arquivo com a posição do vídeo para a qual o cliente se moveu

O terceiro arquivo (arquivo 3) contém os instantes onde o cliente irá paralisar/retornar a exibição do vídeo. Na Figura 5.7 temos um exemplo onde o cliente paralisa a exibição do vídeo no instante 100.0 segundos e retorna a exibição 50.0 segundos depois. Em todos os três arquivos existe uma segunda coluna que contém sempre o valor 1. Isto se deve a uma característica do simulador (Tangram-II) que indica o número de eventos que irão ocorrer naquele intervalo de tempo. No caso

deste modelo é sempre 1.

100.0	1
50.0	1

Figura 5.7: Exemplo do arquivo de pausa

O objeto cliente tem como principais variáveis de estado:

- *Buffer* é um vetor que indica as partes do vídeo já armazenados no *buffer* em disco do cliente, Caso o cliente se movimente, este vetor é examinado primeiro evitando a solicitação do bloco ao servidor;
- *Group* indica qual o grupo atual do cliente;
- *Active\_Patch* indica se o cliente está ou não recebendo *patch*;
- *Movie\_End* indica o fim do vídeo;
- *Pause* indica se o cliente está ou não em pausa;
- *Move\_Position* é o vetor que indica para quais posições o cliente se moverá no decorrer do vídeo. Este vetor é preenchido com os dados do arquivo 2 descrito acima.

O primeiro evento que ocorre na simulação do cliente é o *play*, onde o cliente informa ao servidor que está solicitando o vídeo para uma determinada posição, contida na primeira posição do vetor *Move\_Position*. O servidor procura, de acordo com o procedimento descrito no objeto servidor, um grupo para o cliente e retorna o identificador deste grupo e o tamanho do *patch* a ser solicitado. Com o recebimento desta mensagem, o cliente atualiza o seu grupo e começa a acumular os dados, tanto do *patch*, caso necessário, quanto do fluxo principal ao qual ele se uniu. Após o término do evento *play*, o evento do término de vídeo é habilitado para disparar de acordo com a distribuição determinística com taxa  $1/Tempofimvideo$ , onde o parâmetro *Tempofimvideo* é o valor da última posição do vetor *Move\_Position*.

Entre o evento de *play* e o término do vídeo podem ocorrer os eventos de pausa, avanço e retrocesso de acordo com os arquivos de 1 e 3. Os arquivos 1 e 3 definem o intervalo entre eventos. Cada vez que um evento desses ocorre são executados os procedimentos de pausa e salto. No evento de pausa o cliente envia uma mensagem ao servidor informando que ele está em estado de pausa e habilita o evento de *resume* para que em um determinado tempo este cliente saia do estado de pausa, solicitando ao servidor um novo grupo. No evento de salto, o cliente primeiramente verifica se ele já possui aquele dado armazenado em disco, caso positivo ele informa ao servidor que está lendo do buffer. Caso negativo, o cliente solicita ao servidor, informando a posição do vídeo para a qual ele se moveu, um novo grupo para o recebimento dos dados.

## 5.3 Resultados

Através do modelo desenvolvido algumas medidas podem ser obtidas como: número de clientes ativos, número de transmissões completas do vídeo, número de *patches* transmitidos a todos os cliente. Estas medidas podem auxiliar, por exemplo, na configuração do protótipo para o seu melhor desempenho. Para todas as simulações foi estabelecido o valor do *delta before* igual a 60 segundos, pois este valor deve ser suficientemente pequeno para que o usuário não perceba que está recebendo blocos anteriores ao que foi solicitado, conforme descrito na seção 4.7.

Para a análise do algoritmo proposto, 6 cenários foram avaliados. Nestes cenários foram avaliados o *Patching* Interativo e o *Patching* original. No *Patching* original, cada vez que um cliente realiza um movimento de pausa, avanço ou retrocesso, é estabelecido um novo fluxo unicast para ele. Duas medidas principais foram avaliadas: o total de fluxos sendo transmitidos para os clientes (fluxo principal + *patches*) e a fração do tempo que o servidor permaneceu com um determinado número de fluxos ativos. A fração do tempo foi calculada da seguinte forma: para todos os instantes de simulação, quando a soma do número transmissões completas do vídeo mais os *patches* para todos os clientes for maior que um dado nível, acumula-se uma unidade. Este nível pode ser, por exemplo, zero onde mostra quanto tempo o servidor

ficou ocupado com pelo menos uma transmissão.

### Cenário 1:

Neste cenário todos os clientes iniciam a visualização do vídeo pelo estado 3.1 (do diagrama da Figura 5.4), e o acesso é seqüencial, ou seja, o cliente não pausa o vídeo e não executa saltos. São utilizados 200 clientes e uma taxa de chegada de 2 clientes por minuto. Neste caso, as curvas do *Patching* original e do *Patching* Interativo devem ser idênticas, considerando que para acesso seqüencial o PI se comporta da mesmo forma que o *Patching*.

Na Figura 5.8, podemos observar as curvas que representam o número de fluxos, principais e *patches*, ativos no servidor ao longo do tempo. Nessa figura as curvas do PI e do *Patching* ficaram sobrepostas, confirmando o comportamento esperado. A terceira curva indica o número de clientes ativos, ou seja, quantos fluxos estariam abertos se fosse transmitido um fluxo para cada cliente. Esta curva estará sempre acima das demais, pois não podemos ter mais fluxos que usuários ativos. A Figura 5.9 representa a fração do tempo em que o servidor permanece acima de determinados níveis, pois com isso podemos avaliar o percentual do tempo que o servidor está transmitindo uma determinada quantidade de fluxos. Como exemplo temos que o servidor transmitiu mais de 12 fluxos durante 40 por cento do tempo. Nessa figura podemos observar o mesmo comportamento, onde as curvas do PI e do *Patching* estão sobrepostas.

### Cenário 2:

Neste cenário o comportamento do cliente é representado pelo diagrama da Figura 5.4, ou seja, o cliente executa operações de pausa, retrocesso e avanço com uma certa probabilidade. Foram considerados 100 clientes e uma taxa de chegada de 1 cliente por minuto. Serão usados os mesmo valores de *delta\_after* definidos para cenário 1, com exceção do valor de 570 (janela ótima do *Patching* no cenário 1) que foi substituído pelo valor da janela ótima do *Patching* neste cenário que é 790.

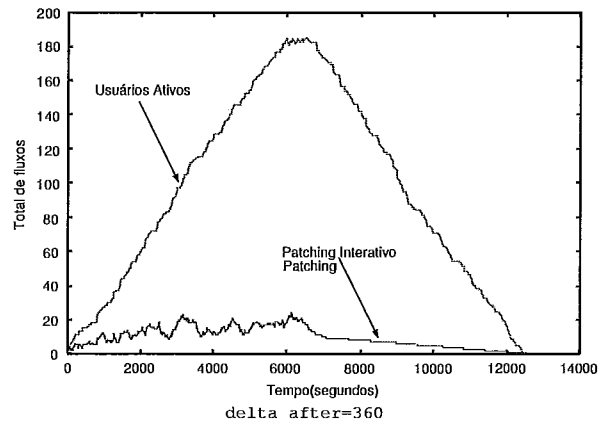


Figura 5.8: Total de fluxos no cenário 1

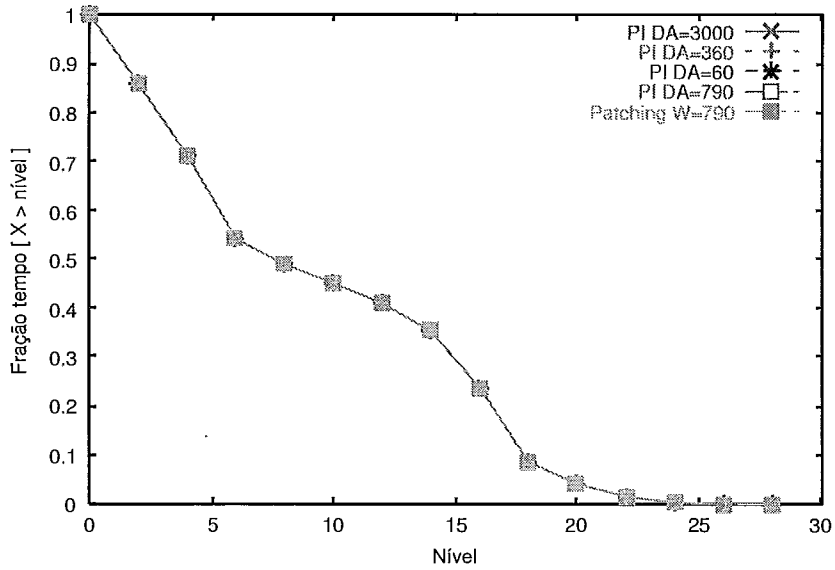


Figura 5.9: Fração de tempo no cenário 1

Como pode ser observado na Figura 5.10, o número de fluxos ativos do PI é bem menor que o do *Patching* original durante todo o tempo e para todos os valores de  $\delta_{after}$ . Este comportamento é esperado pois no *Patching* original clientes que se movem não compartilham mais o mesmo fluxo, recebendo os dados via *unicast*. A Figura 5.11 representa a fração do tempo em que o servidor permanece acima de determinados níveis e nota-se que o melhor comportamento se dá com o uso do PI com  $\delta_{after}$  igual a 360 (caso b da Figura 5.10), mostrando que este é o melhor valor do parâmetro  $\delta_{after}$ , dentre os valores considerados neste cenário. Pode-se observar nos casos a e d, da Figura 5.10, que as curvas do PI ficaram bem acima das curvas em b e c, mostrando que o valor de  $\delta_{after}$  não deve ser muito maior e nem muito menor que a janela ótima do *Patching*. No caso a, o aumento na curva do PI se deve a diminuição no número de uniões entre fluxos de clientes, pois a janela para união ( $\delta_{after}$ ) é muito pequena. Já no caso d, onde o  $\delta_{after}$  é igual a 3000, o aumento na curva ocorre em consequência da união de clientes muito distantes, o que ocasiona solicitação de *patches* longos.

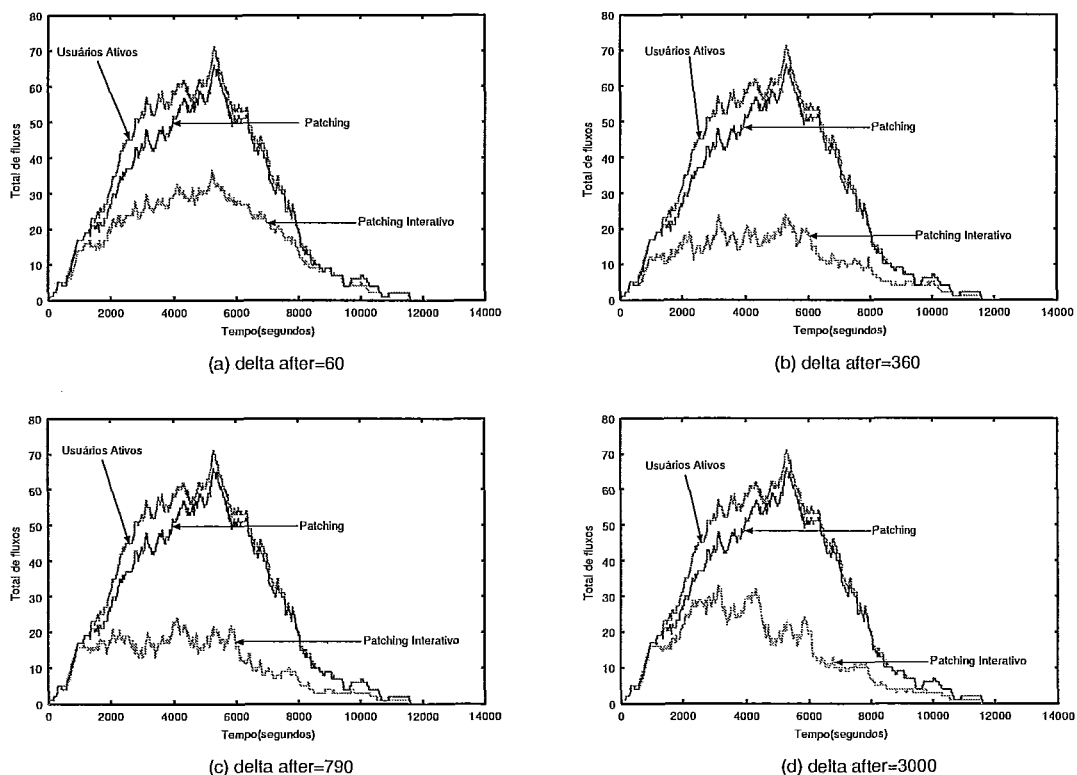


Figura 5.10: Total de fluxos no cenário 2

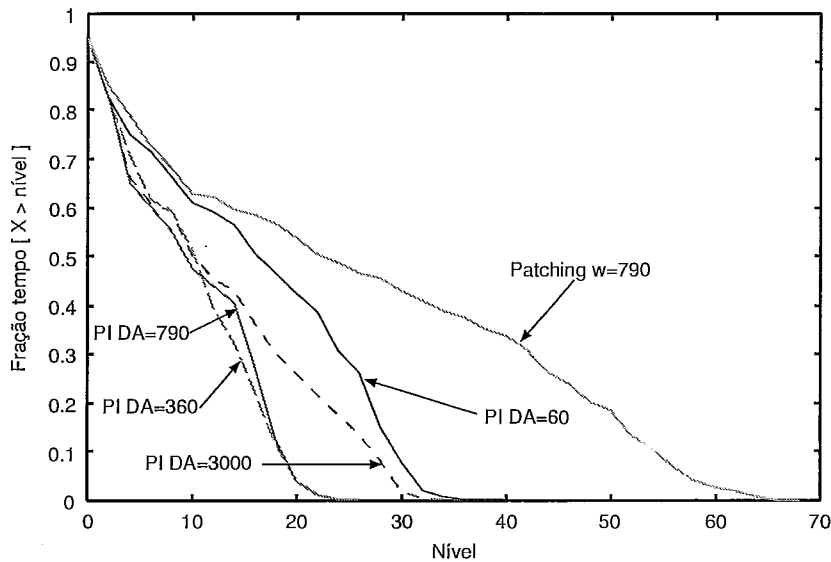


Figura 5.11: Gráfico fração de tempo no cenário 2

### Cenário 3:

Neste cenário incrementou-se o número de clientes para que o total de usuários ativos no sistema aumente. Foi utilizado o diagrama da Figura 5.4 para representar o comportamento dos clientes. O número total de clientes considerado é 200 e a taxa de chegada é de 2 clientes por minuto. Como pode ser observado na Figura 5.12, os deltas 200, 300 e 360 obtiveram os melhores resultados, pois se mantiveram com um total de fluxos baixo durante toda a simulação. Na Figura 5.13 pode-se observar a fração do tempo de permanência acima de alguns níveis e nela nota-se uma pequena vantagem do *delta\_after* 360 em relação ao 300 e 200. Da mesma forma que no cenário 2, as curvas do PI nos casos a e e, onde o *delta\_after* é igual a 60 e 3000 respectivamente, ficam acima dos demais casos.

### Cenário 4:

Neste cenário o modelo do comportamento do cliente foi alterado para que todos os clientes iniciassem o vídeo no estado 3.1, ou seja, no começo do vídeo e interagem com o mesmo de acordo com o modelo apresentado na Figura 5.4.

Na Figura 5.14, se observa uma redução no número de fluxos abertos pelo *Pat-*

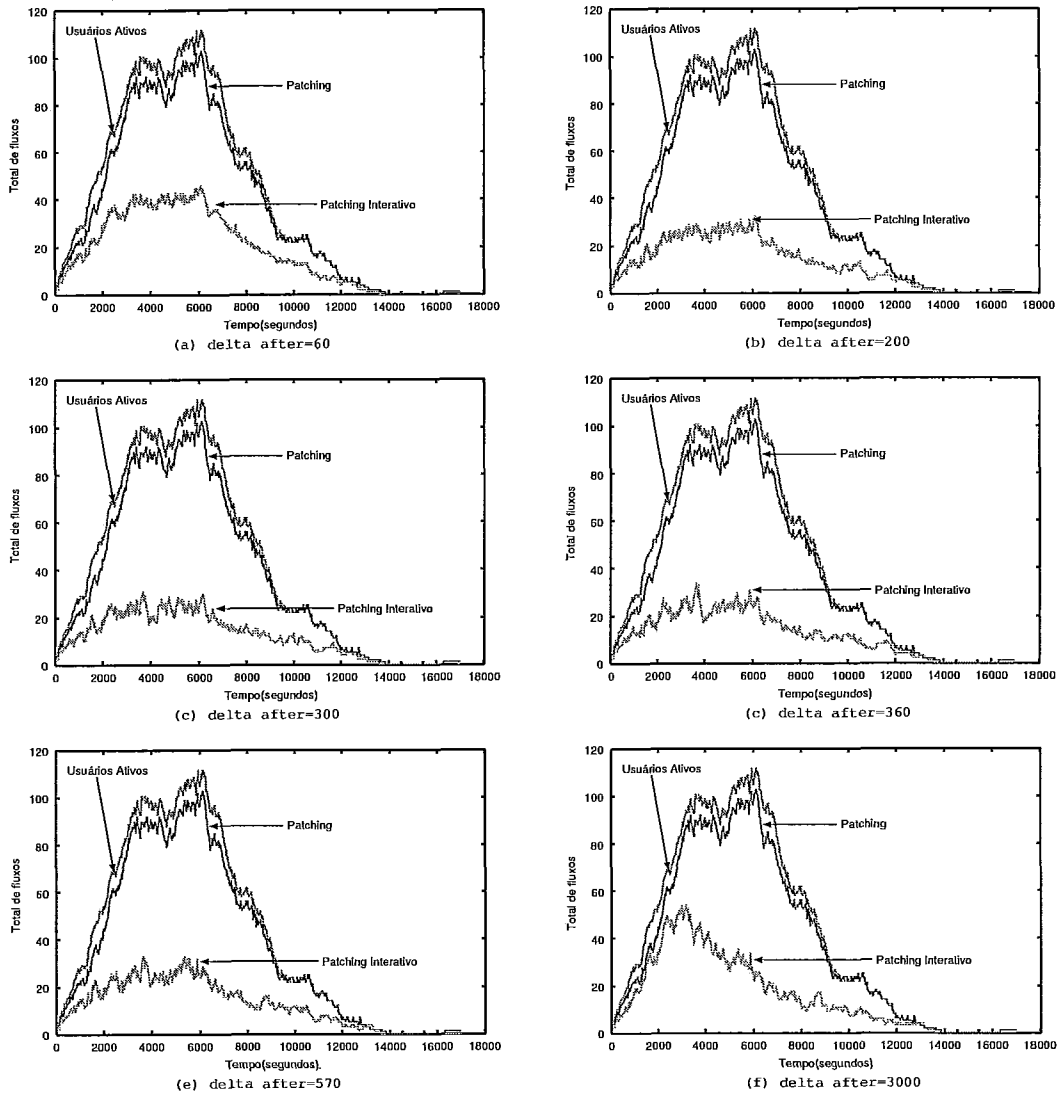


Figura 5.12: Total de fluxos no cenário 3



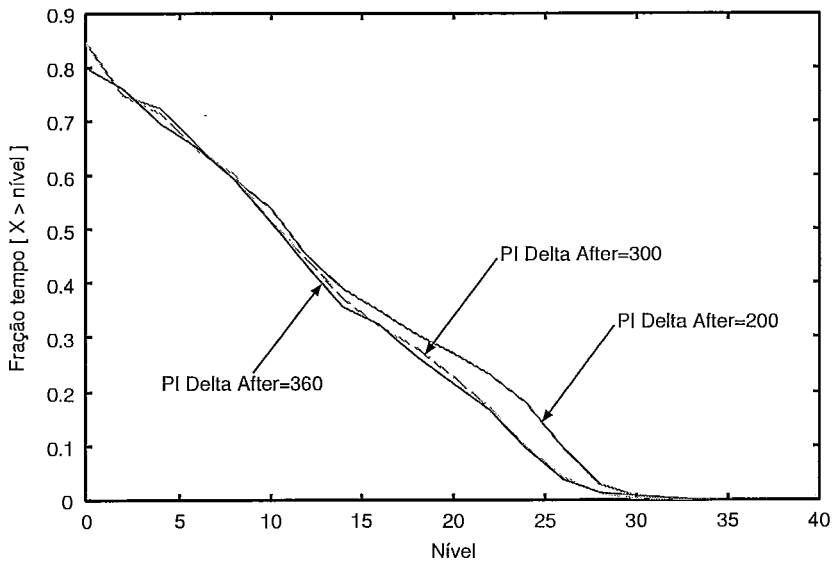


Figura 5.13: Gráfico fração de tempo no cenário 3

*ching*, confirmando o comportamento esperado, e um aumento no número de usuários ativos em relação ao cenário 3. Este último fato se deve ao maior tempo de permanência dos clientes, tendo em vista que estes sempre iniciam do começo do vídeo. No caso do *Patching* Interativo, se observa que para os valores de *delta after* iguais a 360 e 570 obteve-se os melhores resultados, fato este confirmado na Figura 5.15 em que esses mesmos valores se mantêm com as menores frações de tempo em comparação com as demais curvas.

#### Cenário 5:

No cenário 5, foram obtidos novos *logs* de comportamento do cliente através de algumas alterações nas probabilidades de transição do diagrama, apresentado na Figura 5.4. Essas mudanças foram feitas com o objetivo de melhorar o desempenho do *Patching*, aumentando as probabilidades do cliente assistir seqüencialmente o vídeo e também de iniciar a exibição pelo começo da mídia, ou seja, pelo estado 3.1. O novo diagrama de estados pode ser visto na Figura 5.16.

Como mostra a Figura 5.17, o *Patching* teve uma redução no número de fluxos ativos, como esperado, mas continua acima do PI, pois mesmo com essas mudanças ainda existem clientes que se movimentam e nem todos iniciam a exibição pelo

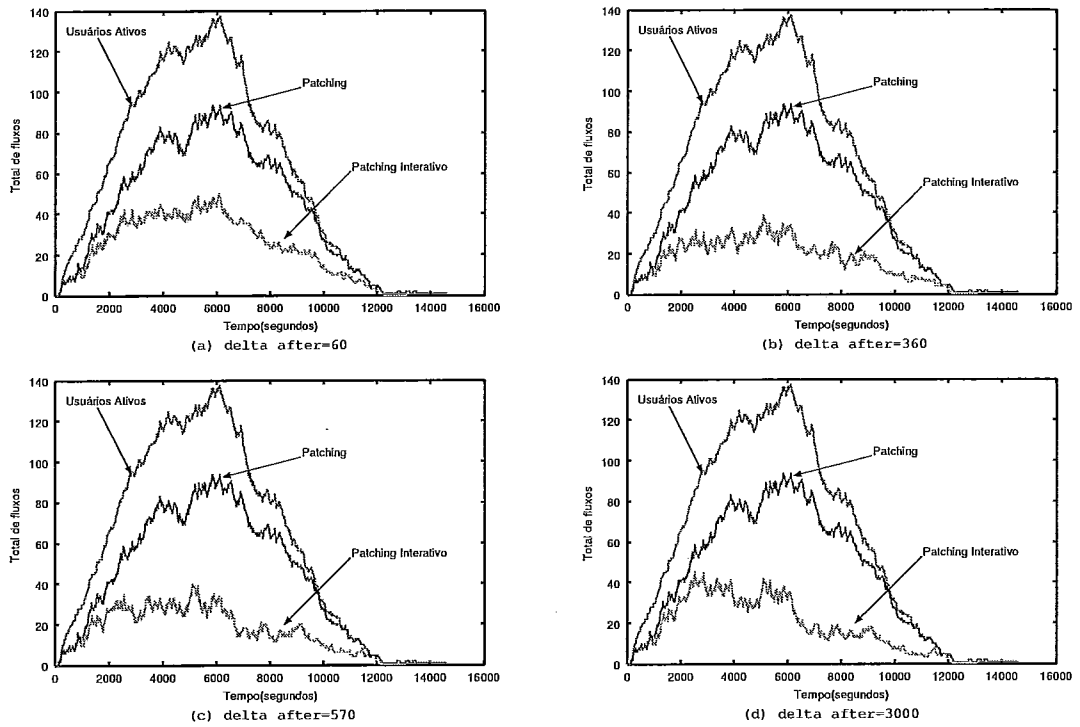


Figura 5.14: Total de fluxos no cenário 4

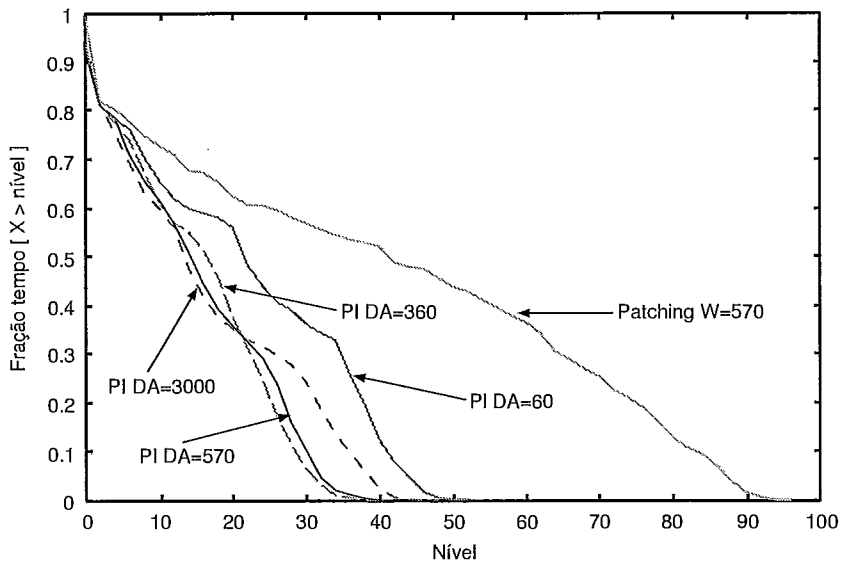


Figura 5.15: Gráfico fração de tempo no cenário 4

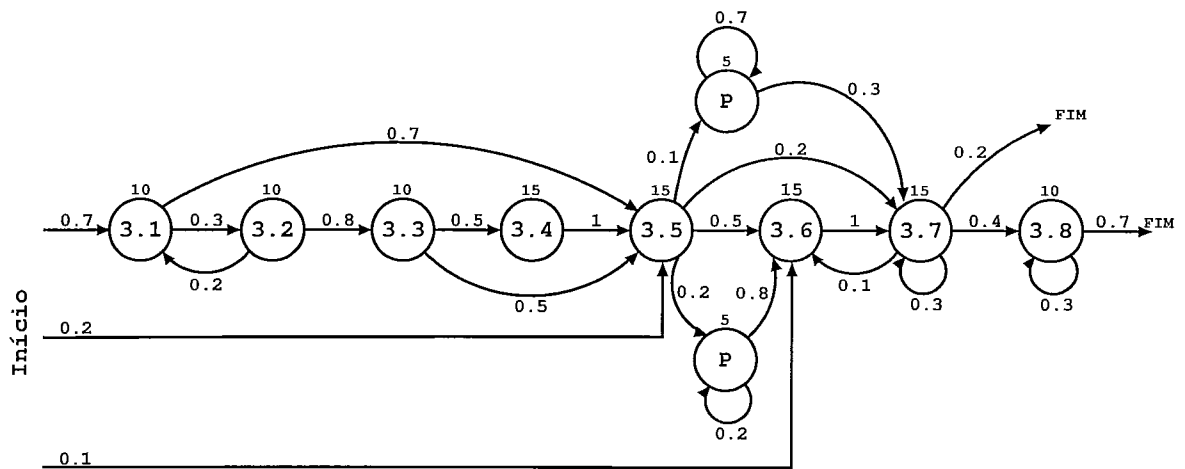


Figura 5.16: Diagrama de estados com novas probabilidades

começo do vídeo. Com o passar do tempo, a movimentação dos clientes faz com que a curva do *Patching* se torne igual a dos usuários ativos. Da mesma forma que no cenário 4, o PI com *delta\_after* igual a 360 e igual a 570 teve os melhores resultados, também mostrados na Figura 5.18.

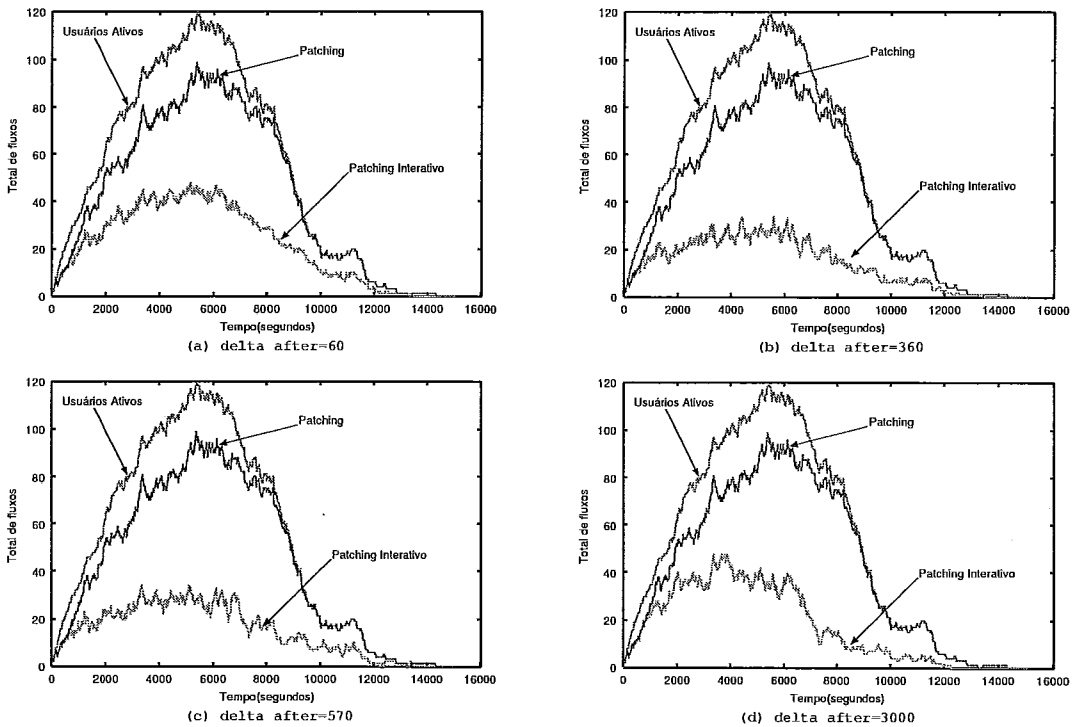


Figura 5.17: Total de fluxos no cenário 5

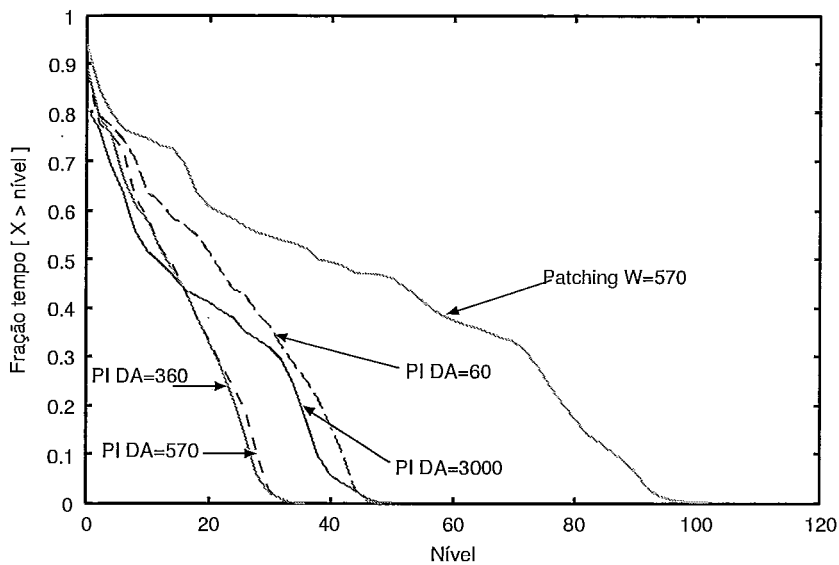


Figura 5.18: Gráfico fração de tempo no cenário 5

### Cenário 6:

Neste cenário foram utilizados logs reais obtidos através do sistema MANIC do Professor Jim Kurose da Universidade de Massachusetts. Através dos logs foi desenvolvido um modelo de comportamento dos usuários para a geração de carga no servidor, pois para o vídeo escolhido (dentro os vídeos do curso de Redes de Computadores) obtemos apenas 59 clientes. Este vídeo foi escolhido pois continha o maior número de clientes. Este vídeo contém 20 slides onde representamos cada um deles como um estado do modelo. O modelo construído a partir dos logs pode ser visualizado na Figura 5.19 e foram utilizados 200 cliente e uma taxa de chegada de 2 clientes por minuto seguindo um processo de *Poisson*. Nesta figura o bloco superior esquerdo representa as probabilidades do cliente que está em pausa permanecer em pausa, o bloco superior direito indica as probabilidades do cliente que está em pausa retornar a exibição, o bloco inferior esquerdo representa as probabilidades do cliente que está exibindo o vídeo entrar em estado de pausa e o bloco inferior direito representa as probabilidades dos clientes que estão exibindo uma determinada transparência e passam a exibir outra, sendo a próxima transparência (acesso seqüencial) ou não (saltos).

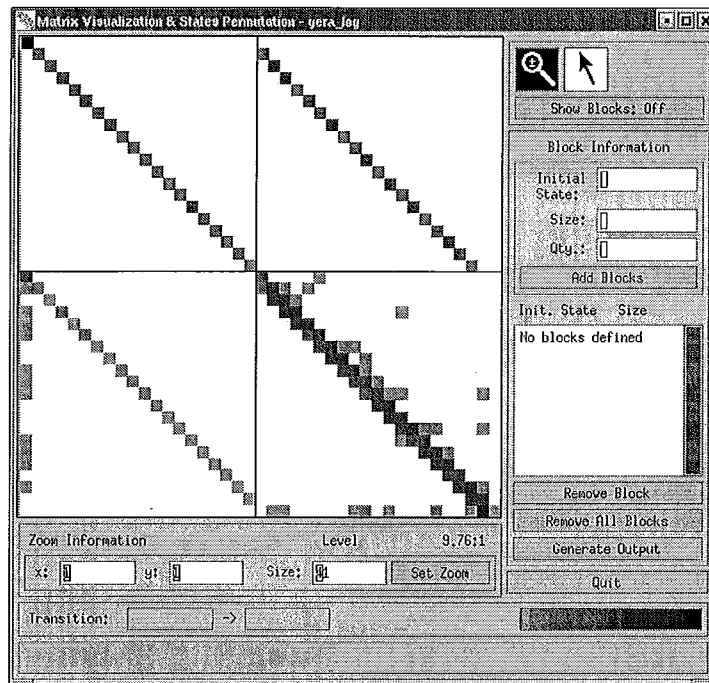


Figura 5.19: Modelo do comportamento do cliente no cenário 6

Com base nas Figuras 5.20 e 5.21, podemos observar que o melhor valor para o *Delta After* para este cenário foi 467, onde nota-se que a curva do Patching Interativo com *Delta After* igual 467 fica abaixo das demais. Em todos os casos o *Patching Interativo* ficou bem abaixo do *Patching Original*, mostrando-se mais eficiente.

Outro resultado obtido para comparação entre as técnicas foi o número máximo de fluxos ativos e o número máximo de usuários ativos para todos os cenários apresentados. Como pode ser observado na tabela 5.2, para o cenário 1, este número é igual a 24 para todos os casos considerados, sendo bastante inferior ao número de usuários ativos. Estes valores confirmam os dados mostrados no cenário 1, onde todas as curvas ficaram sobrepostas, com exceção da curva dos usuário ativos. Para o cenário 2, podemos observar que os menores picos são do PI com *delta\_after* igual a 360 e 790, confirmando os resultados apresentados no cenário 2. Já para o cenário 3, observa-se que o menor pico é com *delta\_after* igual 300. Com base nesta tabela e na Figura 5.13, concluímos que para este cenário, o melhor valor para *delta\_after* é 300. No cenário 4 os picos das curvas vieram confirmar os dados apresentados anteriormente, onde os deltas iguais a 360 e 570 tiveram o melhor desempenho. O

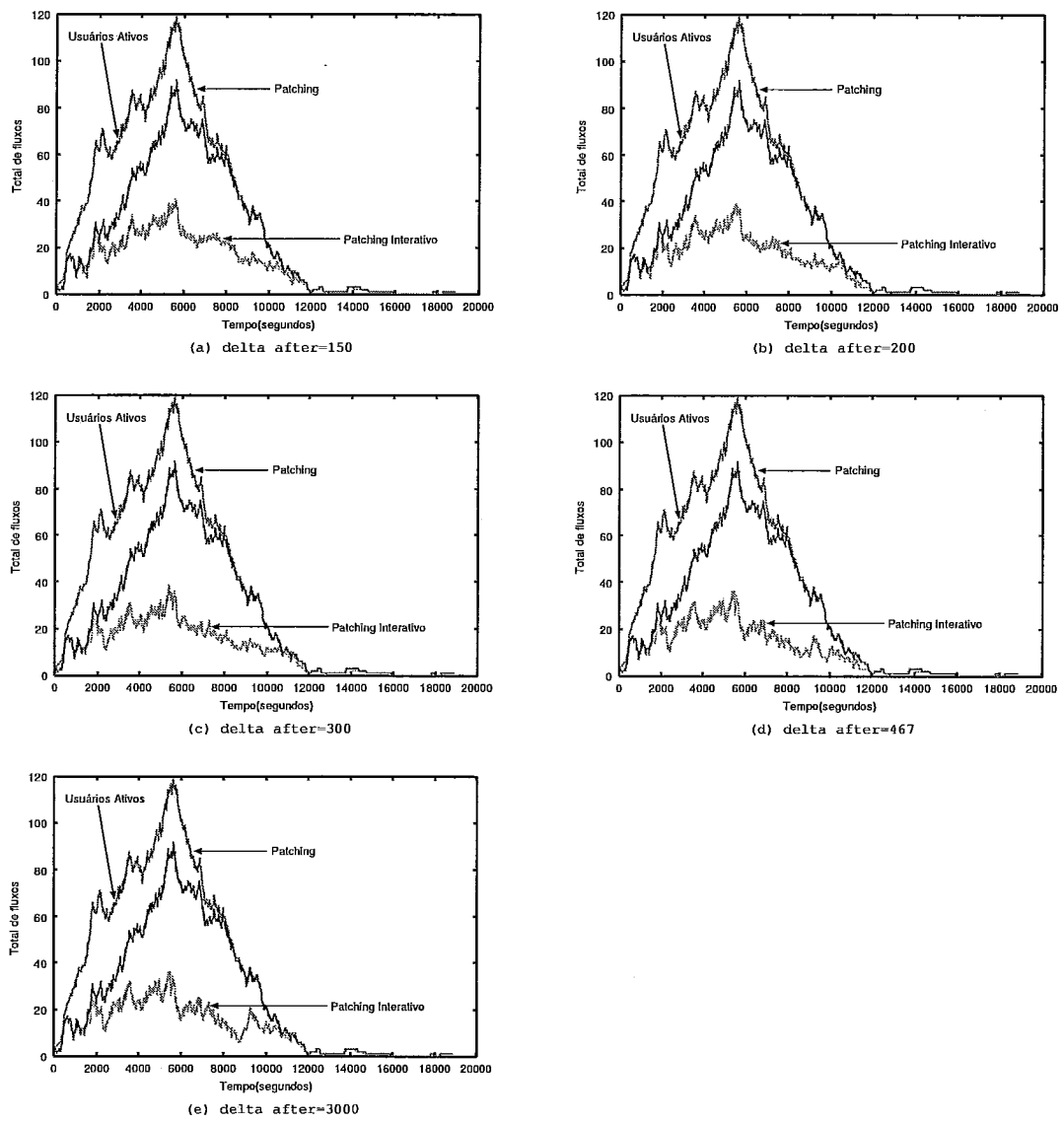


Figura 5.20: Total de fluxos no cenário 6

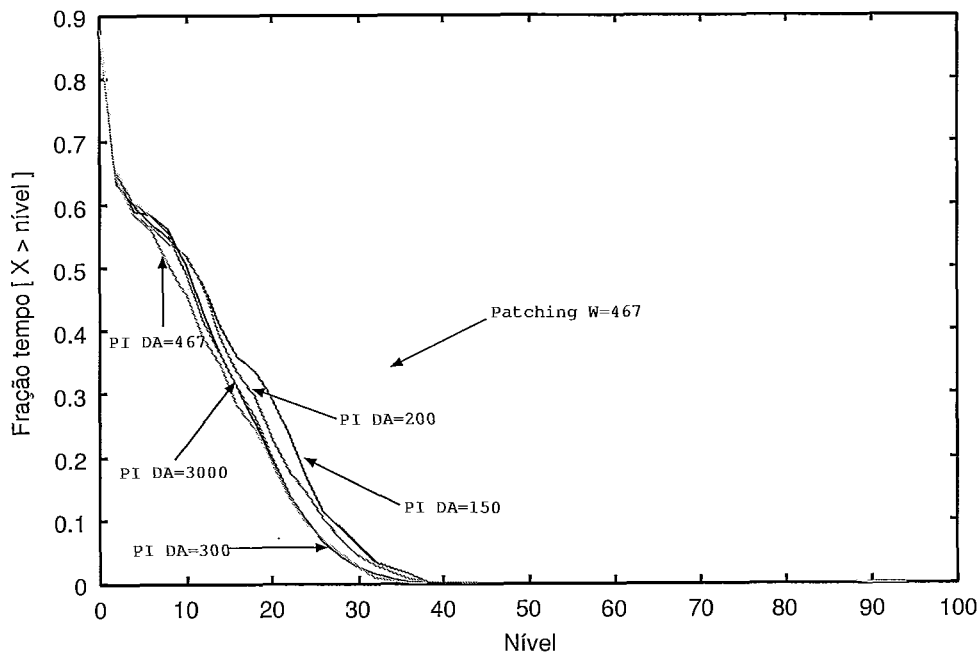


Figura 5.21: Gráfico fração de tempo no cenário 6

mesmo fato ocorreu no cenário 5, pois os picos dos deltas 360 e 570 ficaram muito próximos, 34 e 35 respectivamente, e as curvas, mostradas nas Figuras 5.17 e 5.18, também se comportaram de forma muito parecida.

Cenário/ <i>Delta After</i>	1	2	3	4	5	6
60	24	37	46	50	48	-
150	-	-	-	-	-	39
200	-	-	32	-	-	41
300	-	-	31	-	-	39
360	24	24	34	39	34	-
467	-	-	-	-	-	36
570	24	-	33	40	35	-
790	-	24	-	-	-	-
3000	24	33	54	46	48	36
<i>Patching Original</i>	24	66	103	94	99	92
Usuários Ativos	185	71	112	138	120	119

Tabela 5.2: Número máximo de fluxos ativos das técnicas *Patching* e *Patching Interativo*

Na tabela 5.3 pode-se observar a eficácia do método de *Patching Interativo*,

quando comparado com os métodos *Patching* Original e Usuários Ativos, denominados na tabela respectivamente de PO e UA. As informações contidas na tabela 5.3 são relativas aos picos de fluxos ativos para cada uma das técnicas em cada cenário. Os dados apresentados em cada célula da tabela representam o percentual de banda economizada com a utilização do método de *Patching* Interativo em relação ao *Patching* Original e ao número de usuários ativos para todos os cenários avaliados. Desta forma podemos observar, por exemplo, que o *Patching* Interativo economizou 69.9% da banda utilizada para a transmissão dos dados no cenário 3 em relação ao *Patching* Original. Em todos os cenários avaliados o *Patching* Interativo se mostrou melhor ou igual ao *Patching* Original e se apresentou sempre melhor que o número de usuários ativos.

Cenários	1		2		3		4		5		6	
$\Delta_{After}$	PO	UA	PO	UA	PO	UA	PO	UA	PO	UA	PO	UA
60	0	670	43.9	47.8	55.3	58.9	46.8	63.7	51.5	60	-	-
150	-	-	-	-	-	-	-	-	-	-	57.6	67.2
200	-	-	-	-	68.9	71.4	-	-	-	-	55.4	65.5
300	-	-	-	-	69.9	72.3	-	-	-	-	57.6	67.2
360	0	670	63.6	66.1	66.9	69.6	58.5	71.7	65.6	71.6	-	-
467	-	-	-	-	-	-	-	-	-	-	60.8	69.7
570	0	670	-	-	67.9	70.5	57.4	71	64.6	70.8	-	-
790	-	-	63.6	66.1	-	-	-	-	-	-	-	-
3000	0	670	50	53.5	47.5	51.7	51	66.6	51.5	60	60.8	69.7

Tabela 5.3: Economia de banda em relação aos picos em porcentagem



# Capítulo 6

## Conclusões

A demanda por aplicações que geram mídias contínuas vem crescendo no decorrer dos anos, impondo a necessidade de aperfeiçoamento das técnicas para melhoria da QoS oferecida aos usuários e ao mesmo tempo o aumento da escalabilidade dessas aplicações. Para aumentar a escalabilidade, diversas técnicas tem sido propostas. Estas técnicas são baseadas na transmissão *multicast* para o compartilhamento dos fluxos de dados enviados aos clientes e também na capacidade do usuário receber dados em mais de um canal e armazená-los até o momento de serem exibidos.

Neste trabalho, foi apresentada uma proposta para o compartilhamento dos fluxos considerando que os usuários podem executar comandos, tais como: avançar, retroceder e pausar. Este mecanismo, chamado de *Patching* Interativo, é baseado na técnica *Patching*. O *Patching* foi escolhido pois oferece um serviço imediato ao cliente, é bastante simples e possui um bom desempenho se comparada a algumas outras propostas.

No mecanismo proposto, cada vez que o cliente executa um comando, ou seja, realiza um acesso não seqüencial ao fluxo, é examinado se este cliente pode se juntar a alguma transmissão *multicast* em curso. A adição deste cliente a um grupo *multicast* só é realizada se o bloco requisitado por ele estiver dentro de uma janela de tempo anterior ou posterior ao último bloco transmitido para o grupo *multicast*. Além disso, todos os blocos recebidos pelo cliente são armazenados localmente. Desta forma, cada vez que o cliente deseja exibir um bloco novamente, não é necessário

recuperá-lo no servidor.

Resumindo, as principais contribuições deste trabalho foram o desenvolvimento e implementação de um novo cliente e do mecanismo *Patching* Interativo que foram detalhados, respectivamente, nos capítulos 4 e 5 desta dissertação. O desenvolvimento foi feito mantendo a compatibilidade com os procedimentos anteriores, ou seja, o cliente pode escolher a conexão direta ao servidor, recebendo os dados apenas via *unicast*, ou irá se conectar ao PI, onde receberá os dados via *multicast* e apenas o *patch* via *unicast*.

Foi desenvolvido, também, um modelo para a simulação da proposta do *Patching* Interativo, onde as características inseridas no cliente e no servidor podem ser avaliadas para a melhor configuração dos parâmetros do servidor e do cliente. Este modelo tem como entrada um *trace* que representa o comportamento de um cliente que realiza operações de avançar, pausar, retroceder. Os resultados obtidos no capítulo 6 puderam mostrar que o número total de fluxos abertos no servidor para a técnica do *Patching* Interativo é bem menor que o número de fluxos para a técnica de *Patching* e para o mecanismo mais simples que envia um fluxo *unicast* para cada cliente. Com isso, conclui-se que a técnica proposta possui desempenho bastante superior ao *Patching* original permitindo economia de banda de uma ordem de grandeza na maioria dos cenários avaliados.

Este trabalho deverá ser usado para educação à distância, através do CEDERJ, em vários pólos dentro do estado do Rio de Janeiro. Os alunos também terão à sua disposição os *slides* das aulas, já sincronizadas com o vídeo, para o seu melhor entendimento. Eles poderão, também, interagir com o vídeo através da interface gráfica, executando comandos como avançar e retroceder. Os *slides* estão sincronizados com o vídeo garantindo maior facilidade ao usuário.

## 6.1 Trabalhos Futuros

Algumas linhas de atuação podem estender este trabalho, como por exemplo:

- 
- Implementação de outras técnicas de compartilhamento de banda no servidor RIO;
  - Desenvolvimento de um modelo analítico para o cálculo das janelas de interatividade, *delta\_before* e *delta\_after*;
  - Obter resultados do modelo usando *logs* reais, coletados através do servidor RIO;

# Referências Bibliográficas

- [1] A. DAN AND D. SITARAM AND P. SHAHABUDDIN. Scheduling policies for an on-demand video server with batching. In *Proceedings of the second ACM international conference on Multimedia* (1994), pp. 15–23.
- [2] AGGARWAL, C. C., WOLF, J., E YU, P. On optimal batching policies for video-on-demand storage servers. In *Proceedings of the IEEE Conf. on Multimedia Systems* (1996), pp. 253–258.
- [3] AGGARWAL, C. C., WOLF, J. L., E YU, P. S. On optimal piggyback merging policies for video-ondemand systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1996), pp. 200–209.
- [4] ALMEIDA, J. M., EAGER, D. L., FERRIS, M., E VERNON, M. K. Provisioning content distribution networks for streaming media. In *Proceedings of Infocom* (2002).
- [5] BERSON, S., MUNTZ, R. R., E WONG, W. R. Randomized Data Allocation for Real-time Disk I/O. In *First IEEE Computer Society International Conference (COMPCON'96)* (1996), pp. 286–290.
- [6] C. C. AGGARWAL AND J. L. WOLF AND P. S. WU. A permutation-based pyramid broadcasting scheme for vodo-on-demand systems. In *IEEE Multimidia Systems* (1996).
- [7] CAI, Y., HUA, K., E VU, K. Optimizing patching performance. In *Conference on Multimedia Computing and Networking* (January 1999), pp. 204–215.

- [8] CARMO, R., DE CARVALHO, L., DE SOUZA E SILVA, E., DINIZ, M., E MUNTZ, R. Performance/Availability Modeling with the TANGRAM-II Modeling Environment. *Performance Evaluation* 33 (1998), 45–65.
- [9] CARTER, S. W., LONG, D. D. E., MAKKI, K., NI, L. M., SINGHAL, M., E PISSINOU, N. Improving video-on-demand server efficiency through stream tapping. In *Proc. 6 International Conference on Computer Communications and Networks* (1997), pp. 200–207.
- [10] CEDERJ. Centro de Educação de Ensino Superior a Distância do Estado do Rio de Janeiro. URL <http://www.cederj.rj.gov.br>.
- [11] CHENG, W. C. Tangram graphic interface facility. URL <http://bourbon.cs.umd.edu:8001/tgif/>.
- [12] D. EAGER AND M. VERNON AND J. ZAHORJAN. Minimizing Bandwidth Requirements for On-Demand Data Delivery. *IEEE Transactions on Knowledge and Data Engineering* 13, 5 (September 2001), 742–757.
- [13] DAN, A., SITARAM, D., E SHAHABUDDIN, P. Dynamic batching policies for an on-demand video server. *ACM Multimedia Systems* (1996), (4):112–121.
- [14] DE QUEVEDO CARDOZO, A. Mecanismos para Garantir Qualidade de Serviço de Aplicações de Vídeos sob Demanda. Tese de Mestrado, COPPE/UFRJ, 2002.
- [15] DE SOUZA E SILVA, E., LEAO, R. M., E MUNTZ, R. R. Analytical, simulation and measurement techniques: Experiences with an integrated modeling environment. In *Performance Evaluation*. 2003.
- [16] DE SOUZA E SILVA, E., LEAO, R. M., RIBEIRO-NETO, B., E CAMPOS, S. Performance issues of multimedia applications. In *Performance Evaluation of Complex Systems: Techniques and Tools*, vol. 2459 of *Lecture Notes in Computer Science*. Springer, 2002, pp. 374–405.

- [17] EAGER, D., E VERNON, M. Dynamic skyscraper broadcasts for video-on-demand. In *4th International Workshop on Multimedia Information Systems* (September 1997), pp. 89–100.
- [18] EAGER, D., VERNON, M., E ZAHORJAN, J. Bandwidth skimming: A technique for cost-effective video-on-demand. In *Proc. 2000 Multimedia Computing and Networking* (Jan 2000).
- [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] GAO, L., E TOWSLEY, D. F. Supplying instantaneous video-on-demand services using controlled multicast. In *Proc. of IEEE International Conference on Multimedia Computing and Systems* (1999), pp. 117–121.
- [21] GEREOFFY, A. mplayer - Movie Player for Linux. URL <http://www.mplayerhq.hu/homepage/>.
- [22] GORZA, M. L. Uma técnica de compartilhamento de recursos para transmissão de vídeo com alta interatividade e experimentos. Tese de Mestrado, UFF, 2003.
- [23] GUO, Y., SEN, S., E TOWSLEY, D. Prefix caching assisted periodic broadcast: Framework and techniques to suport streaming for popular videos. In *Proc. of ICC* (2002).
- [24] 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.
- [25] HUA, K. A., E SHEU, S. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. In *SIGCOMM* (1997), pp. 89–100.
- [26] JIN, S., E BESTAVROS, A. Scalability of multicast delivery for non-sequential streaming access. In *Proceedings of Sigmetrics'2002: The ACM International Conference on Measurement and Modeling of Computer Systems* (2002).
- [27] KUROSE, J. F., E ROSS, K. W. *Computer networking: a top-down approach featuring the Internet*, 2<sup>nd</sup> ed. Addison Wesley Longman, EUA, 2002.

- [28] LAND. *Laboratory for modeling, analysis and development of networks and computing systems*. URL <http://www.land.ufrj.br>, April 2001.
- [29] LEÃO, R. M., E DE SOUZA E SILVA, E. The TANGRAM-II Environment. In *Computer Performance Evaluation / TOOLS* (2000), vol. 1786 of *Lecture Notes in Computer Science*, Springer, pp. 366–369.
- [30] MA, H., E SHIN, K. G. A new scheduling scheme for multicast true vod service. In *Lecture Notes in Computer Science PCM* (2001), pp. 708–715.
- [31] MAPPED. Mecanismos de Adaptação e Controle para Recuperação e transmissão Multimídia com Aplicação ao Centro de Educação Superior a Distância do Estado do Rio de Janeiro.
- [32] REJAIE, R., HANDLEY, M., E ESTRIN, D. Multimedia proxy caching mechanism for quality adaptative streaming applications in the internet. In *Proc. of IEEE Infocom* (2000), pp. 980–989.
- [33] S. VISWANATHAN AND T. IMIELENSKI. Pyramid broadcasting for video on-demand service. In *IEEE Multimedia Computing and Networking* (1995), pp. 66–77.
- [34] SANTOS, J., MUNTZ, R., E RIBEIRO-NETO, B. Comparing Random Data Allocation and Data Striping in Multimedia Storage Servers. In *Proceedings of ACM SIGMETRICS* (2000), pp. 44–55.
- [35] SANTOS, J. R. G. *RIO: A Universal Multimedia Storage System Based on Random Data Allocation and Block Replication*. PhD thesis, UCLA, 1998.
- [36] SEN, S., REXFORD, J., E TOWSLEY, D. Proxy prefix caching for multimedia streams. In *Proc. of IEEE Infocom* (1999), pp. 1310–1319.
- [37] TAN, H., E ET AL. Delimiting the range of effectiveness of scalable on-demand streaming. In *Performance Evaluation* (2002), pp. 387–410.
- [38] WANG, B., SEN, S., ADLER, M., E TOWSLEY, D. Optimal proxy cache allocation for efficient streaming media distribution. In *Proceedings of IEEE Infocom* (2002), pp. 1726–1735.