DISSERTAÇÃO

UFRJ

Dissertação submetida para a obtenção do título de

Mestre em Ciências em

Engenharia de Sistemas e Computação

ao Programa de Pós-Graduação de Engenharia de Sistemas e Computação

da COPPE/UFRJ

por

Ebenézer Rangel Botelho

Maio 2008

PATCHING INTERATIVO EFICIENTE: IMPLEMENTAÇÃO E ANÁLISE DE DESEMPENHO

Ebenézer Rangel Botelho

DISSERTAÇÃO 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.ª Rosa Maria Meri Leão, Dr.

Prof. Aloysio de Castro Pinto Pedroza, Dr.

Prof. Célio Vinicius Neves de Albuquerque, Ph.D.

Prof. Felipe Maia Galvão França, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

MAIO DE 2008

BOTELHO, EBENÉZER RANGEL

Patching Interativo Eficiente: Implementação e Análise de Desempenho [Rio de Janeiro] 2008

XV, 126 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2008)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

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

I. COPPE/UFRJ II. Título (Série)

Agradecimentos

Em primeiríssimo lugar eu agradeço a Deus, autor da minha fé, maior responsável por tudo o que alcancei até hoje e que ainda tenho a alcançar pela frente. Tem sido cumprido constantemente em minha vida o significado literal do meu nome: “Até aqui o Senhor me tem ajudado”(I Sam. 7:12).

Agradeço aos meus pais, Eliezer e Marta, pelo apoio incondicional que têm me dado ao longo da minha vida, pelo exemplo de vida que são pra mim, pelas motivações, puxões de orelha e pelo amor que dia após dia têm me dado. O mesmo posso dizer dos meus irmãos, Vanessa, Júnior e Priscila, meus melhores amigos, melhores companheiros, meus melhores elos com o passado, com o presente e com o futuro.

À minha noiva, minha eterna namorada, Midori, por fazer toda a diferença na minha vida e por ser um apoio indispensável neste momento tão especial da minha vida.

Aos meus “amigos-de-toda-a-vida", Tatiana, Bruno(Jacob), Eduardo, Marcus, Cesar, Aninha e Ebenéser (acredite! é isso mesmo! Eu tenho um amigo chará!!!) simplesmente por existirem e me darem o prazer desta amizade forte e duradoura de tantos anos... Esta vitória também é em parte de vocês!

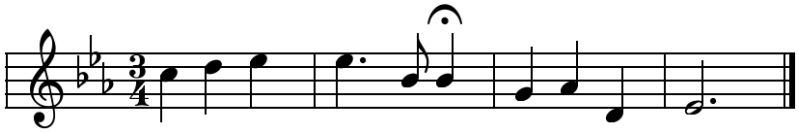
Aos amigos mais recentes, mas que já me apoiam como se nos conhecêssemos do ventre materno: estimadíssimo pr. Anderson, Karinna, Igor, Wagner, Cleber, Huelber e Irwing.

Aos companheiros do LAND, uma equipe seleta da qual tenho orgulho em fazer

parte: Fabrício, pelos muitos sudo's e helps em horas desesperadoras, sempre com a maior disponibilidade apesar de sobrecarregado; Hugo, Carolzinha, Edmundo e Boechat, na luta comigo há mais tempo; Fernando, Allyson, Guto, Fabiannnnne, Luiz, Bruno, Ariadne, Jefferson, Lucas e Ana Paula; Flávio (um cara prático e com as idéias de soluções mais diversificadas que já vi, vulgos "workarounds"); Bernardo, um cara que nunca vou esquecer pelo coleguismo, apoio e ajudas a todo tempo, toda hora, desde o início à conclusão do meu mestrado... Valeu cara!!!; como não poderia deixar de ser, à CALÓL!!! Mãezona de todos, conselheira, multitarefa, amiga, ..., peça fundamental no LAND...

Reconheço, ainda, a importância significativa que os meus orientadores Edmundo Albuquerque de Souza e Silva, o mais novo imortal, e Rosa Maria Meri Leão tiveram não só na minha formação acadêmica, mas na minha formação profissional e pessoal. É um orgulho e privilégio tê-los como orientadores e poder participar de seu time.

Por fim, agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), à Fundação Centro de Ciências e Educação Superior a Distância do Estado do Rio de Janeiro (Fundação CECIERJ) e à Financiadora de Estudos e Projetos (FINEP) pelo financiamento deste estudo através de bolsas de mestrado, pesquisa e projeto respectivamente.



"A benignidade do Senhor jamais acaba, as Suas misericórdias não têm fim; renovam-se a cada manhã. Grande é a Tua fidelidade."

(Lm 3.22,23)

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

PATCHING INTERATIVO EFICIENTE: IMPLEMENTAÇÃO E ANÁLISE DE DESEMPENHO

Ebenézer Rangel Botelho

Maio/2008

Orientadores: Edmundo Albuquerque de Souza e Silva
Rosa Maria Meri Leão

Programa: Engenharia de Sistemas e Computação

Com a maior popularidade dos cursos de ensino a distância que fazem uso de transmissão de vídeos, vários estudos vêm sendo feitos e novas técnicas desenvolvidas para garantir a qualidade e aumentar a escalabilidade deste serviço onerando o mínimo possível a banda da rede. As técnicas propostas prevêem que o acesso aos vídeos não é feito de forma seqüencial, como acontece com vídeos de conteúdo informativo e de entretenimento, mas é passível de muitas interações por parte dos usuários.

A variedade de técnicas é grande, mas a maioria delas se baseia em transmissões compartilhadas, onde um fluxo de dados é enviado para um ou mais clientes. Esses tipos de transmissão dividem-se em dois: *multicast* e *broadcast*. Este trabalho se foca em técnicas baseadas em transmissão *multicast*.

Implementamos uma técnica de compartilhamento de banda cujo algoritmo tem o nome de "Pathing Interativo Eficiente(PIE)". Comparamos o seu desempenho com outra técnica da literatura, a técnica de Patching Interativo (PI) e também com a transmissão *unicast*. Foi observado que a economia de banda pode ser significativa com relação à transmissão *unicast* para diversos níveis de interatividade de clientes.

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

EFFICIENT INTERACTIVE PATCHING: IMPLEMENTATION AND PERFORMANCE EVALUATION

Ebenézer Rangel Botelho

May/2008

Advisors: Edmundo Albuquerque de Souza e Silva
Rosa Maria Meri Leão

Department: Systems Engineering and Computer Science

With the increasing popularity of the distance learning courses that use video transmission, several studies have been done and new techniques have been developed to guarantee the quality and to increase the scalability of this service is network bandwidth. The proposed techniques assume that video access is not sequential as in entertainment videos, but it is subject to many user interactions.

There are several techniques and the majority of them are based on the bandwidth sharing using multicast techniques or broadcast transmission.

This work focus on multicast transmission. We implemented a bandwidth sharing technique called "Efficient Interactive Patching (EIP)". We compared the performance of EIP with that the Interactive Patching (IP) and also with the unicast transmission.

The results show that the bandwidth required by EIP is significantly less than that required by the unicast transmission for several client interactive levels.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Introdução

Transmissão de fluxos multimídia estão cada vez mais freqüente na internet. Com a grande popularidade de sites como o *youtube* [Youtube 2008] e a crescente transmissão de áudio e vídeo pela rede mundial, é cada vez mais presente a preocupação com o que se chama "colapso mundial na internet". Alguns estudos, entre eles o feito pela empresa *Nemertes Research Group* [Group 2008a], prevêem este colapso para o ano de 2010 alegando que a atual estrutura provavelmente não comportará o volume de dados nos próximos anos.

Diversas técnicas têm sido elaboradas nos últimos anos que tentam minimizar o impacto destas transmissões através do compartilhamento de fluxos de áudio e vídeo, onde através de uma única transmissão atende-se mais de um usuário.

O objetivo deste estudo é implementar e analisar o desempenho de um algoritmo de compartilhamento de fluxo de vídeo, elaborado em [Rodrigues 2006], bem como compararmos o desempenho desta técnica com a transmissão *unicast* e com a técnica de *patching* interativo, proposto em [Netto 2004].

O contexto deste trabalho é o de transmissão de vídeos de um curso de ensino a

distância, o CEDERJ (**C**entro de **E**ducação superior a **D**istância do **E**stado do **R**io de **J**aneiro) [CEDERJ 2008], onde diversos alunos se conectam a um único servidor multimídia para assistirem as aulas nele armazenadas. Este curso iniciou-se em 2005, quando tinha apenas 400 alunos distribuídos em 4 pólos. Hoje já são 1500 alunos distribuídos em 11 pólos e esta estrutura continua crescendo e chegará a um porte muito maior do que é hoje tendo em vista a amplitude de nível nacional que se quer alcançar com o projeto em andamento da Universidade Aberta do Brasil.

Os vídeos armazenados neste servidor estão codificados em uma taxa de aproximadamente 1.1 Mbps o que nos permitiria atender menos de 90 clientes simultâneos em uma rede de 100 Mbps considerando uma utilização de 100% de um canal (o que é impossível como pode ser visto em [Kurose e Ross 2000]) ideal (sem perdas). Com a utilização de transmissão *multicast*, podemos atender quantidades significativamente maiores de usuários simultâneos, principalmente se o acesso ao vídeo for seqüencial.

As técnicas de otimização de banda da rede na transmissão de fluxos de multimídia atuam de forma que estes dados sejam distribuídos de forma mais econômica, usando-se *multicast* ou difusão (este último muito comum nas transmissões de TV-a-cabo). Com o avanço das aplicações multimídia, tem sido cada vez mais comum o uso da *internet* no ensino a distância, não só através de arquivos de texto mas também de fluxos de aúdio e vídeo.

Um usuário de um serviço de ensino a distância difere-se de um outro que assiste a um filme de entretenimento no que diz respeito à interatividade. Este último tende a assistir o vídeo ou escutar o áudio seqüencialmente, do início ao fim sem muitas interações. Isto já não acontece com o primeiro tipo de usuário. Geralmente um aluno tem dúvidas ao longo da aula e tende a querer ouvir/assistir novamente aquele trecho que não entendeu. Outro comportamento bem comum é o aluno pular uma parte da aula que já domine. Há, ainda, casos em que o aluno quer ouvir um trecho específico da aula para sanar uma dúvida pontual. Todos estes cenários caracterizam o acesso ao vídeo como interativo e requer um tratamento diferenciado

pois o compartilhamento de fluxos deste tipo não é tão simples e trivial quanto o seqüencial.

Serão comparadas, neste trabalho, duas técnicas que tentam diminuir o consumo de banda de rede em acessos a vídeos que demandam interatividade por parte de seus usuários: *Patching Interativo*(PI)[Netto 2004] e *Patching Interativo Eficiente*(PIE)[Rodrigues 2006].

1.2 Motivações e objetivos

O LAND(**L**aboratório de **AN**álise e **D**esenvolvimento de redes e sistemas de computação) [Land 2008] vem, desde 2005, atuando com o CEDERJ através de uma parceria. O CEDERJ utiliza o servidor multimídia RIO, atualmente desenvolvido no LAND, em seu curso a distância de tecnologia em sistemas de computação de nível superior. Este curso está distribuído em diversos pólos no interior do estado do Rio de janeiro cujos laboratórios de informática são equipados cada um com uma instância própria deste servidor.

O número de alunos inscritos no curso do CEDERJ vem crescendo e com isso é importante aumentar a escalabilidade do servidor RIO. A versão do servidor que encontra-se operacional nos pólos é baseada na transmissão de um fluxo de vídeo para cada usuário que se conecta ao serviço. Portanto, considerando que cada fluxo é transmitido com taxa média de 1.1Mbps, a conexão do servidor com a rede pode ser um possível gargalo para o serviço.

No final de 2007 o LAND teve um projeto aprovado pela Rede Nacional de Ensino e Pesquisa (RNP)[RNP 2008] cujo objetivo é disponibilizar conteúdo educacional usando o servidor RIO para os usuários da RNP. Inicialmente serão disponibilizados os cursos do CEDERJ. Instâncias do servidor serão executadas nos POP's da RNP. Certamente nesta aplicação o número de usuários acessando o serviço pode ser bem maior do que os alunos de um pólo do CEDERJ, o que motiva a implementação de

técnicas para aumentar a escalabilidade do servidor.

Mostraremos, baseados em resultados de experimentos, que o servidor RIO, já é escalável mesmo com transmissão *unicast*[1], e torna-se ainda mais poderoso, atendendo a demandas bem maiores, quando opera com transmissões *multicast*. Compararemos os desempenhos das técnicas PI e PIE entre si e destas com a transmissão *unicast*.

Dentre as principais contribuições deste trabalho encontra-se o aprimoramento do cliente de vídeo em receber tantos fluxos *multicast* quanto as técnicas utilizadas determinarem, a adição de mais uma técnica de otimização de banda de rede no servidor e a implementação de uma variedade de ferramentas complexas que possibilitam a emulação deste ambiente onde são executados quantos clientes forem necessários e os mesmos são coordenados de uma máquina central de forma simples e totalmente automatizada.

1.3 Organização do texto

Terminada a apresentação do objetivo e motivação do trabalho entraremos no segundo capítulo deste texto. Lá trataremos dos conceitos necessários ao entendimento do sistema RIO no nível de arquitetura e de funcionamento. Apresentaremos algumas técnicas de compartilhamento de banda existentes, detalharemos as que são objetivo deste estudo e mostraremos como elas são implementadas no sistema RIO. Ainda sobre este sistema, será dado um breve histórico de sua existência, de sua arquitetura e mostraremos como os componentes interagem entre si através de mensagens de controle.

No Capítulo 3, descreveremos o ambiente de emulação, as máquinas que o compõem, o funcionamento do emulador do cliente de vídeo e o conjunto de *scripts*

[1] Em [Valle 2007] é mostrado que o servidor RIO aumenta consideravelmente sua capacidade de atendimento conforme aumentam suas unidades de armazenamento combinado com o uso adequado de réplicas de dados.

utilizados no disparo e administração das emulações. Descreveremos, ainda, a ferramenta utilizada na análise do tráfego de dados entre as máquinas envolvidas na emulação e a topologia da rede utilizada nos experimentos. Para finalizar o capítulo, descreveremos a metodologia utilizada.

No quarto capítulo detalharemos os experimentos feitos e faremos as comparações entre as técnicas e os cenários a partir dos resultados obtidos nas emulações.

No quinto capítulo apresentamos as nossas conclusões e possíveis trabalhos futuros.

Capítulo 2

Compartilhamento de banda e o Sistema Multimídia RIO

2.1 Introdução

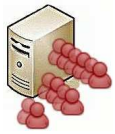
Há pouco tempo atrás diríamos que aplicações multimídia em tempo real seriam uma tendência das próximas gerações. Hoje já não dizemos isto pois estamos vivendo esta realidade; temos um incontável número de recursos de áudio e vídeo disponíveis na internet e uma adesão cada vez maior dos usuários por este tipo de entretenimento.

Nesta onda de aplicações multimídia temos, também, os cursos de ensino a distância que, em um futuro próximo, demandarão uma capacidade cada vez maior de transmissão de dados e uma necessidade de compartilhamento desta capacidade visando melhorar a escalabilidade. Abaixo apresentaremos algumas das técnicas propostas na literatura para compartilhamento de recursos e, em seguida, o sistema multimídia RIO.

2.1.1 Legenda das figuras

Ao longo deste texto usaremos imagens que representarão os elementos do sistema RIO e alguns equipamentos utilizados nesta estrutura. Estes elementos, cliente, *Patching Layer, dispatcher* e *storage* serão apresentados e explicados no decorrer do texto bem como a interação destes entre si e com os demais equipamentos. Na Figura 2.1 exibimos estas imagens em uma legenda que deverá ser usada como apoio na leitura deste documento.

Cliente de vídeo - riommclient.



Patching Layer - PL.

Dispatcher.

Nó de armazenamento - Storage.

Servidor RIO = Dispatcher + Storage's =  **+**  **.**

NeTraMeT

Switch da rede local.

Figura 2.1: Legenda das imagens do texto.

2.2 Algumas Técnicas de compartilhamento de fluxo de vídeo

Existem diferentes serviços multimídia. Alguns são de tempo real, outros fornecem o serviço sob demanda; alguns aceitam interação do usuário enquanto outros apenas transmitem do início ao fim ou até que o usuário se dê por satisfeito. Por causa dos diferentes tipos de serviço, temos diversas técnicas de compartilhamento de banda que se aplicam cada uma a seu devido contexto. Estas técnicas dividem-se em três grupos: mecanismos de difusão periódica, técnicas orientadas às requisições dos clientes e transmissões p2p (*peer-to-peer*)[1]. Na figura 2.2 temos o resultado de uma pesquisa feita pela empresa alemã iPoque [iPoque 2008] entre agosto e setembro de 2007 na qual se mostra a participação de diferentes tipos de fluxos de dados na rede mundial de computadores. Note que o tráfego relativo a fluxos de vídeo, com seus apenas 8.26%, desconsiderando-se os fluxos de vídeo gerados pelo *skype* ou por sites como o *youtube* [Youtube 2008], chegam, no período a que se refere a figura 2.2, a aproximadamente 300 *terabytes*[2]. Para se ter uma idéia do que isto representa em termos de massa de dados, levaríamos, em uma conexão a 100Mbps(Mega bits por segundo) livre de ruídos, colisões, perdas ou interferências, mais de nove meses para transmitirmos esta mesma quantidade de dados entre duas máquinas conectadas diretamente uma à outra[3].

Abaixo apresentamos algumas das técnicas que se aplicam nesses contextos.

1. **Mecanismos de difusão periódica**

   Os mecanismos de difusão periódica são ideais para vídeos muito populares pois, independente do número de usuários sendo atendidos, requerem uma

[1] Par-a-par.

[2] Na pesquisa estimou-se o fluxo total em aproximadamente 3 *petabytes*. Um *petabyte* equivale a 2^{10} *terabytes*, que por sua vez equivalem a 2^{40} bytes

[3] Quando sugerimos a conexão direta entre duas máquinas o fazemos visando um tempo ótimo, isto é, descartamos a contribuição de potenciais retardos devido aos roteadores no caminho entre as máquinas.

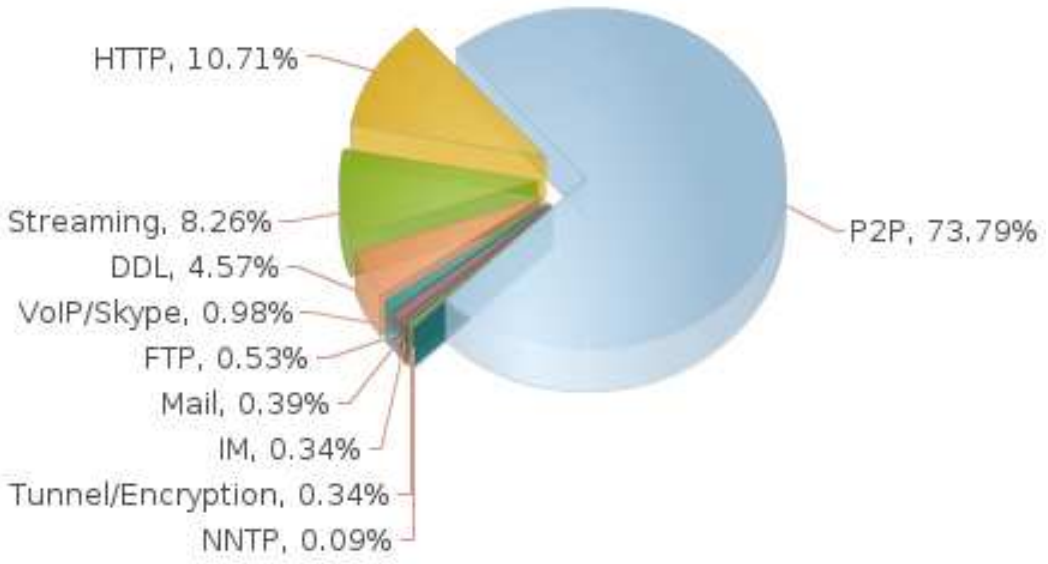


Figura 2.2: Consumo mundial do tráfego da internet[iPoque 2008].

largura de banda constante.

De acordo com [Zafalão 2004], os mecanismos de difusão periódica se dividem em três famílias: a família piramidal, onde os segmentos[4] são transmitidos a uma largura de banda constante com segmentos de tamanho crescente; a família harmônica, onde os segmentos possuem tamanho igual e largura de banda decrescente; e a terceira família que é um híbrido das anteriores.

Para entendermos melhor vamos analisar um destes mecanismos: *Skyscraper Broadcasting*. Esta técnica [Hua e Sheu] propõe que cada segmento seja continuamente transmitido à taxa do vídeo em um canal. Seu objetivo é atender a entrega dos vídeos sem que o cliente tenha que ter uma banda muito larga e alta capacidade de armazenamento: neste protocolo apenas dois canais são usados simultaneamente (no máximo) e o espaço requerido para o armazenamento temporário é compreendido entre três e trinta por cento do tamanho total do vídeo.

[4]Define-se segmento por uma parte do objeto a ser transmitido. Esta divisão é feita pela quantidade de *bytes* e pode ou não ser feita em partes iguais, dependendo da técnica aplicada.

A segmentação do canal é crescente e se dá pela seguinte função:

$$f(n) = \begin{cases} 1 & n = 1 \\ 2 & n = 2, 3 \\ 2f(n-1)+1 & n \mod 4 = 0 \\ f(n-1) & n \mod 4 = 1 \\ 2f(n-1)+2 & n \mod 4 = 2 \\ f(n-1) & n \mod 4 = 3, \end{cases}$$

o que nos dá a seguinte série: 1, 2, 2, 5, 5, 12, 12, 25, 25, 52, 52, ... sendo o maior deles igual a W, um valor fornecido como parâmetro ao protocolo, informando o tamanho máximo do segmento. Cada segmento S_i equivale, portanto, a $S_i = \min\{f(i)S_1, W\}$.

Note que da forma como esta função de crescimento foi idealizada, o requisito de banda sempre será atendido pois o cliente não disputa pelo canal pois a transmissão é por difusão e, por outro lado, nunca haverá um momento em que o cliente não tenha dados já recebidos para serem consumidos. Isto fica mais claro de ser entendido pelo exemplo da figura 2.3, onde b é a taxa de consumo do vídeo. Nesta figura o cliente entra no sistema no momento em que um segmento S_1 está sendo transmitido, a janela w. Sendo assim ele aguarda o início da próxima transmissão deste segmento para começar a receber os dados do mesmo, o que acontece tão logo ele entre na segunda janela w. Neste momento ele apenas guarda em *buffer* os dados do segmento recebido até que esta janela termine. Quando está na terceira janela o cliente inicia o consumo do objeto ao mesmo tempo que começa a receber os dados dos segmentos S_2 e S_3. Ao término desta janela o cliente terminou de exibir o segmento S_1, que será descartado, inicia a execução dos dados que já tem de S_2 e continua recebendo o complemento dos segmentos S_2 e S_3. Como pode ser percebido na figura o cliente sempre terá dados em *buffer* prontos pra consumo imediato e a banda requisitada nunca será superior a $2b$, o que lhe garante ter sempre suas requisições de banda atendidas.

Uma característica importante é observada nos mecanismos de difusão onde

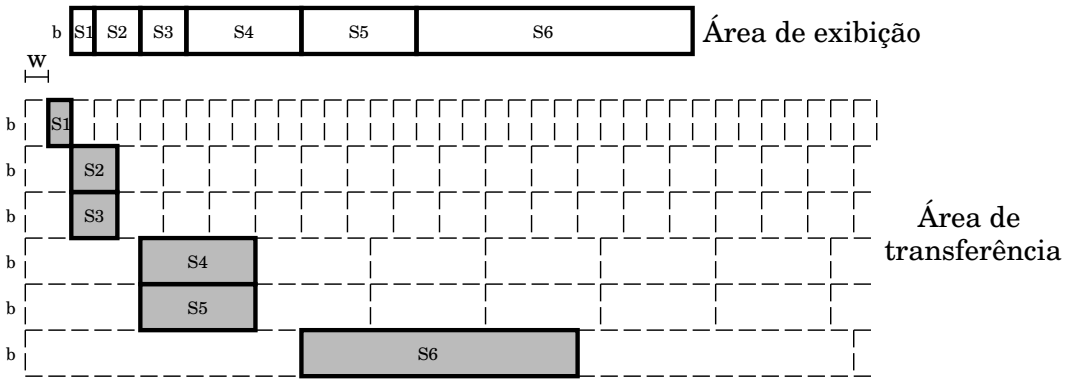


Figura 2.3: Técnica de *skyscraper*.

não se espera a ocorrência do início do segmento para receberem dados: por não desperdiçarem a banda passante, são candidatos a se aproximar do limite teórico[5].

2. **Técnicas orientadas às requisições dos clientes**

   Assim como os mecanismos de difusão periódica, as técnicas orientadas às requisições dos clientes têm como objetivo a redução dos requisitos de banda. Para isto elas atuam de forma a transmitir os objetos conforme o surgimento dos clientes tentando agrupá-los sempre que possível. As técnicas mais conhecidas são: *batching, piggybacking, hierarchical stream merging, patching*. Abordaremos sucintamente algumas delas e daremos mais atenção às duas últimas por se tratarem do objeto de estudo deste trabalho.

   (a) *Batching*[A. Dan e Shahabuddin , Dan et al.]

   A técnica de *batching* consiste em reter requisições por um determinado intervalo de tempo guardando mais clientes que queiram assistir o mesmo trecho de vídeo com o objetivo de atender o maior número de clientes possível com um só fluxo. Esse intervalo de tempo recebe o nome de **Janela** ou **Intervalo de** *Batching* e é o principal responsável pela latência inicial de exibição do vídeo pelo cliente.

[5]No exemplo da figura 2.3 isto significaria dizer que o cliente já receberia a parcela dos dados da primeira janela w enviados após sua chegada no sistema.

O funcionamento deste protocolo pode ser entendido melhor analisando-se a figura 2.4. O intervalo de *batching* é iniciado quando chega a primeira requisição do objeto, que no exemplo é a requisição r_0. A partir deste momento o servidor aguardará por um período W antes de iniciar o envio do vídeo de forma a atender com um único fluxo a todos os clientes que solicitarem este objeto entre o instante da chegada de r_0 e $r_0 + W$. No nosso exemplo novas requisições, r_1 e r_2, chegaram dentro deste intervalo e, com isto, r_0, r_1 e r_2 serão atendidos pelo mesmo fluxo s.

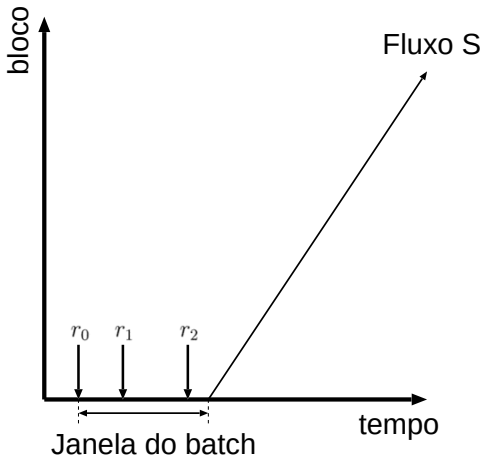


Figura 2.4: Técnica de *batching*.

Sendo $p_{u_{vcr}}$ a probabilidade de um usuário fazer alguma interação, toda vez que um grupo de n usuários é aceito no sistema, $n \times p_{u_{vcr}}$ canais devem ser reservados para transmitir os fluxos abertos após as interações dos usuários.

(b) *Piggybacking*[C. C. Aggarwal e Yu]

Na técnica de *piggybacking* elimina-se a latência inicial encontrada na técnica de *batching* pois os pedidos são atendidos imediatamente. A forma que esta técnica tem de fazer com que dois ou mais usuários compartilhem um mesmo fluxo é alterando a taxa de consumo de dados dos mesmos. Aproveitando-se do fato de que uma alteração na ordem de 5% na taxa de exibição de um vídeo não é perceptível aos humanos, esta técnica aumenta a taxa de exibição do vídeo do cliente que está mais atrasado

e a diminui no cliente mais adiantado até que eles se unam e passem a escutar um único fluxo de dados.

(c) *Patching*[K. A. Hua e Sheu]

Na técnica de *patching*, como na de *piggybacking*, o serviço é imediato e com latência bastante reduzida em comparação ao *batching*: na chegada da primeira requisição é aberto um fluxo *multicast* por onde este cliente é atendido imediatamente. Conforme novas requisições para o mesmo objeto e dentro de um limiar de tempo, denominado **janela ótima**, forem chegando, elas irão escutar este fluxo e receberão os blocos da parte inicial perdida a parte, via transmissão *unicast*, isto é, solicitarão estes blocos diretamente ao servidor e o receberão em um fluxo não compartilhado, o que chamamos de *patch*. Assim que um cliente recupera toda a parte perdida ele passa a escutar apenas o fluxo *multicast* originado pela primeira requisição e extingüe o fluxo *unicast*.

Independente de estar à frente ou atrás dos demais clientes de seu grupo, todos os clientes, diferentemente da técnica de *piggybacking*, tocam o vídeo na mesma taxa de exibição. Com isto é sabido que a banda consumida por um cliente é de até duas vezes a taxa de exibição do vídeo. Terminada a janela ótima, a primeira requisição que chegar após ela originará um novo fluxo *multicast*.

Na figura 2.5 [Rodrigues 2006] podemos acompanhar a dinâmica da técnica de *patching*: temos W, uma janela ótima de tamanho cinco e um objeto de tamanho total de dez unidades de tempo. No tempo t_0 temos uma requisição que não se encontra dentro de nenhuma janela ótima anteriormente aberta, o que faz com que seja aberto um novo fluxo *multicast* S_0. Nos instantes $t_1 = 1$, $t_2 = 3$ e $t_3 = 4$ temos novas requisições. Como todas elas aconteceram dentro da janela ótima do fluxo S_0, isto é, antes de $t_0 + W$, então elas escutarão o fluxo *multicast* S_0 além de fazerem seus *patches* para recuperar os blocos perdidos, gerando assim, cada uma delas, seu fluxo *unicast*, que serão, respectivamente, os fluxos P_1, P_2 e P_3

representados na figura 2.5.

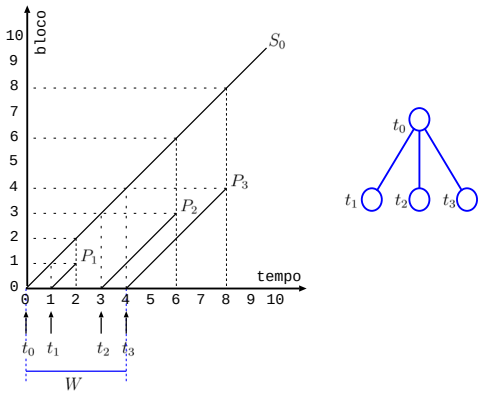


Figura 2.5: Técnica de *patching*.

Analisando a requisição que chegou em t_1, percebe-se que ela chegou uma unidade de tempo após a abertura do fluxo S_0 em t_0. Isto significa que P_1 levará uma unidade de tempo para se juntar ao fluxo S_0 e poder extingüir seu fluxo de *patch* P_1, o que realmente acontece na unidade de tempo igual a dois. Para a requisição que chega em t_2, visto que P_2 entrou três unidades de tempo após a abertura de S_0, então P_2 levará exatamente este tempo para alcançar o fluxo *multicast* e fará *patch* por três unidades de tempo, o que acontece em $t_2 + 3 = 6$, quando seu fluxo *unicast* P_2 será extinto e passará a escutar apenas o fluxo S_0. Para a requisição iniciada em t_3, quatro unidades de tempo após o início de S_0, teremos a extinção de P_3, seu *patch*, em $t_3 + 4 = 8$. Neste momento todos as requisições já alcançaram o fluxo *multicast* e estão escutando apenas um único fluxo.

Na figura 2.5 mostramos uma árvore onde todo nó filho escuta o fluxo do nó pai. Ela indica que as requisições que chegaram nos tempos t_1, t_2 e t_3 escutam o fluxo S_0 criado para a requisição que chegou em t_0, que é a raiz da ávore. O nível de profundidade desta árvore de requisições nunca excederá duas unidades pois cada cliente *multicast* tem apenas duas possibilidades: escutar apenas o fluxo relacionado à raiz da árvore (o que equivale no exemplo anterior ao fluxo S_0) ou escutar S_0 e o seu

fluxo de *patch*. Como dito anteriormente, isto limita a banda necessária de cada cliente a duas vezes a taxa de exibição do vídeo.

(d) **Hierarchical Stream Merging** (HSM)[D. L. Eager e Zahorjan]

Esta técnica é bastante semelhante à técnica de *patching* pois os clientes são atendidos imediatamente após suas requisições e, além de escutar seu próprio fluxo (o fluxo equivalente ao *patch* da técnica de *patching*), escutam um outro fluxo à frente, *target*, ao qual irão se unir futuramente. A diferença é que, aqui, este fluxo de *patch* também é *multicast*, o que permite que outros clientes também possam escutá-lo, e que quando um cliente alcança seu *target* ele elege um novo *target*, passa a escutá-lo e seu fluxo inicial é extinto. Como isto se repete indefinidamente, tem-se uma união hierárquica de fluxos em contraposição ao esquema de *patching*, onde a união de fluxos só ocorre uma vez para cada cliente. Uma diferença muito importante da técnica HSM para a técnica de *patching* é a de que toda vez que ocorre a união de dois fluxos é como se tudo iniciasse novamente, ou seja, é como se todos os membros deste novo grupo composto pelos dois antigos acabassem de entrar no sistema. Sendo assim, tudo aquilo que o grupo mais antigo estava recebendo via *multicast* de um terceiro grupo é descartado para que os grupos que se uniram andem juntos.

Na figura 2.6 repetimos os eventos do exemplo anterior (ver figura 2.5) da técnica de *patching* para analisarmos e compararmos com o que acontece na técnica HSM. Novamente temos um objeto com dez unidades de tempo e quatro chegadas que ocorrem nos instantes $t_0 = 0$, $t_1 = 1$, $t_2 = 3$ e $t_3 = 4$ gerando, respectivamente, os fluxos S_0, S_1, S_2 e S_3.

Acompanhando a figura 2.6, para a requisição que chega em t_0 é aberto o fluxo *multicast* S_0 e esta requisição recebe todo o objeto por este mesmo fluxo (isto só acontece pois não há um outro fluxo à frente de S_0 ao qual este fluxo possa se unir). Em t_1 há uma nova chegada que passa a receber os dados do fluxo S_0 e de um novo fluxo S_1 aberto para ela. Como S_1 foi

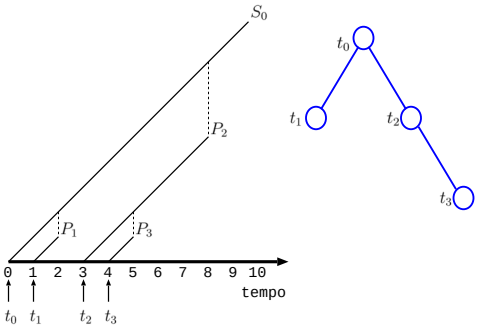


Figura 2.6: Técnica HSM.

iniciado uma unidade de tempo após S_0, então S_1 é extinto em $t_1 + 1 = 2$ e o cliente que chegou em t_1 passa a escutar apenas S_0. Neste caso nada acontece com os antigos clientes de S_0 pois eles não estavam recebendo dados de outro fluxo que não do seu próprio.

Uma nova requisição chega em t_2 e começa a receber os dados de S_0 e do novo fluxo S_2 criado para ela. Como esta nova conexão está apenas três unidades de tempo atrás de S_0, é de se esperar que ela alcance o fluxo S_0 exatamente três unidades a partir do instante t_2, mas isto não acontecerá pelo evento que se sucede. Em t_3 uma nova requisição chega e esta passa a escutar o fluxo S_2 e o novo fluxo S_3 criado para ela. Como a requisição que aconteceu em t_3 se deu apenas uma unidade de tempo após o início do fluxo S_2, S_3 levará apenas uma unidade de tempo para se unir completamente a S_2, o que acontece no instante $t_3 + 1 = 5$. Neste instante S_2 e S_3 tornam-se um só fluxo e, como dita o protocolo desta técnica: quando ocorre a união de dois fluxos é como se tudo se iniciasse novamente e funciona como se este novo grupo, composto pela união dos grupos dos fluxos S_2 e S_3, acabasse de entrar no sistema, isto é, como se este novo grupo entrasse no sistema em t=5. Ora, se isto realmente fosse verdade, então os dados que o fluxo S_2 recebeu de S_0 antes de se unir a S_3 não poderiam existir (nos referimos aos blocos 3, 4 e 5 que S_2 recebeu de S_0 enquanto recebia seus próprios blocos 0, 1 e 2). O que acontece neste caso é que estes blocos (3, 4 e 5) são descartados pelos antigos clientes

de S_2 para que S_2 e S_3 realmente andem juntos e nas mesmas condições do ponto de união para frente.

Esta prática de descartar blocos já recebidos pode soar como um desperdício de banda. Acontece que o grupo mais novo, do extinto fluxo S_3, não tem estes blocos e fará requisição dos mesmos de qualquer jeito; portanto se os blocos forem descartados pelos antigos clientes de S_2 sabendo-se que eles os receberão novamente pelo mesmo fluxo que atenderá aos novos membros de S_2 então isto não fará diferença quando olharmos sobre o aspecto de consumo/economia de banda.

A técnica de *patching* interativo (PI) e de *patching* interativo eficiente (PIE) serão descritas a seguir e, por serem objeto de nosso estudo, serão explicadas com maior profundidade e nível de detalhes.

(e) *Patching Interativo* **(PI)**

A técnica de *patching* interativo, doravante referida simplesmente por PI, é muito semelhante à técnica de *patching* com pequenas modificações que a adequam ao ambiente interativo. Estas modificações se fizeram necessárias pois a técnica de *patching* puro foi desenvolvida para o acesso seqüencial e assume que os clientes assistem o filme do início ao fim sem interações. Quando falamos "do início ao fim" estamos sendo literais, isto é, o cliente assiste o filme do seu primeiro ao seu último segundo de conteúdo.

Quando um cliente faz algum tipo de movimento (interage), é procurado um fluxo que melhor o atenda de acordo com o trecho do vídeo que ele quer assistir. Aos clientes que escutam um mesmo fluxo denominamos grupo, sendo assim, quando um cliente se movimenta procuramos um grupo que melhor atenda às suas novas requisições de vídeo.

Dada uma interação de um cliente há várias possibilidades para o atendimento do mesmo. Seja x o bloco de vídeo que o cliente quer assistir após sua interação e W o valor da janela ótima da técnica de *patching*.

Se existe uma janela ativa de *patching* e o bloco x está dentro da janela,

ele é atendido como um novo cliente, exatamente igual à técnica *patching* com a diferença de que o seu *patch* pode não ser à partir do bloco inicial, ou seja, o cliente solicita, via fluxo *unicast*, um bloco $x|0 \leq x \leq W$ onde 0 é o primeiro bloco de vídeo e W o tamanho da janela ótima da técnica de *patching*.

Se não existe uma janela ativa, há três possíveis casos a serem analisados:

Caso 1: Existe um grupo G que neste momento está recebendo o bloco $G(y)$ onde $G(y) < x$, isto é, o cliente foi para um ponto que fica à frente de G a uma distância δ' onde

$$\delta' = x - G(y) \leq \delta_{\text{before}}; \tag{2.1}$$

Caso 2: Existe um grupo G que neste momento está recebendo o bloco $G(y)$ onde $G(y) > x$, isto é, o cliente foi para um ponto que fica atrás de G a uma distância δ' onde

$$\delta' = G(y) - x \leq \delta_{\text{after}}; \tag{2.2}$$

Caso 3: Não existe um grupo G que neste momento esteja recebendo um bloco y tal que a condição (2.1) ou a condição (2.2) seja satisfeita.

No Caso 1, o cliente passa a escutar o fluxo de G apesar de o cliente ter solicitado um bloco posterior ao que o grupo está recebendo. Isto é feito pois o cliente não é tolerante a não receber o que pediu, mas tolera receber o que não pediu (obviamente dentro de um certo limite). Quando o cliente pede o bloco x e é alocado em um grupo G que está recebendo o bloco y conforme o caso 1, é certo que em algum momento à frente ele receberá o bloco x e todos os subseqüentes. Chamaremos de δ_{before} o número máximo de blocos de vídeo anteriores ao solicitado que um cliente tolera receber. Em outras palavras, quanto tempo extra de vídeo um usuário típico tolera assistir anterior ao que ele realmente quer assistir.

O Caso 2 é tratado de forma idêntica à técnica de *patching*, com a diferença de que o cliente não precisa estar no ponto inicial do vídeo. Se o cliente solicitou um bloco x e existe um grupo G que está recebendo um bloco y na frente de x, este cliente é inserido no grupo G, passa a receber os blocos transmitidos para este grupo e faz o *patch* dos blocos que faltam, ou seja, ele fará requisição unicast dos blocos x, $x+1$, $x+2$, ..., $y-1$ ao mesmo tempo que recebe os blocos y, $y+1$, $y+2$, ... oriundos do fluxo que atende o grupo G. Assim como há estudos para o δ_{before} tolerável há também estudos sobre o valor ótimo para δ_{after} de forma a maximizar o compartilhamento da banda.

No Caso 3 não há nehum grupo que sirva para este cliente (ou não existe nenhum outro grupo ativo ou nenhum grupo está perto o suficiente para que este cliente se una a ele). Neste caso um novo grupo *multicast* é criado para este cliente que passa a escutá-lo e a exibir o vídeo imediatamente. Este caso é semelhante ao da técnica de *patching* quando não há nenhum cliente no sistema e um novo cliente chega, exceto pela diferença de que aqui o cliente pode solicitar qualquer parte do vídeo e não necessariamente o bloco inicial.

Há casos em que há mais de um grupo disponível para a entrada de um cliente novo no sistema ou que interagiu. Neste caso o critério de desempate é o seguinte: selecionam-se os grupos que estão no caso 1 descrito anteriormente, isto é, os grupos onde $x > G(y)$, e calcula-se a menor distância temporal entre o cliente e estes grupos:

$$\min_{\forall i | x - G_i(y) \leq \delta_{\text{before}}} [x - G_i(y)] \tag{2.3}$$

tomando-se o mais próximo, resultado do cálculo acima, como o grupo escolhido. Não havendo um grupo em posição anterior à do pedido do cliente, toma-se o grupo mais próximo dentre os que estão em posição posterior à do cliente, região δ_{after}, com o seguinte cálculo,

$$\min_{\forall i | G_i(y) - x \leq \delta_{\text{after}}} [G_i(y) - x] \tag{2.4}$$

A figura 2.7 exemplifica estes 3 casos. O cliente interage e solicita o bloco x. Os grupos G_1, G_2, G_3 e G_4 estão, neste momento, recebendo os blocos $x - 10$, $x - 5$, $x + 2$ e $x + 7$ respectivamente. Para os grupos que se encontram na região do δ_{before} selecionamos o grupo G_2 conforme a Equação 2.3; para os grupos da região δ_{after} selecionamos G_3 conforme a Equação 2.4. Entre G_2 e G_3, G_2 será o escolhido pois está na região δ_{before}. Isto se dá desta forma pois se o cliente começar a escutar o fluxo de G_2 não haverá necessidade de se fazer *patch*, o que evitará a abertura de um novo fluxo de dados, economizando banda.

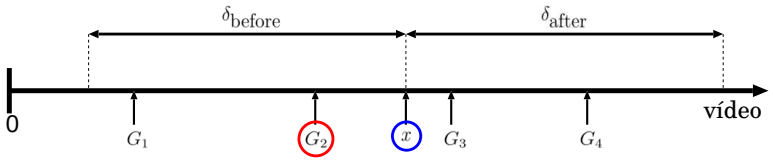


Figura 2.7: Janelas δ_{before} e δ_{after}.

Na figura 2.8 temos o algoritmo da técnica PI. Fica claro no algoritmo que o primeiro quesito a ser testado é se o cliente caiu dentro de alguma janela ativa; caso este quesito não seja atendido então o desempate será feito na ordem "ser o mais próximo dentre os que estão em δ_{before}" e "ser o mais próximo dentre os que estão em δ_{after}". Não sendo atendido nenhum destes quesitos então um novo fluxo *multicast* é aberto (um novo grupo é criado) e o cliente é inserido neste grupo.

(f) *Patching Interativo Eficiente* **(PIE)**

A técnica PIE é semelhante à PI em vários aspectos e características de seu funcionamento. Fundamentalmente, assim como na técnica PI, o único evento que pode provocar uma união de fluxos no sistema é a chegada de uma requisição. Mantendo a mesma simbologia usada até agora explicamos melhor como funciona esta técnica.

```
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 2.8: Algoritmo da técnica PI.

De início ela difere da técnica PI quanto ao primeiro critério usado para escolha do grupo que um cliente que fez uma nova requisição irá pertencer, que é verificar se existe alguma janela ativa. A técnica PIE não usa esta janela ativa e toda requisição nova que chega recai diretamente nos três casos apresentados para a técnica PI (ver seção/item 2e, página 17).

Nos casos 1 e 2 a solução é exatamente a mesma da descrita na seção que fala sobre a técnica PI. No caso 3, quando um cliente novo chega ao sistema, faz uma requisição do bloco x e não existe nenhum grupo no sistema ou nenhum dos grupos existentes está próximo de x o suficiente para que haja o agrupamento então um novo fluxo *multicast* F_n é criado para atender esta requisição formando-se um novo grupo N. Feito isto, é ainda verificado se há no sistema um grupo G cujo fluxo F_g esteja transmitindo um bloco $G(y)$ a uma distância máxima δ_{merge} de x, isto é, $G(y) \leq x - \delta_{\text{merge}}$, e que nenhum de seus membros esteja fazendo *patch*[6]. Se tal grupo existir então o fluxo inicialmente criado para atender a requisição do bloco x, F_n, se tornará um fluxo alvo para o grupo G e todos os seus membros além de escutar o fluxo F_g de G passarão a escutar também o fluxo F_n. Sendo y o bloco que o grupo G estava recebendo no momento em que foi avisado de que deveria escutar também o fluxo F_n, ao atender a este aviso, G receberá simultaneamete os blocos x, $x+1$, $x+2$, ... e os blocos y, $y+1$, $y+2$, ..., $x-1$. No momento em que o grupo G receber de F_g o bloco $x-1$, F_g será extinto pois o bloco x já foi recebido pelo fluxo F_n. Neste momento o grupo G se uniu completamente ao grupo N e, portanto, deixará de existir.

Na figura 2.9 temos um exemplo de como é a dinâmica da técnica PIE quando um novo grupo é criado: no instante t_0 temos a chegada de um cliente para o qual é aberto o fluxo F_g, criando-se assim o grupo G. No instante t_1, quando G recebe o bloco y, temos a chegada de um outro cliente, que requisita o bloco x. Como não há nenhum fluxo em

[6]Esta restrição impede que a árvore de fluxos tenha profundidade maior que dois.

andamento que esteja dentro de uma das janelas δ_{before} e δ_{after}, então não há a possibilidade desta nova requisição ser agrupada em algum grupo existente. Em face disto, um novo fluxo F_n, e conseqüentemente um novo grupo N, é aberto para atender a esta requisição. Criado o fluxo, é verificado no sistema se há algum grupo que esteja recebendo blocos a uma distância de até δ_{merge} dos blocos transmitidos por F_n, situação cujo grupo G se enquadra. Com isto o grupo G é avisado de que deve começar a ouvir, também, os dados transmitidos pelo fluxo F_n. Ao fazer isto o grupo G começa a receber os blocos x, $x+1$, $x+2$, ... enquanto continua ouvindo seus blocos y, $y+1$, $y+2$, ..., $x-1$. No momento em que G recebe de F_g o bloco $x-1$, o fluxo F_g é extinto pois o bloco x já foi recebido via F_n. Com isto o grupo G deixa de existir definitivamente e todos os seus membros são, agora, membros exclusivos de N.

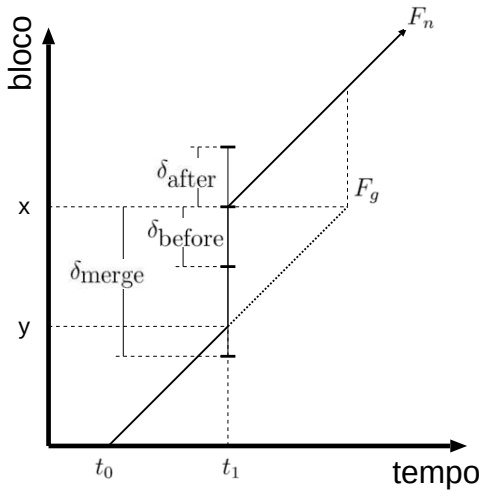


Figura 2.9: *Patching* interativo eficiente.

É importante dizer que no momento em que o grupo G é avisado de que deve escutar também o fluxo de N, significa dizer, também, que G tornou-se um **subgrupo** de N e que assim permanecerá até que não necessite mais de seu fluxo original F_g. Quando acontecer a eliminação do fluxo de G então todos os seus membros se tornarão membros exclusivos de N, F_g será extinto e G conseqüentemente deixará de existir.

2.3 Sistema Multimídia RIO

Definimos por Sistema Multimídia RIO o conjunto formado pelo nó servidor (*dispatcher*), nós de armazenamento (*storage servers*), *patching layer*, cliente administrativo e cliente de vídeo. Definimos, ainda, por servidor RIO o conjunto formado pela dupla *dispatcher* e *storage*.

O nome RIO vem de **R***andomized* **I***nput* **O***utput*, o que reflete bem o comportamento de como os objetos são armazenados e distribuídos no *storage server*.

Nesta seção apresentaremos os elementos que compõem o sistema multimídia RIO e como se dá o relacionamento entre cada um deles.

2.3.1 O Servidor RIO

O primeiro protótipo do servidor RIO foi desenvolvido na *University of California at Los Angeles* (UCLA)[UCLA 2008], foi implementado para SUN E4000 para simulações tridimensionais e exibição de vídeos no formato MPEG. A partir do ano 2000 este protótipo foi codificado e aperfeiçoado com inclusão de novas funcionalidades e novas rotinas pela equipe do LAND. Sua implementação era voltada para transmissão *unicast* até que nos anos de 2003 e 2004 foram desenvolvidas dissertações de mestrado [Netto 2004, Gorza 2003] que tinham como objeto de estudo o compartilhamento de banda através de transmissões IP *multicast*. Foi nessa época que foi incorporado ao RIO um módulo (*Patching Layer*, descrito adiante no texto) inserido em sua arquitetura, que permitiu o compartilhamento dos fluxos de dados enviados pelo *storage server* entre os clientes de vídeo do sistema.

Como dito anteriormente, o servidor RIO é composto de dois elementos, *dispatcher* e *storage*, cujo relacionamento pode ser entendido pela figura 2.10[7]. Repare que no servidor RIO temos apenas um *dispatcher* e um ou mais *storage servers*. Os *storages* podem ou não ficar na mesma máquina que o *dispatcher* além de poderem ficar

[7]Nesta figura ainda não está representada a participação do *patching layer* no sistema RIO.

em máquinas distintas entre si. O *dispatcher* é responsável por receber a conexão do cliente, receber sua requisição e informar ao *storage* que tiver os dados requisitados que os envie diretamente ao cliente. Note que a troca de mensagens é feita via TCP (**T***ransmission* **C***ontrol* **P***rotocol*) e que os fluxos de dados (*stream* de vídeos) são enviados via UDP (**U***ser* **D***atagram* **P***rotocol*). O protocolo TCP é usado pois na troca de mensagens de controle tem que se ter a garantia de que a comunicação será sem perdas de mensagens, o que acarretaria no mal funcionamento do protocolo de comunicação entre os componentes; já o *stream* de vídeo é tolerante a perdas pois a perda de um ou outro pacote não é extremamente prejudicial à exibição do vídeo e, dependendo da perda, nem é perceptível ao usuário final[8].

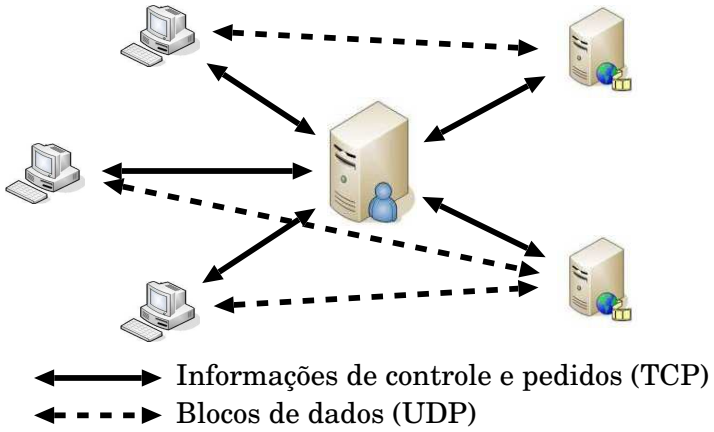


Figura 2.10: Arquitetura básica do sistema RIO *unicast*.

Os tipos de dados armazenados no servidor RIO podem ser diversos: vídeo, texto, áudio, imagem. Esta diversidade é que categoriza o servidor RIO como um sistema de armazenamento multimídia universal [Netto 2004]. Os dados armazenados no servidor podem ser replicados para evitar que um único *storage* fique sobrecarregado e ainda para aumentar a confiabilidade do sistema. Estas réplicas dificultam a administração dos objetos pois toda inserção, alteração ou remoção de dados devem ser feitas em todas as réplicas, mas quando se analisa a relação custo-benefício ela é

[8]Para se ter uma idéia, um bloco típico de vídeo do RIO (cuja exibição dura aproximadamente um segundo) é composto de noventa fragmentos de 128 bytes; sendo assim, cada pacote perdido é um noventa avos de um bloco, o equivalente a aproximadamente 0.011111 segundos.

bem vantajosa pois a escalabilidade do sistema é muito maior com o uso de replicação de dados [Valle 2007, Pacheco 2007, Botelho 2006].

A seguir detalharemos o funcionamento interno do *dispatcher* e do *storage*.

1. **Nó servidor, ou** *Dispatcher*

   Na tarefa de gerenciar as conexões que chegam dos clientes, das requisições destes e da interação com os *storages*, o *dispatcher* implementa vários gerenciadores que, em conjunto, permitem que os dados sejam armazenados, gerenciados, recuperados e transmitidos de forma eficiente.

   Dentre os gerenciadores está a *SessionManager*[9] que é responsável por receber todos os pedidos dos clientes. Ela é o ponto de conexão entre o servidor RIO e seus clientes. Cada cliente que entra no sistema e se conecta ao *dispatcher* é atendido pela *SessionManager* e tem uma *thread*[10] criada especificamente para tratar os seus pedidos.

   Os fluxos de dados de vídeo do servidor são, por sua vez, gerenciados pela *StreamManager*[11]. Esta é responsável tanto pelos fluxos já abertos quanto pela abertura de novos outros. Uma tarefa importante deste gerenciador é o controle de admissão [Cardozo 2002], no qual um novo cliente só é aceito dentro de certas condições para que se mantenha, sempre que possível, uma boa qualidade no serviço. Resumidamente, este controle de admissão considera a taxa de transferência já alocada T_{A} e a taxa solicitada T_{S} pelo novo cliente. Sabendo que temos dois tipos de clientes, o cliente de vídeos e o de administração, temos, para cada um deles um tipo de transmissão de vídeo: tempo real e sem restrição de tempo respectivamente. O controle de admissão impede que os clientes cujo serviço tem restrição de tempo (clientes de vídeo) monopolizem a banda impedindo que clientes sem restrição de tempo,

[9] Gerenciador de sessão.

[10] Ou linha de execução em português, é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas simultaneamente[MediaWiki 2008].

[11] Gerenciador de fluxos.

que consomem uma taxa T_{NRT}, sejam atendidos em momentos de pico de utilização do sistema. Sendo T_{T} a taxa máxima de transferência do sistema, aberturas de novos fluxos só serão permitidas pela *StreamManager* se

$$(T_{\mathrm{A}} + T_{\mathrm{S}}) \leq (T_{\mathrm{T}} - T_{\mathrm{NRT}}).$$

Um serviço semelhante ao de controle de admissão prestado pelo *StreamManager* é o policiamento de pedidos. Seu objetivo é não permitir que um cliente faça muitas requisições em um curto intervalo de tempo pois isto poderia prejudicar a entrega dos dados dos demais clientes, principalmente os clientes com restrição de tempo, que teriam seus blocos atrasados pois a fila de pedidos a serem atendidos, a ser explicada adiante, estaria constantemente cheia até que alguns clientes saíssem do sistema.

Cada objeto armazenado no servidor RIO tem seus metadados[12]. A responsabilidade sobre estes metadados é do *ObjectManager*[13]. Este gerenciador, além desta tarefa, é quem administra as operações sobre os objetos tais como criação, exclusão, abertura, fechamento, escrita e leitura. A cada alteração, criação ou remoção de um novo objeto, o *ObjectManager* é responsável por manter consistentes todas as réplicas e os metadados do objeto manipulado. É ele também quem distribui os blocos nos discos dos nós de armazenamento, definindo onde cada bloco vai ficar e, se estiver sendo usada redundância de dados, quais discos serão contemplados com cada bloco de dados, sempre restringindo cada disco a no máximo uma cópia de cada bloco.

Um dos gerenciadores do *dispatcher* é o *router*[14]. É ele quem decide qual nó de armazenamento atenderá qual solicitação. Isto é feito através da implementação de filas que separam as requisições de tempo real das sem restrição de tempo.

[12]Metadados, ou Metainformação, são dados capazes de descrever outros dados, ou seja, dizer do que se tratam, dar um significado real e plausível a um arquivo de dados [MediaWiki 2008].

[13]Gerenciador de objetos.

[14]Roteador

Na figura 2.11 entendemos melhor como funcionam estas filas. Quando chega ao sistema um pedido de leitura de um determinado bloco, o *router* procura o disco que tem tal bloco e, dentre os que têm, escolhe o que tiver a menor fila, direcionando o pedido do cliente para ele. No caso de uma requisição de escrita o *router* envia a requisição para todos os discos que têm aquele bloco pois é necessário que todas as réplicas sejam atualizadas para que os dados do servidor RIO mantenham-se consistentes.

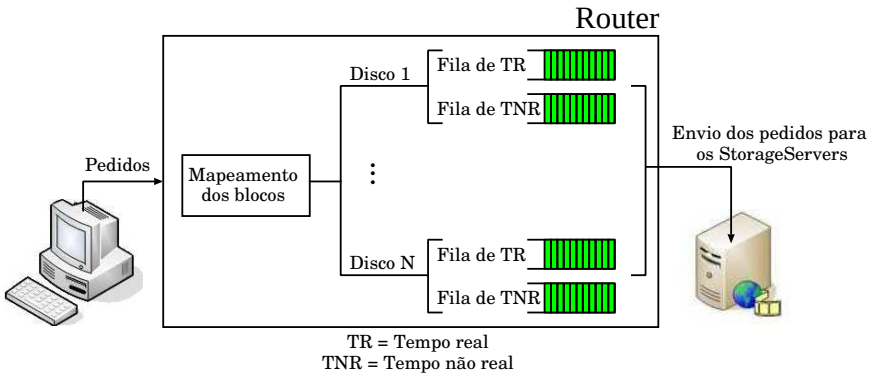


Figura 2.11: Gerenciamento de pedidos no *router*.

2. **Nó de armazenamento, ou** *Storage*

   No *storage* é que ficam armazenados os objetos multimídia. O nó de armazenamento foi implementado de forma que pode ser múltiplo, isto é, ter mais de um nó para um mesmo servidor RIO. Estudos recentes [Botelho 2006, Valle 2007] mostraram que o sistema aumenta sua capacidade de atendimento conforme se aumenta o número de *storages*. A seguir é descrito de forma mais detalhada o funcionamento do *storage*, seus componentes e, o mais interessante, sua política de armazenamento de dados. Entenderemos agora de forma mais consistente, com este último tópico, o motivo deste sistema ter o nome **RIO**.

   (a) **Os componentes do** *storage*.

   Um nó de armazenamento é composto por quatro componentes: *StorageDevice, StorageManager, RouterInterface e ClientInterface*. O *StorageDevice* é tão somente o dispositivo de armazenamento e é responsável pe-

las operações de entrada e saída de dados através de chamadas a funções do sistema operacional. Cada *StorageDevice* é controlado exclusivamente por um *StorageManager*. É o *StorageManager* que faz o escalonamento dos pedidos dos dados armazenados no dispositivo de armazenamento e isto é feito através de uma fila onde os pedidos são colocados até serem atendidos. Para cada pedido é associado um *buffer* para o armazenamento do bloco solicitado. Quando a operação do cliente é de leitura, o *StorageManager* solicita ao *ClientInterface* o envio do bloco ao cliente. No caso da operação de escrita, o *StorageManager* se responsabiliza pelo armazenamento do bloco.

As requisições, envio e recebimento de informações, feitas entre o *storage* e o *dispatcher* são gerenciadas pelo *RouterInterface*. Por se tratar de troca de mensagens, o protocolo usado é o TCP. Ao receber um pedido de bloco de dados de um cliente, que foi repassado a este pelo *dispatcher*, o *RouterInterface* o repassa ao *StorageManager* que procede conforme descrito anteriormente.

O último e mais interessante módulo do *storage* é a *ClientInterface*, que é quem interage com o cliente e é responsável pelo envio e recebimento de dados a serem transmitidos e armazenados respectivamente. Com o objetivo de compatibilizar o tamanho de um bloco de vídeo com o MTU[15](Maximum Trasmit Unit ou Unidade Máxima de Transmissão) da rede, cada bloco de vídeo é dividido em noventa fragmentos que são enviados em rajada. Isto tem que ser feito na aplicação pois se forem enviados blocos maiores que o valor da MTU os mesmos serão fragmentados, isto é, divididos em pedaços menores, e enviados. Com isto, o cliente só poderá exibir o bloco quando todos os fragmentos tiverem sido recebidos, além de provocar um atraso no envio dos dados devido ao custoso processo de fragmentação.

[15]MTU se refere ao tamanho do maior datagrama que uma camada de um protocolo de comunicação pode transmitir.

Uma vantagem do envio de um bloco na forma de rajada de 90 fragmentos está em que a perda de um fragmento não implica na perda de um bloco inteiro, pois a probabilidade de alguns fragmentos chegarem é muito maior do que a de todos os noventa fragmentos se perderem. Entretanto, é preciso ter cuidado na transmissão pois esta rajada de noventa fragmentos pode esgotar o *buffer* da placa de rede, o *buffer* do *switch*, *buffer* do roteador ou de qualquer dispositivo que necessite receber os dados e retransmití-los no caminho entre o cliente e o servidor. Uma solução adotada para minimizar o problema da perda devido à rajada de noventa fragmentos é a de implementar um controle de fluxo por cliente e um segundo controle do fluxo total.

O relacionamento entre estes componentes é exemplificado na figura 2.12, extraída de [Netto 2004].

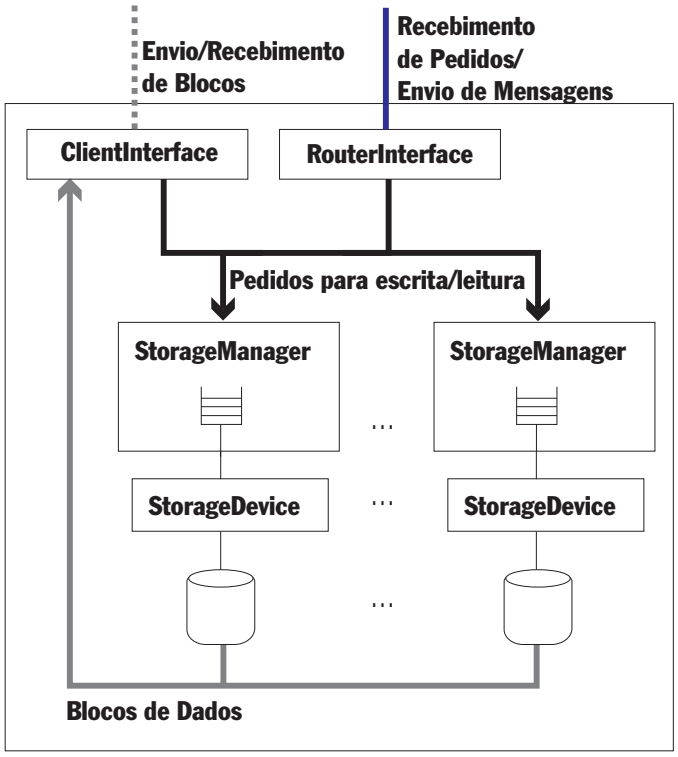


Figura 2.12: Arquitetura interna do *storage*.

(b) **Política de armazenamento de dados**

A política de armazenamento dos objetos no servidor é de vital importância para o desempenho do sistema e uma boa qualidade no serviço. Se a política a ser adotada não for cuidadosamente escolhida, o servidor pode não ser capaz de atender um número alto de clientes e, pior, pode não ser escalável.

A política mais simples seria a de alocar os objetos seqüencialmente, mas isto seria um problema grande pois os discos que armazenassem os objetos mais populares sofreriam uma sobrecarga enquanto outros com objetos menos requisitados estariam ociosos.

Para evitar esta sobrecarga em um disco, novas técnicas foram desenvolvidas cujo foco é distribuir os objetos pelos discos que compõem o sistema. Abordaremos aqui duas dessas técnicas: *Striping* e *Random data Allocation*[Netto 2004].

A técnica *striping* é, de longe, a mais utilizada. Seu uso é bem comum em máquinas que desempenham papel de servidor de banco de dados ou servidor de backups que manipulam grande quantidade de dados de forma intensa. Nestas máquinas vários discos são instalados mas todos são combinados em apenas uma única unidade lógica de armazenamento. Esta tecnologia se chama RAID (**R***edundant* **A***rray of* **I***ndependent* **D***isks*)[16] e a técnica *striping* é essencial para ela. As vantagens deste tipo de armazenamento são diversas, dentre as mais importantes estão:

- Ganho de desempenho no acesso;
- Redundância em caso de falha em um dos discos;
- Uso múltiplo de várias unidades de discos;
- Facilidade em recuperação de conteúdo "perdido".

No *striping* os dados são subdivididos em segmentos consecutivos (stripes, ou faixas) os quais são escritos de forma seqüencial através de cada um dos discos combinados. Cada segmento tem um tamanho definido

[16]O objetivo principal da tecnologia RAID é ganhar segurança e desempenho

em blocos. Esta técnica pode ser ajustada para que o tamanho dos blocos possa ser ajustado à necessidade da aplicação que o utiliza. Nestes casos, quando aplicada, esta técnica de *array* de discos oferece um desempenho muito melhor se comparada ao uso de discos individuais. A diferença da aplicação de *striping* no servidor RIO para a do RAID é que no RIO os discos podem, opcionalmente, estar distribuídos em máquinas geograficamente distantes.

Na figura 2.13 [Netto 2004] é exemplificado como funciona o armazenamento físico neste *array* de discos.

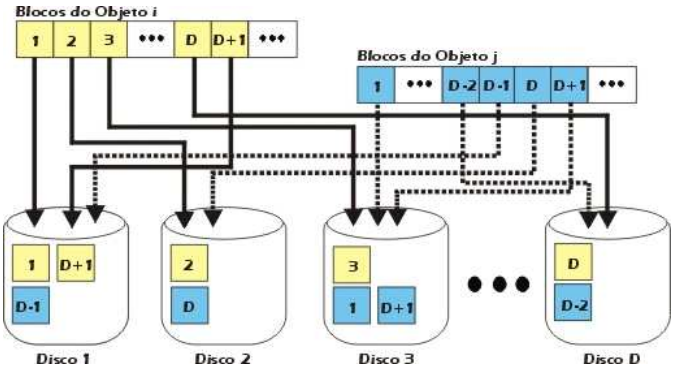


Figura 2.13: Técnica de *striping*.

A técnica *Random Data Allocation*, que é a implementada no servidor RIO, também armazena um mesmo objeto distribuidamente sobre os diversos discos do sistema. A sua política de armazenamento é aleatória. Para cada bloco de dados a ser armazenado no *storage server* é escolhida uma posição aleatória dentro de um disco escolhido também aleatoriamente.

No caso do servidor RIO há algumas regras importantes a serem consideradas: quando se faz uso de réplicas existe uma restrição na escolha aleatória dos discos: um bloco não pode ser armazenado mais de uma vez no mesmo disco. Com isto surge, implicitamente, uma restrição ao número máximo de réplicas que um objeto pode ter no servidor: restringe-se ao número de discos; logo, se o servidor RIO trabalha com n discos então

pode trabalhar com até n réplicas de cada objeto.

A figura 2.14 [Netto 2004] ilustra como um objeto é armazenado no servidor RIO e/ou em sistemas que implementam a técnica *Random data Allocation*. Note que o objeto i, cujo tamanho é de cinco segmentos, tem seu primeiro segmento armazenado no disco 1, o segundo no disco 2, e assim sucessivamente até o D-ésimo segmento, armazenado no disco D. Como temos, no exemplo, apenas D discos, o segmento $D+1$ é armazenado no disco 1 novamente, o $D+2$ no disco 2 e assim sucessivamente até todo o objeto estar armazenado nos D discos do sistema. Analogamente o objeto j, também com tamanho de cinco segmentos, tem seu primeiro segmento armazenado no disco 3 e os demais segmentos distribuídos com o mesmo critério adotado para o objeto i.

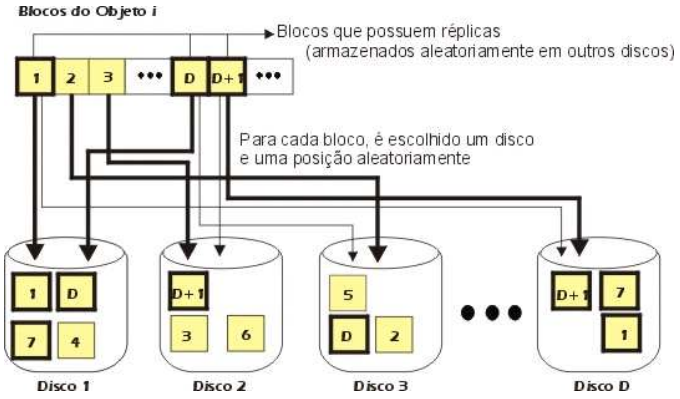


Figura 2.14: Técnica de *Random Data Allocation*.

2.3.2 *Patching layer* (PL)

Na seção 2.3.1 comentamos que o servidor RIO foi, inicialmente, implementado para trabalhar com fluxos *unicast* e que algumas poucas alterações foram feitas em sua estrutura para que ele passasse a trabalhar também com fluxos *multicast*. Para que o código do servidor RIO fosse minimamente alterado, as atividades gerenciais de compartilhamento dos fluxos foram, em sua maior parte, implementadas em um módulo separado do servidor, o *patching layer*. Este módulo é um aplicativo que roda

independente do servidor e que intermedia a comunicação entre este e os clientes. Na figura 2.15 tem-se uma idéia desta intermediação (vale ressaltar que mesmo com o PL fazendo parte do sistema, um cliente pode, ainda, conectar-se diretamente ao *dispatcher* como mostrado na figura 2.10).

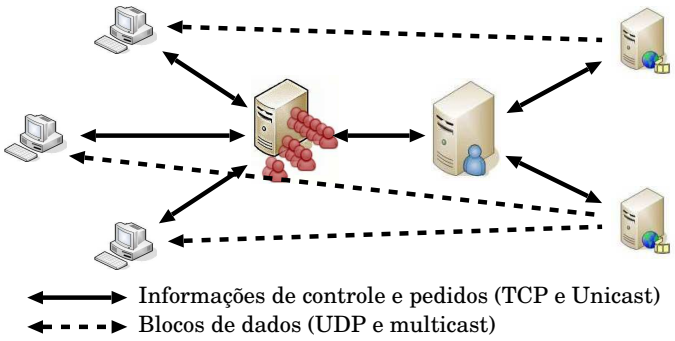


Figura 2.15: Arquitetura básica do sistema RIO *multicast*.

Assim como o *dispatcher* e o *storage*, o PL também é formado por diversos componentes que agem conjuntamente na tarefa de compartilhar fluxos entre clientes do sistema. Dentre esses componentes descrevemos, sucintamente, os mais importantes:

SessionManager: responsável pelo recebimento dos pedidos de blocos feitos pelos clientes e do carregamento dos arquivos de configurações que definem parâmetros importantes a serem utilizados pelo PL como tamanho das janelas δ_{after}, δ_{before} e δ_{merge}, endereço de rede do *dispatcher*, dentre outros.

StreamManager: este componente não tem nenhuma função importante no PL. No processo de estruturação do PL procurou-se manter a mesma arquitetura que a do servidor RIO. Sendo assim, este componente é equivalente à *StreamManager* do servidor e seu papel aqui é tão somente repassar a ela todas as mensagens que vierem do cliente e devam ser tratadas por ela.

ObjectManager: responsável pelo tratamento dos pedidos que a *SessionManager* recebe dos clientes, pela abertura, fechamento e requisições de blocos dos objetos armazenados no servidor RIO.

CommMulticast: este componente é a interface de comunicação entre o cliente e o PL. Por ele são trocadas mensagens que orientam o cliente sobre ações a serem tomadas em resposta aos seus movimentos, como unir-se ou abandonar um grupo, dentre outras orientações.

RIOInterface: assim como a *StreamManager*, este componente serve apenas para repasse de mensagens do cliente que se destinam à *RIOInterface* do servidor.

O algoritmo apresentado na figura 2.8 mostra como funciona a política de agrupamento de fluxos no PL. Em nível de implementação, isto se dá por uma troca de mensagens entre os clientes e o PL, onde os clientes informam ao PL de suas interações e o PL responde informando ao cliente qual ação deve ser executada. Vamos apresentar agora o algoritmo, em alto nível, deste processo nas diversas situações possíveis:

Novo cliente entra no sistema: Cliente entra no sistema e faz uma troca de mensagens iniciais onde recupera informações sobre o objeto de vídeo que quer ter acesso e recebe um identificador único, fornecido pelo PL, que é um número que o PL utiliza para distingüir este cliente dos demais conectados a si. O cliente responde ao PL informando seu identificador e tem como resposta o endereço do fluxo *multicast* que deve passar a escutar. Esta resposta informa, ainda, se o cliente é o responsável pelas requisições deste fluxo *multicast* ou se ele deve apenas escutá-lo, o que significa que um outro cliente está responsável pelas requisições do fluxo. A primeira situação ocorre quando não há um fluxo disponível para o cliente. Neste caso um novo fluxo *multicast* é criado e este cliente é eleito seu líder, isto é, o responsável por fazer as requisições do mesmo. A segunda situação acontece quando já existe um fluxo *multicast* em andamento que atenda às necessidades deste cliente, isto é, o cliente está a uma distäncia máxima de δ_{before} ou δ_{after} deste fluxo.

Um cliente não líder interage: Dizer que um cliente não é líder significa dizer que ele não é responsável pelas requisições dos blocos de um fluxo *multicast* em

andamento. Como estamos falando de um ambiente *multicast*, é obrigatório que este cliente que interagiu esteja em algum grupo com pelo menos dois membros: ele e seu líder, isto é, aquele que requisita os blocos para o grupo ao qual ele faz parte. Uma vez que este cliente não é o líder do grupo, sua saída não trará impacto algum para o grupo e se dá de forma bem simples: o cliente envia uma mensagem ao PL informando qual a sua interação e para qual bloco de vídeo, se for o caso. O PL, ao receber a mensagem, verifica se há um outro fluxo em andamento que atenda as necessidades deste cliente e o informa que deve parar de escutar o fluxo *multicast* que estava escutando e que deve passar a escutar o fluxo *multicast* deste grupo. No caso de não haver um fluxo que atenda tais necessidades, o PL cria um novo fluxo *multicast* e informa ao cliente que ele deve passar a escutá-lo e que, ainda, é o líder do mesmo, sendo, assim, responsável por fazer as requisições dos blocos de vídeo. O processo acima só acontece caso a interação do cliente seja um salto para uma região do vídeo ainda não recebida anteriormente. Caso a interação seja para um trecho do vídeo já recebido anteriormente, o que significa que o cliente já tem tais blocos em disco, ou a interação é de pausa ou parada, o PL aloca o cliente no grupo de clientes inativos e o informa que ele deve parar de escutar o fluxo *multicast* que estava escutando anteriormente.

Um líder de grupo interage: A troca de mensagens entre este líder e o PL é idêntica à troca entre o PL e um cliente não líder. A diferença nesta situação reside no fato de que um novo líder deve ser eleito para que as requisições dos blocos de vídeo do grupo continuem a ser feitas. Para isto o PL elege como líder o cliente que entrou imediatamente após o antigo líder e envia uma mensagem a ele informando que a partir daquele momento ele é o responsável pelas requisições do grupo, ou seja, que ele foi eleito o novo líder do grupo. Nesta situação dizemos que este é um líder forçado, pois a liderança foi imposta pelo PL e não foi fruto de uma interação deste cliente, mas de outro. No caso de o grupo não ter nenhum outro membro que não o antigo líder então o fluxo *multicast* associado a ele é extinto imediatamente.

Um novo fluxo *multicast* é aberto: Seja N o grupo para o qual este novo fluxo foi aberto. Seja qual for a razão da abertura deste novo fluxo, o PL verifica se existe um grupo em andamento cujo fluxo já aberto possa ser fundido a este novo fluxo, conforme descrito na seção 2f. Encontrado este grupo, digamos G, o PL envia uma mensagem a cada um de seus membros informando que os mesmos devem passar a escutar também este novo fluxo. Uma mensagem especial é enviada ao líder deste grupo informando-o de que de líder de G ele passou a ser sublíder de N e que deve continuar a fazer requisições até que seu fluxo se una ao novo fluxo criado. Quando esta união acontece, este sublíder envia uma mensagem ao PL informando que alcançou o fluxo principal (o novo fluxo); o PL, como resposta a esta mensagem, envia uma mensagem a cada cliente que escuta o fluxo deste sublíder, antigos membros de G, que eles não precisam mais escutar o fluxo antigo pois o mesmo está extinto, mas permanecerão escutando o fluxo de N.

Uma descrição mais detalhada destas trocas de mensagens e do corpo destas será apresentada na seção 2.4, onde falaremos das contribuições deste trabalho para o sistema RIO.

2.3.3 Cliente de administração - riosh

O cliente administrativo possibilita que objetos sejam inseridos no servidor RIO enviando-se os dados ao *dispatcher* que se responsabiliza por sua alocação nos *storages*. Com este cliente é possível a criação de objetos e pastas, renomeação, remoção e remanejamento dos mesmos de forma transparente, independente do número de *storages* e de o servidor usar ou não réplicas. Com o riosh é possível até mesmo manter um sincronismo entre os servidores, funcionalidade implementada recentemente. Há duas possíveis interfaces de uso: modo texto e modo gráfico. Na figura 2.16 apresentamos a tela deste cliente. Note que é muito semelhante aos gerenciadores de arquivos mais comuns.

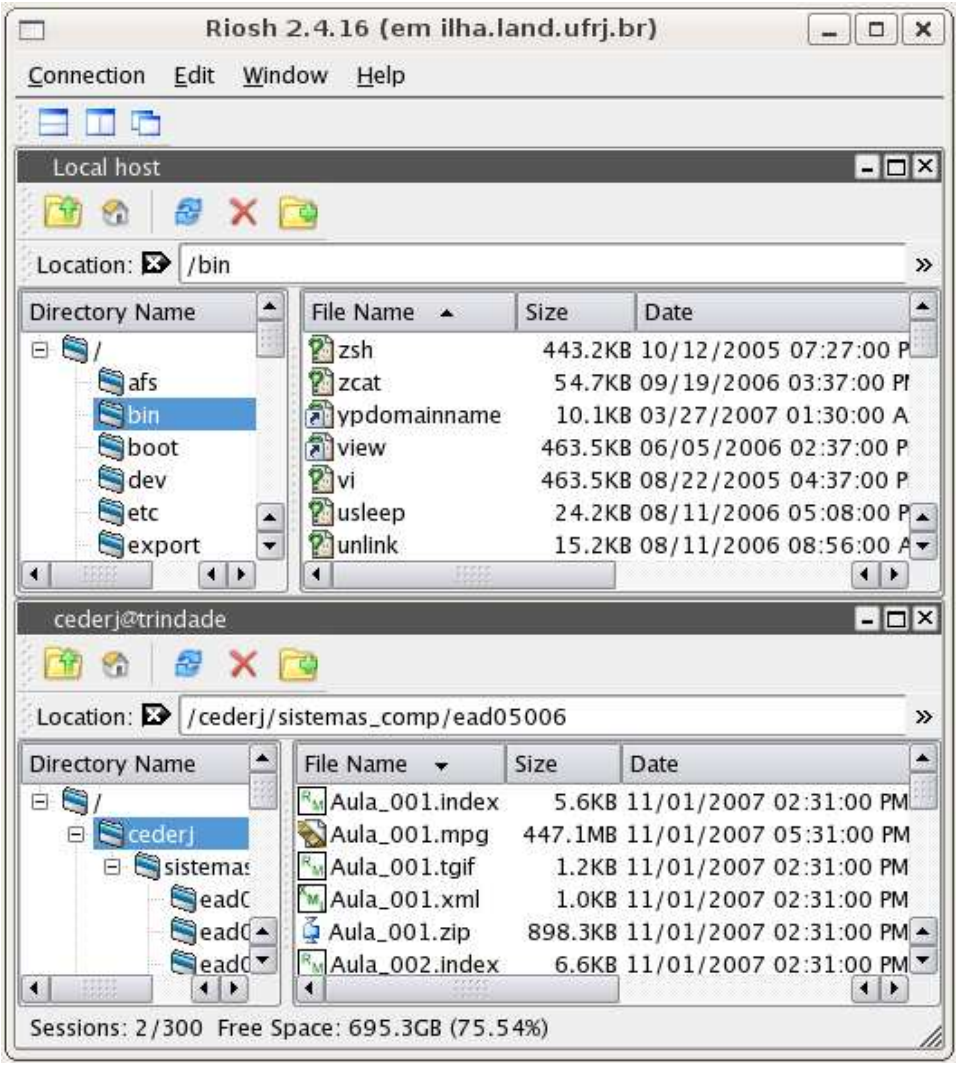


Figura 2.16: Cliente administrativo do servidor RIO.

2.3.4 Cliente de vídeos - riommclient

O cliente de vídeos é responsável por recuperar o vídeo e as transparências do servidor, exibir o vídeo e sincronizá-lo com as transparências. Isto é feito através de uma interface que existe entre o cliente e o servidor RIO.

As transparências podem ser geradas em TGIF (*Tangram Graphic Interface Facility*) [Cheng 2008] ou em HTML (*HyperText Markup Language*)[17], sendo que em TGIF tem se a vantagem de se poder fazer animações que são executadas em mo-

[17]Linguagem de Marcação de Hipertexto.

mentos específicos e sincronizadas com o conteúdo da aula e a fala do professor.

Para a exibição do vídeo é usado o *mplayer* [Gereoffy 2008], para quem os dados de vídeo são enviados através de um *pipe*.

O cliente implementa um *buffer*, onde armazena os blocos a serem exibidos em um curto período de tempo, chamado de *PlayOutBuffer*, e uma *cache*, chamada *BufferStream*, onde armazena, em disco, os blocos recebidos, seja via *multicast*(que serão exibidos em um momento mais à frente) ou *unicast*. Caso o dado recebido tenha que ser gravado tanto na *cache* quanto no *PlayOutBuffer*, dá-se prioridade a este último na ordem de gravação. É importante deixar claro que um bloco só é gravado nestes locais após ser recebido por completo, isto é, cada um dos noventa fragmentos. Há ainda uma ocasião especial em que os fragmentos são gravados no *PlayOutBuffer* independente de se ter os noventa ou apenas parte deles, que é quando o cliente precisa exibir aquele bloco de vídeo imediatamente e ainda não recebeu todos os fragmentos. Neste caso é chamado um método de nome **FreeBlock** que libera os fragmentos que já estiverem disponíveis e os grava no *PlayOutBuffer* imediatamente. Um bloco liberado com a *FreeBlock* não é armazenado na *cache* pois ele está com perdas e na *cache*, por convenção, só se gravam blocos íntegros e completos.

A utilidade e importância do *PlayOutBuffer* é diminuir o *jitter* introduzido pela rede. O cliente, antes de iniciar a exibição do vídeo, espera que este *buffer* seja totalmente preenchido com os primeiros blocos de vídeos a serem exibidos. Para redes locais este *buffer* não precisa ter tamanho superior a duas unidades pois este valor já se mostrou mais do que suficiente para garantir a variação no tempo de atraso de envio dos pacotes. Os blocos de vídeo não são gravados na *BufferStream* de forma seqüencial, mas na exata ordem em que são recebidos. Um índice é criado para permitir que o cliente recupere da *cache* um bloco já recebido do servidor em outro momento em que precise re-exibí-lo.

A interface gráfica do riommclient assemelha-se a um painel de um aparelho

multimídia como os de um *DVD-player* ou de um aparelho de vídeo-cassete. Na figura 2.17 é possível ver esta interface e os comandos que ela disponibiliza ao usuário, a saber, *play, pause, stop, rewind, forward, help* e *quit*. Além destes, é possível ao usuário arrastar a barra de progresso de exibição que fica na parte superior do painel bem como clicar em um dos tópicos apresentados no índice. Isto fará com que a exibição do vídeo pare onde estiver e dê um salto para a posição da barra ou o tópico escolhido no índice.

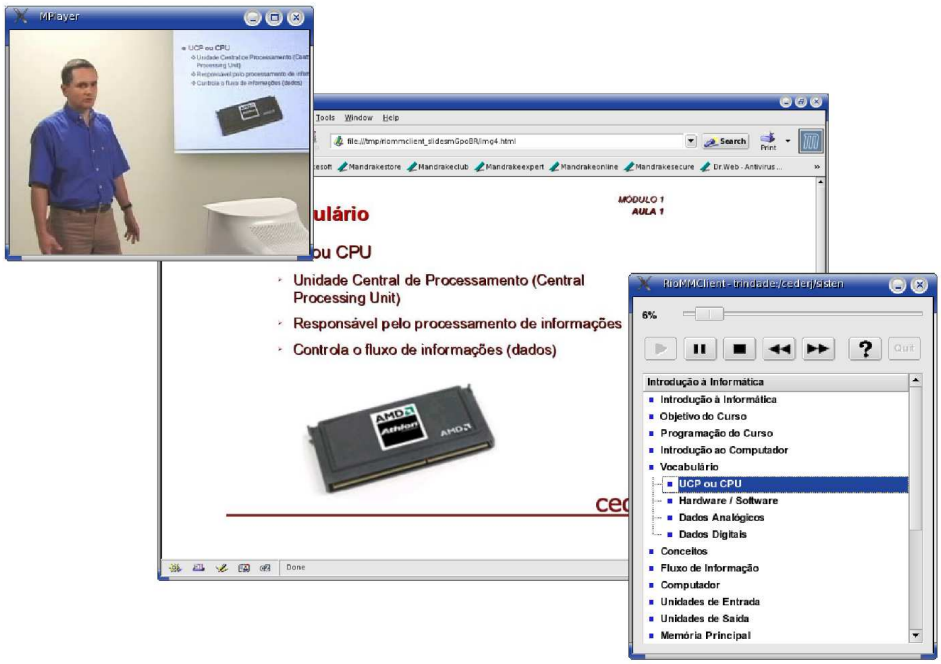


Figura 2.17: Cliente de vídeo e seus componentes.

A integração e inter-relacionamento dos componentes apresentados pode ser melhor entendida analisando-se a figura 2.18 [Netto 2004].

Integração do riommclient ao *browser* - riommbrowser

Tendo em vista a utilização do sistema RIO em projetos de ensino a distância, foi criado um *plugin* para o *Mozilla-Firefox* [Mozilla 2008] e para o *Netscape* [Netscape 2008] que permite que os alunos do curso usem este serviço diretamente pelo *browser* de forma a ser o mais simples e transparente o possível aos usuários. Este *plugin* faz

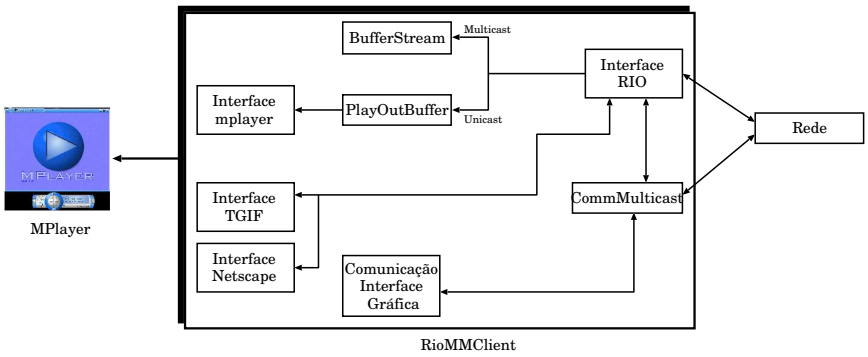


Figura 2.18: Arquitetura do cliente de vídeo riommclient.

com que o riommclient seja exibido dentro do *browser* e o mesmo acontece com os slides quer estejam feitos para os *browser's Netscape* ou *Mozilla-Firefox* ou feitos em TGIF. A figura 2.19 mostra um cliente quando inserido no *browser*.

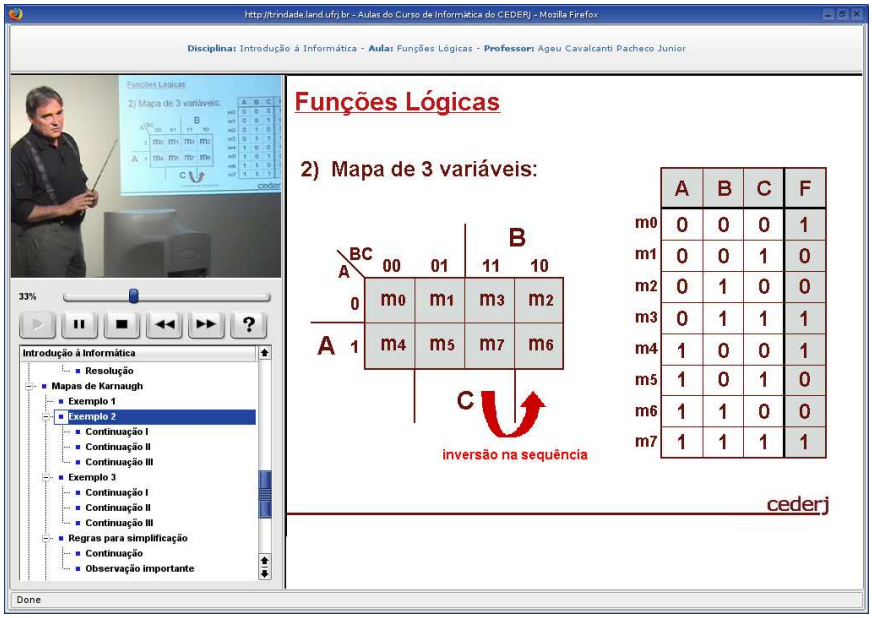


Figura 2.19: Cliente de vídeo acoplado ao *browser*.

Diferenças e semelhanças entre o PL e o cliente de vídeos

Ao olharmos para as figuras 2.15, 2.21, 2.22 e 2.23 temos, sob a ótica do cliente, que o PL é um servidor de vídeos que agrupa clientes para que dois ou mais clientes compartilhem um mesmo fluxo de dados. Já sob a ótica do servidor RIO ele pode

ser visto como um cliente que faz muitas requisições de diversos pontos do vídeo.No entanto, sob o ponto de vista do servidor o PL é bem mais que um cliente pois um cliente abre apenas uma sessão e um fluxo de dados por vez. O PL, por sua vez, abre tantas sessões quantos forem os clientes. O que acontece, de fato, é que quando o PL está intermediando esta comunicação, o servidor sabe que tem muitos clientes conectados, mas como muitos deles estão ouvindo fluxos *multicast* passivamente, o servidor não recebe nenhuma requisição destes clientes e pra ele é como se tais clientes estivessem em *pause* ou se tivessem sido finalizados. Em outras palavras, para o servidor é como se o sistema não estivesse sendo bastante utilizado ou como se estivesse sendo subutilizado, quando na verdade ele está sendo otimizado.

Uma outra possibilidade seria a abertura de uma única sessão e um único canal de fluxo para enviar os pedidos de todos os clientes. Isto não é feito pois o servidor tem um controle de atendimento de pedidos, descrito na seção 1, que impede que um cliente sature o sistema fazendo muitas requisições em um curto período de tempo. Se o PL tivesse apenas um a sessão com o servidor, fatalmente muitos de seus pedidos seriam descartados pois esta sessão concentraria os pedidos de todos os clientes e teríamos rajadas de pedidos com taxa muito alta.

2.4 Contribuições deste trabalho no sistema RIO

Nesta seção apresentamos as contribuições deste trabalho para o sistema RIO. Serão detalhados aspectos de implementação, identificados problemas encontrados na versão anterior e as soluções aplicadas. Para facilitar o entendimento separamos esta seção em subseções, onde abordaremos, separadamente, contribuições no módulo do cliente e no módulo do *Patching Layer*.

2.4.1 Contribuições para o *Patching Layer*(PL)

Nesta seção abordamos a implementação da administração dos grupos *multicast* pelo PL e a troca de mensagens entre este e o cliente de vídeo.

A implementação dos grupos se dá através de listas duplamente encadeadas que se extendem na vertical e na horizontal. Cada linha representa um grupo distinto do sistema. Na figura 2.20 temos o exemplo de como esta estrutura se apresenta em uma situação onde temos três grupos; o primeiro com apenas um membro, o segundo com três membros e o terceiro com dois membros. Os membros mais à esquerda de cada linha desta estrutura são os líderes de seus grupos e responsáveis pelas requisições *multicast* dos mesmos. Note que apenas os líderes têm ponteiros para os grupos vizinhos. Os membros de cada grupo sempre são inseridos ao final do seu respectivo grupo. Se no exemplo da figura 2.20 tivéssemos que acrescentar um novo membro ao grupo do cliente 2 ele seria inserido à direita do cliente 4 independente do bloco de vídeo que esteja exibindo ou requisitando ou de ter sido alocado neste grupo por interagir para um bloco dentro dos seus intervalos δ_{after} ou δ_{before}.

As mensagens trocadas entre o PL e os clientes, através do componente *Comm-Multicast* acima descrito, são quatro: MSGCODE_PID, MSGCODE_PID, MSGCODE_MOVE e MSGCODE_MODE. Apesar de parecerem poucas, cada uma delas pode trazer uma variedade de informações em diferentes situações.

Na versão anterior do PL, onde estava implementada apenas atécnica PI, a troca de mensagens entre este e o cliente estava firmemente baseada no fato de que um cliente só poderia escutar um único fluxo *multicast* e que o cliente só tinha 3 possíveis papéis no sistema: passivo, ativo ou inativo. Um cliente passivo, chamado simplesmente "membro", é aquele que escuta um fluxo *multicast* mas não é responsável por fazer as requisições; o ativo é o chamado "líder", e é quem está responsável por fazer as requisições de blocos de um grupo *multicast*. Finalizando, um cliente é inativo quando não precisa ouvir nenhum fluxo, isto é, ou ele está em pausa, ou está com a

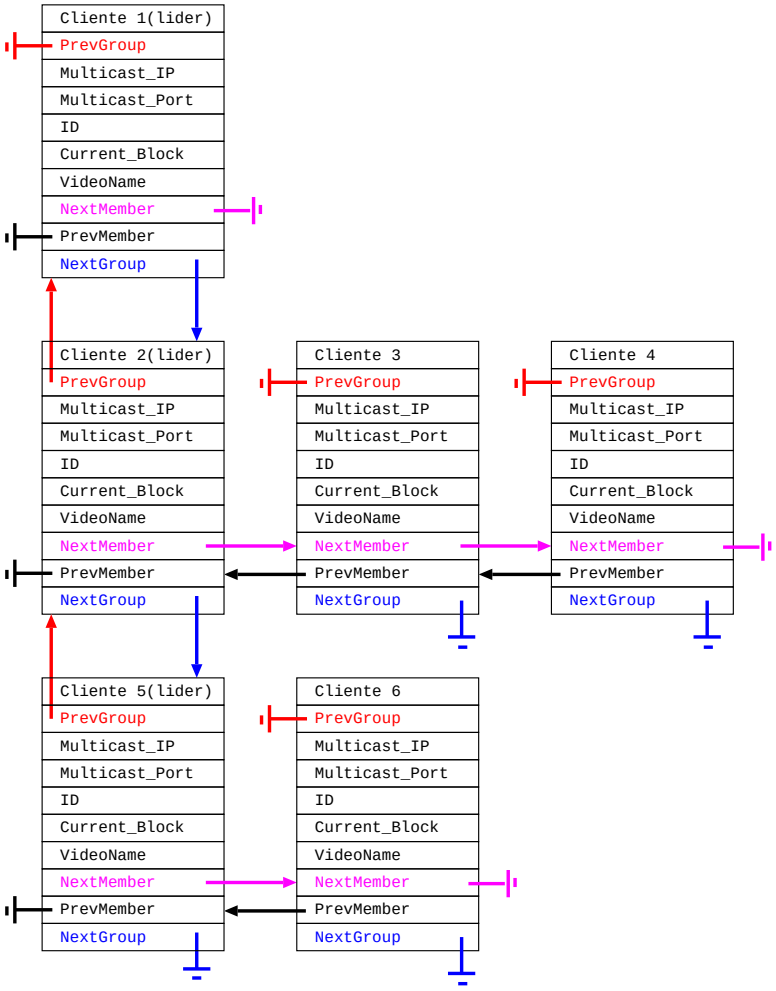


Figura 2.20: Estrutura dos grupos administrados pelo PL.

execução interrompida[18] ou está obtendo os dados diretamente do disco. O cliente inativo também é chamado de "membro", mas é membro do grupo de inativos.

Ao implementarmos a técnica PIE no PL, algumas novas situações surgiram. Neste novo cenário os clientes podem escutar até dois fluxos *multicast* simultaneamente, o que gera situações novas, a saber: os possíveis papéis de um cliente agora são passivo, ativo ou inativo, como os anteriores, e duplamente-passivo e passivo-ativo. Um cliente duplamente-passivo, chamado simplesmente de "submembro", é aquele que escuta passivamente dois fluxos *multicast* e não é responsável pelas re-

[18] A diferença entre o estado de pausa e o estado de interrupção é que o primeiro interagiu com o botão *pause* e o segundo com o botão *stop*.

quisições de nenhum deles; já o cliente passivo-ativo escuta passivamente um fluxo *multicast* e faz requisições para um outro fluxo *multicast*. Neste caso o cliente é chamado "sublíder", recebe passivamente o fluxo cujas requisições é feita pelo seu "líder" e faz as requisições para um segundo fluxo. O fluxo cujas requisições são feitas pelo líder do grupo é escutado por todos os integrantes do grupo, isto é, pelo próprio líder, pelos membros, pelo sublíder e pelos submembros; já o fluxo cujas requisições são feitas pelo sublíder é escutado apenas pelo próprio sublíder e pelos submembros. Mais adiante este relacionamento ficará mais claro, conforme apresentarmos a estrutura utilizada pelo PL na administração destes clientes nos diversos grupos e subgrupos.

A adição destes novos papéis e da possibilidade de um cliente escutar mais de um fluxo *multicast* nos levou a modificar o antigo protocolo de troca de mensagens, adicionando mais informações às mesmas. Estas novas informações são de "ação a ser tomada pelo cliente", "novo papel do cliente no sistema", "tamanho de *patching* do cliente", "*feedback* do cliente ao PL sobre terminar seu fluxo de *patch*" e "*feedback* do sublíder ao PL sobre alcançar o fluxo principal". Para estas mudanças foram utilizados os novos parâmetros "IPAction", "ClientMode", "target" e "MoveAction".

Vejamos agora a composição das mensagens e o papel delas na comunicação entre o PL e os clientes a ele conectados.

MSGCODE_PID: Por gerenciar as mensagens de diversos clientes ao mesmo tempo, a instância da *CommMulticast* do PL precisa identificar cada um de forma única. Aproveitando o fato de que cada cliente é tratado por uma *thread* exclusivamente criada para ele no gerenciador de sessões do PL e que cada *thread* de um processo tem um identificador (ID) único, faz-se uso deste identificador para distingüir um cliente dos demais. O processo se dá da seguinte forma: o cliente tenta estabelecer conexão com o PL fazendo um pedido de abertura do objeto. O PL abre o objeto, cria a *thread* que tratará os eventos deste cliente e envia, junto com a resposta da requisição, o ID desta *thread*. O cliente, de posse deste ID, envia a mensagem MSGCODE_PID à *CommMul-*

ticast, com este identificador. Um *socket*[19] é criado para a comunicação com este cliente assim que o PL recebe esta mensagem e uma associação direta é feita entre o ID fornecido e o *socket* criado. Como resposta a esta mensagem o PL envia uma mensagem do tipo MSGCODE_IP, que será descrita mais adiante.

Em nível de código esta mensagem é definida em uma estrutura conforme nos mostra a tabela 2.1. Note que há um elemento de nome *code*. Ele é comum a todas as mensagens e serve unicamente para identificar qual é a mensagem que está sendo enviada (neste caso é informado que a mensagem é do tipo MSGCODE_PID).

Tipo	Nome
unsigned char	code
int	pid

Tabela 2.1: Dados transmitidos na mensagem MSGCODE_PID.

Nenhuma alteração foi feita nos parâmetros ou uso desta mensagem. Ela está apresentada nesta seção por ser necessária ao entendimento das demais mensagens e das trocas destas entre o PL e os clientes.

MSGCODE_IP: Esta mensagem é enviada pelo PL ao cliente informando a ele a qual grupo *multicast* ele deve se unir a fim de que passe a receber os dados transmitidos ao grupo em questão e é enviada em resposta a dois tipos de mensagens recebidas: MSGCODE_PID, descrita acima, e MSGCODE_MOVE, a ser descrita mais adiante.

Quando um cliente se movimenta ou entra no sistema, ele precisa ser alocado em algum grupo, seja um grupo novo ou um outro já existente. Esta mensagem informa, através dos campos mostrados na tabela 2.2, a qual grupo o cliente

[19] Podemos definir um *socket* como a combinação de um endereço IP, um protocolo, e o número da porta do protocolo usados para a comunicação entre dois processos em uma rede de computadores [Stevens 1997].

deve se unir e em que condições. Através dos campos **address** e **port** é informado o endereço *multicast* do grupo e a ação que o cliente deve tomar é pelo parâmetro **ipAction**. A lista de possíveis ações está definida na tabela 2.3.

Dependendo de cada ação, o PL precisa fornecer outros parâmetros que fazem parte da mensagem e que estão descritos a seguir.

Parâmetro *block*: no caso de o cliente ser alocado em um grupo que esteja recebendo um bloco dentro do intervalo δ_{before}, ele não poderá avançar para o exato bloco que solicitou ao PL, mas para um bloco anterior, conforme explicado no caso 1 da seção 2e. Esta informação (o bloco anterior) é fornecida no parâmetro *block* e só é usada nas mensagens onde **ipAction** é igual a JOIN_THIS. Para as demais situações, como a de o cliente ser alocado em um grupo que esteja recebendo um bloco dentro do intervalo δ_{after} ou a de ele ser alocado em um novo grupo criado para ele, este parâmetro não precisa ser usado e é enviado com o valor -1.

Parâmetro *target*: no caso de o cliente ser alocado em um grupo que esteja recebendo um bloco dentro do intervalo δ_{after}, ele receberá os blocos *multicast* deste grupo mas deverá fazer o *patch* dos blocos que não tem; só que o cliente precisa saber por quanto tempo terá que fazê-lo. Esta informação é fornecida pelo parâmetro **target** e só é usada nas mensagens onde o valor do parâmetro **ipAction** é igual a JOIN_THIS.

Parâmetro *mode*: quando o cliente é alocado em um grupo, ele precisa saber qual é o papel dele no sistema e, caso seja alocado em um grupo ativo, precisa saber se será o responsável por fazer as requisições do mesmo ou se ficará recebendo os dados passivamente. Os possíveis papéis de um cliente no sistema são listados e descritos na tabela 2.4. Esta informação é fornecida pelo parâmetro **mode** e é utilizada em todas as mensagens do tipo MSGCODE_IP, independente do parâmetro **ipAction**, e MSGCODE_MODE, esta última a ser descrita a seguir.

Tipo	Nome
unsigned char	code
char	address[17]
unsigned short	port
ClientMode	mode
int	block
int	target
IPAction	ipAction

Tabela 2.2: Dados transmitidos na mensagem MSGCODE_IP.

Ação	Descrição
JOIN_THIS	Cliente deverá se unir a este grupo *multicast*.
LEAVE_THIS	Cliente deverá abandonar este grupo *multicast*.
LEAVE_ALL	Cliente deverá abandonar todos os grupos *multicast*.
KEEP_THIS	Cliente deve permanecer neste grupo *multicast*.

Tabela 2.3: Possíveis instruções do PL ao cliente.

MSGCODE_MOVE: esta mensagem é enviada pelo cliente ao PL sempre que faz uma interação informando qual o seu movimento e para que ponto. Os parâmetros que são enviados são **block** e **action**, conforme exemplificado na tabela 2.5, e os possíveis valores para o parâmetro **action** são apresentados na tabela 2.6.

MSGCODE_MODE: existem situações em que um cliente não faz nenhuma interação mas alguns eventos fazem com que seja necessário alterar seu papel no sistema. Por exemplo, quando um cliente que é líder abandona seu grupo, o membro seguinte, que não fez nenhuma interação, deve ser promovido a líder do grupo[20]. Nesta, como em outras situações, o cliente é avisado de que deve

[20]Como visto na figura 2.20, o membro seguinte é aquele que entrou no grupo após o seu anterior. Não há nenhum critério especial na escolha de um substituto de um líder; toma-se indiscriminadamente o que vier imediatamente após na lista encadeada do grupo em questão

Papel do cliente	Descrição
PASSIVE	Cliente está ativo escutando algum fluxo *multicast*.
LEADER	Cliente é líder de algum grupo.
SUBLEADER	Cliente é sublíder de algum grupo.
INACTIVE	Cliente está na lista de inativos.
UNAVAILABLE	Cliente que não está em grupo nenhum (ativos nem inativos), isto é, está indisponível. Usado durante transição de um grupo a outro.

Tabela 2.4: Possíveis papéis do cliente no sistema.

Tipo	Nome
unsigned char	code
long	block
MoveAction	action

Tabela 2.5: Dados transmitidos na mensagem MSGCODE_MOVE.

mudar seu papel no sistema do antigo para o indicado pelo PL através do parâmetro **mode** desta mensagem. O parâmetro **block** é usado quando se quer notificar um cliente de que ele foi promovido a líder ou sublíder e informa qual o bloco que ele deve requisitar para o grupo[21]. Quando a mensagem retira a liderança de um cliente este parâmetro é enviado com o valor -1, que significa que o cliente deve ignorá-lo.

No caso de esta mensagem ser usada para notificar um líder de que seu grupo deverá se unir a um novo grupo N recentemente criado, este líder, agora eleito sublíder de N, precisa saber qual é o momento em que ele alcançará o grupo principal, quando haverá a fusão dos grupos e extinção do fluxo do subgrupo. Esta informação é passada pelo parâmetro **target**.

[21] Um cliente que herda a liderança devido à saída de seu antigo líder do grupo é dito "líder forçado". A principal diferença de um "líder natural" para um "líder forçado" é que aquele requisita para o grupo os mesmos blocos que requisita para si, enquanto este faz requisições de blocos que ainda não precisa.

Ação	Descrição
ACTION_PLAY	Requisição de exibição de vídeo.
ACTION_STOP	Requisição de interrupção da exibição do vídeo. Neste caso o vídeo é interrompido e a barra de progresso de exibição é reiniciada para a posição inicial do vídeo.
ACTION_PAUSE	Requisição de interrupção da exibição do vídeo. Neste caso o vídeo é interrompido e a barra de progresso se mantém no mesmo ponto para posterior prosseguimento.
ACTION_FORWARD	Requisição de salto para uma posição posterior à atual.
ACTION_REWIND	Requisição de salto para uma posição anterior à atual.
ACTION_READBUFFER	Cliente já tem os blocos no disco e vai ler de lá.
ACTION_REACHLEADER	Cliente terminou o patch.

Tabela 2.6: Possíveis movimentos de um cliente.

Na figura 2.21 exemplifica-se a troca de mensagens inicial entre o cliente e o PL e o funcionamento da abertura de sessão, fluxo e do objeto no servidor RIO[22], isto é, uma situação em que um novo cliente entra no sistema e não há um fluxo *multicast* em andamento que o atenda. Uma vez abertos os ítens citados, o cliente envia uma MSGCODE_PID ao PL, que responde com uma MSGCODE_IP. Trocadas estas mensagens o cliente começa a requisitar e a receber os blocos. De início o cliente requisita o número de blocos necessários para encher o seu *PlayOutBuffer*. Feito isto o cliente pergunta ao servidor, através da mensagem de requisição *canstart*, se o

[22]Como resposta ao pedido de abertura de fluxo, o servidor responde, caso consiga abrir o fluxo, o número máximo de requisições pendentes que o cliente pode ter. Esta informação é útil ao cliente para que ele evite dar rajadas de requisições maiores que a quantidade de requisições pendentes queo servidor lhe permite ter.

Tipo	Nome
unsigned char	code
ClientMode	mode
long	block
long	target

Tabela 2.7: Dados transmitidos na mensagem MSGCODE_MODE.

servidor está com seu *buffer* de envio cheio e preparado para atender as requisições futuras. O objetivo deste *buffer* do lado do servidor é evitar possíveis atrasos no envio de blocos devido ao tempo necessário para a leitura do disco. Ao receber a resposta *canstart* do servidor o cliente começa a consumir os dados e a fazer novas requisições conforme a necessidade. Para entender a necessidade desta última troca de mensagens o leitor pode obter a descrição completa de seu objetivo em [Cardozo 2002].

Ainda sobre esta figura, representa-se nela um caso em que não há um grupo disponível para o cliente, logo um grupo é criado para ele. Neste caso, note que não há a necessidade de o cliente fazer *patch* já que receberá do fluxo *multicast* exatamente o bloco que requisitou.

A troca de mensagens é sutilmente diferente quando o cliente já está no sistema, isto é, quando já fez as requisições de abertura de sessão, fluxo e objeto. Independente de seu papel atual, ao interagir ele envia uma mensagem com detalhes de seu movimento e é informado pelo PL sobre o grupo em que foi alocado. Na figura 2.22 temos um exemplo de um cliente que estava tocando um bloco qualquer e saltou para o bloco 50. Neste exemplo o cliente não tem os blocos 50, 51, ... no disco.

Um terceiro e último caso de troca de mensagens entre o cliente e o PL é quando alguma mudança acontece no sistema e há a necessidade de mudança dos papéis de um ou mais clientes, o PL envia uma mensagem a cada um destes clientes informando o que eles devem fazer através da mensagem MSGCODE_MODE. Há diversas situações em que isto pode acontecer, como por exemplo quando um líder

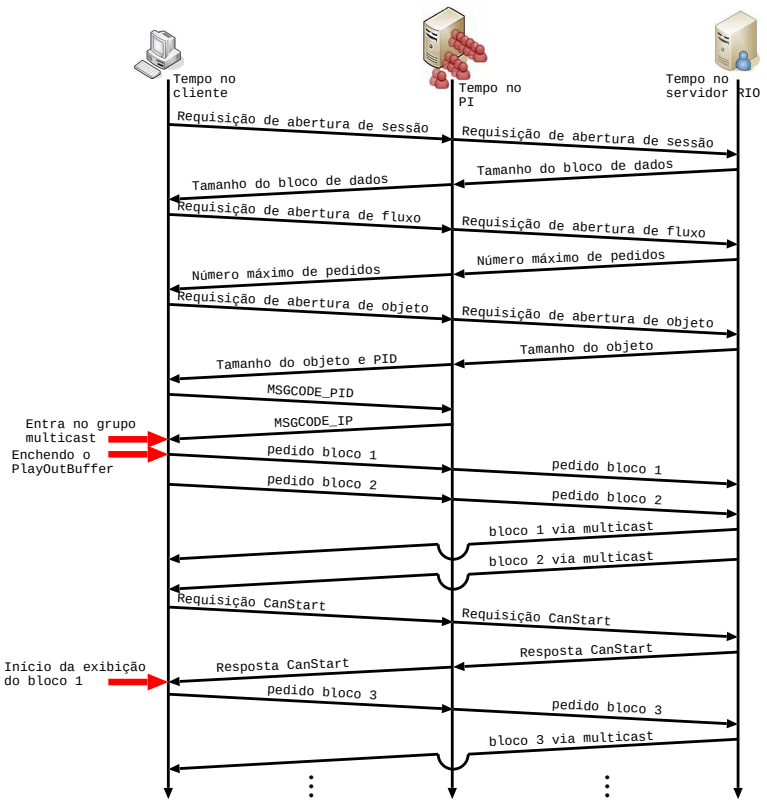


Figura 2.21: Troca de mensagens iniciais do cliente com o PL.

abandona seu grupo, o que torna necessária a eleição de um novo líder. Quando isto acontece o PL envia uma mensagem ao novo líder informando que ele foi promovido a líder de seu grupo e que agora é responsável por fazer as requisições do mesmo. Um outro caso interessante, que só acontece no caso do PL estar trabalhando com a técnica PIE, é quando um novo grupo, digamos N, é criado e encontra-se um grupo já existente no sistema, digamos G, que está dentro da janela δ_{merge} de N. Neste caso cada membro de G será notificado de que é um submembro de N e que, portanto, deverá escutar, além do fluxo de G, o fluxo de N. O antigo líder de G é notificado de que é sublíder de N e que com isto deve, além de fazer as requisições para os membros de G, escutar os dados transmitidos para N. Na figura 2.23 exemplificamos esta troca de mensagens considerando que quem recebe a mensagem é um cliente que está recebendo passivamente os blocos de seu grupo e é eleito líder deste mesmo grupo. No exemplo o grupo está recebendo os blocos 50, 51, ... e o cliente que recebe a mensagem está exibindo, ainda, o bloco 20. Note que a iniciativa de

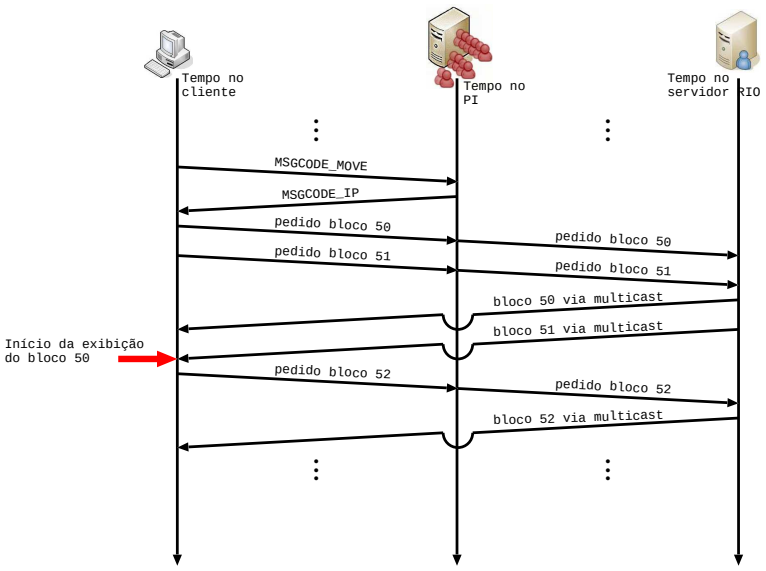


Figura 2.22: Troca de mensagens entre cliente e PL após interação.

envio de mensagem parte do PL e que, apesar de requisitar os blocos 54, 55, ..., o cliente continua sua exibição anterior normalmente[23].

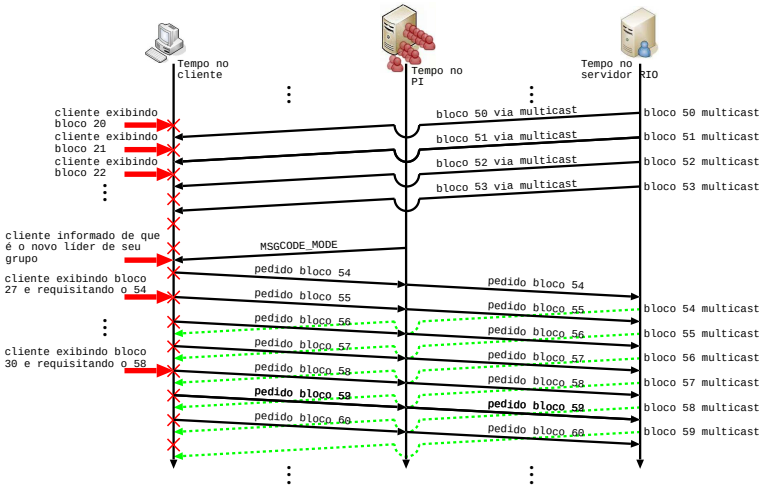


Figura 2.23: Troca de mensagens entre cliente e PL após mudanças no sistema.

O PL, conforme foi sendo desenvolvido, passou por algumas fases. Em sua concepção inicial implementava a técnica PI pura e simplesmente. Conforme foi

[23] As linhas que aparecem tracejadas na figura 2.23 não significam nada em especial; representam transmissão de dados *multicast* tal qual as anteriores em estilo sólido, apenas se apresentam desta forma por questões de visibilidade.

feita a implementação da técnica PIE, gerou-se uma versão intermediária entre estas que é a de PI sem janela ativa. Sendo assim é possível, hoje, no uso do PL, adotar-se uma de três políticas de gerenciamento de grupos (compartilhamento de banda):

1. *Patching* **Interativo com janela ativa**

   O algoritmo adotado é o apresentado na figura 2.8.

2. *Patching* **Interativo sem janela ativa**

   O algoritmo adotado é o apresentado na figura 2.8 com exceção das linhas 2, 3 e 4, isto é, removem-se as linhas que agrupam clientes considerando a janela-ativa, listadas abaixo:

   linha 2: se((janela_ativa) e (posição_grupo[janela_ativa]>posição_de_movimento))

   linha 3: Insere cliente no grupo com a janela_ativa;

   linha 4: senão

   Com isto o algoritmo apresenta-se conforme o exibido na figura 2.24.

3. *Patching* **Interativo Eficiente**

   A técnica PIE é semelhante à PI sem janela ativa e difere-se apenas na situação em que um novo grupo é criado. Neste caso procura-se um grupo já existente no sistema que possa se unir a este novo grupo. Nesta busca considera-se apenas os grupos que estejam a uma distância máxima de δ_{merge} do grupo que acabou de ser criado, isto é, se este novo grupo está recebendo o bloco x, por exemplo, então só serão considerados grupos que estejam recebendo o bloco y onde $x - y >= \delta_{\text{merge}}$. Uma vez escolhido um grupo desta janela δ_{merge}, cada um de seus membros é avisado de que deve escutar, além do fluxo de seu grupo atual, o fluxo deste novo grupo criado. O líder deste grupo é, ainda, avisado de que deve continuar requisitando blocos para seu grupo, que acaba de se tornar subgrupo do grupo que acabou de ser criado.

   Na figura 2.25 apresentamos o algoritmo da técnica PIE para facilitar o entendimento.

```
Patching_Interativo_Sem_Janela( cliente )
  // 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 2.24: Algoritmo da técnica PI sem janela ativa.

```
Patching_Interativo_Eficiente( cliente )
  // Verifica se o cliente pode entrar em algum outro grupo
  i = 0;
  grupo_delta_before     = -1;
  grupo_delta_after      = -1;
  grupo_delta_merge      = -1;
  distância_delta_before = INFINITO;
  distância_delta_after  = INFINITO;
  distância_delta_merge  = 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
      enquanto existir algum fluxo para o mesmo vídeo do cliente
        se( ( ( posição_de_movimento - DELTA_MERGE ) <= 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_merge = i;
        incrementa valor de i;
      Insere grupo i neste novo grupo
Fim-Patching_Interativo
```

Figura 2.25: Algoritmo da técnica PIE.

Administração dos grupos

Na figura 2.20 apresentamos a estrutura utilizada no gerenciamento dos grupos e de seus membros. Vimos também que os membros mais à esquerda são os líderes de seus respectivos grupos. Há ainda dois outros possíveis papéis ainda não apresentados de um membro de um grupo: sublíder e submembro. Definimos como sublíder o líder de um grupo G que é notificado de que deve continuar fazendo suas requisições *multicast* para seu grupo e que deve passar a ouvir, também, um segundo fluxo *multicast* de um outro grupo G'. Nesta situação, este líder passa a ser sublíder de G' pois G' já tem um líder. Seguindo o mesmo raciocínio, cada membro de G torna-se submembro de G' pois escuta dois fluxos *multicast*, o de G e o de G'. Configurada esta situação dizemos que G é um subgrupo de G'.

A única diferença de um sublíder para um líder é a de que o sublíder escuta dois fluxos *multicast* dos quais um está prestes a ser eliminado, o que acontece assim que os fluxos se unem.

Obs:	A relação entre o líder de um grupo e um sublíder deste mesmo grupo é a mesma entre este líder e um outro membro ou submembro qualquer: todos são igualmente beneficiados por suas requisições *multicast* e podem, assim que o líder abandonar o grupo, tomar seu lugar bastando, para isto, ser o próximo na lista encadeada.

Para facilitar o entendimento, considere o cenário da figura 2.20 e a entrada de um novo cliente, o cliente 7, no sistema tal que não exista nenhum grupo ao qual ele possa ser alocado. Nesta situação um novo grupo N é criado. Como visto na seção 2.2, item (f)(pg. 20), neste momento procura-se no sistema um grupo M que esteja recebendo blocos a uma distância máxima de δ_{merge} de N e que nenhum de seus membros esteja fazendo *patching*. Considere, na figura 2.20, que o segundo grupo, composto pelos clientes 2, 3 e 4, seja o grupo M escolhido. Isto significa que M se tornará subgrupo de N. A estrutura da figura 2.20 sofre as alterações apresentadas na figura 2.26.

Obs: Assim como todo membro à direita de seu líder na estrutura da figura 2.20 é membro do grupo deste líder, todo cliente à direita do sublíder também é submembro deste grupo. Quando, por algum motivo, um submembro é promovido a membro, ele é realocado na estrutura de forma a ficar à esquerda de seu sublíder: isto significa que ele não escutará mais o fluxo *multicast* deste sublíder. A situação em que isto ocorre será apresentada mais à frente no texto.

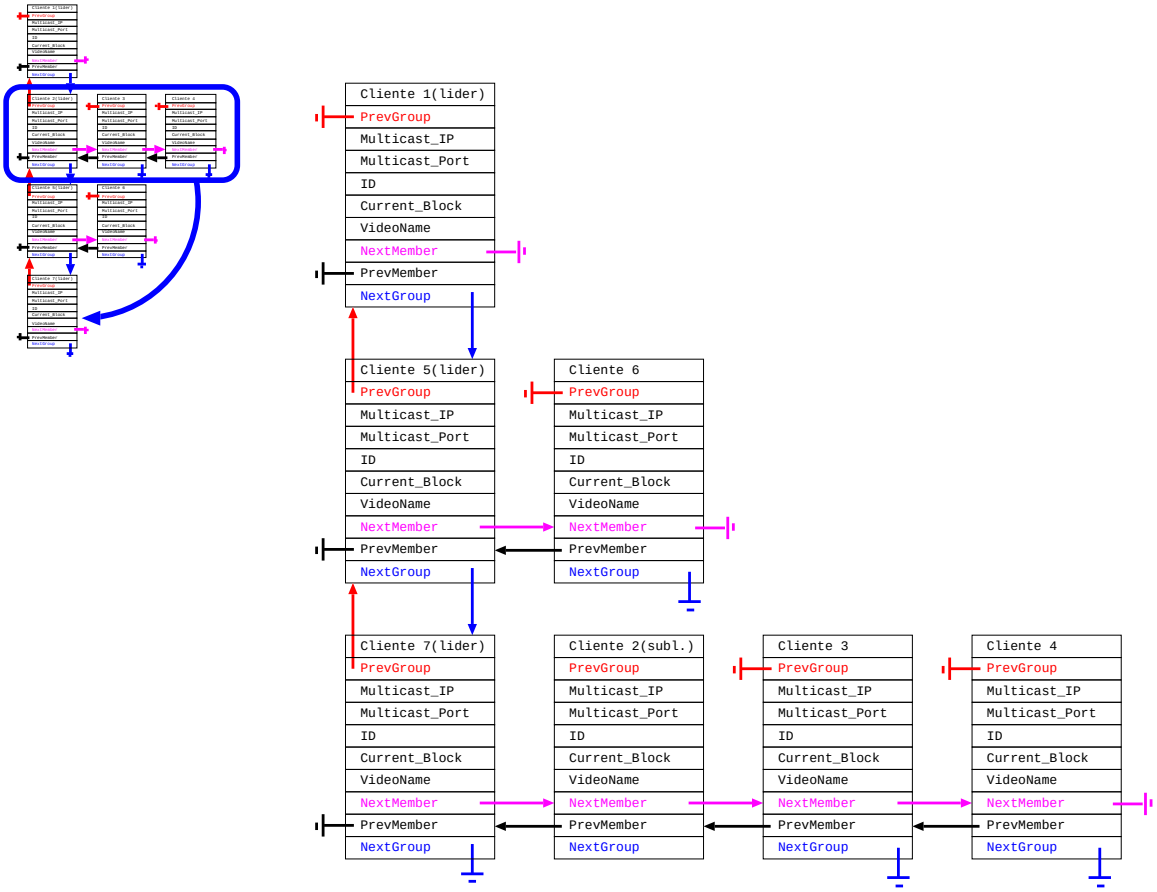


Figura 2.26: Estrutura dos subgrupos administrados pelo PL.

Definimos por cliente passivo o cliente que é membro de um grupo mas não é líder nem sublíder do mesmo. Isto significa que este cliente apenas escuta o(s) fluxo(s) *multicast* e recebe os blocos de vídeo sem precisar fazer as requisições dos mesmos (requisições estas feitas pelo líder e, quando for o caso, pelo sublíder do grupo).

A administração dos grupos e a alocação e remanejamento dos clientes nos grupos

se dá através de mensagens enviadas do PL aos clientes em resposta ao movimento de um ou mais destes. As mensagens, seus conteúdos e a que se destinam, foram apresentadas na seção 2.3.2. Serão descritos aqui as possíveis situações que acontecem de acordo com a interação dos usuários:

Cliente interage e há um grupo para ele: este caso já foi apresentado. Vamos nos aprofundar e apresentá-lo mais detalhadamente.

Quando o PL encontra um grupo para um cliente c que interagiu, temos uma de duas possibilidades: (1) c foi alocado em um grupo que está recebendo blocos posteriores ao que ele requisitou com a interação, sito é, o cliente interagiu para uma região dentro do δ_{after} deste grupo; (2) c foi alocado em um grupo que está recebendo blocos anteriores ao que ele requisitou com a interação, sito é, o cliente interagiu para uma região dentro do δ_{before} deste grupo.

Em ambos os casos o PL alocará o cliente no devido grupo e enviará a ele uma mensagem MSGCODE_IP informando-o que deve escutar o fluxo deste grupo.

Se a situação for a (2), o cliente precisa saber qual é o bloco que precisa começar a exibir. Esta informação é passada a ele pelo parâmetro **block** da mensagem recebida. Como o cliente está recebendo blocos anteriores ao requisitado, não haverá necessidade de fazer *patching* e, portanto, o parâmetro **target** não é utilizado, o que faz com que ele seja enviado com o valor -1, indicando que deve ser ignorado pelo cliente.

Se a situação for a (1), o cliente exibirá os blocos que requisitou, o que faz com que o parâmetro **block** seja desnecessário e, por isso, é enviado com o valor -1, que faz com que o cliente o ignore. Como c está atrás do grupo ao qual quer se unir, terá que fazer *patching* até um determinado momento quando, então, não será mais necessário fazer requisições pois a fusão com o grupo já vai ter ocorrido. O ponto desta fusão é informado ao cliente pelo PL através do parâmetro **target** da mensagem MSGCODE_IP.

Cliente interage e não há um grupo para ele: neste caso um novo grupo N é criado para este cliente. Feito isto são procurados no sistema os grupos que estejam recebendo blocos anteriores antes e a uma distância máxima δ_{merge} deste novo grupo. Outro requisito é que nenhum de seus membros esteja fazendo *patching*. Destes grupos, escolhe-se aquele que estiver mais próximo do novo grupo N, digamos G, e envia-se uma mensagem MSGCODE_IP aos seus membros informando-lhes que devem começar a ouvir, também, o fluxo *multicast* enviado para N. O líder de G, além desta mensagem, é informado de que tornou-se sublíder de N, que deve continuar solicitando os blocos para os membros de G (agora subgrupo de N) mas que deve, também, escutar os dados do fluxo enviado para N até que receba do fluxo de G o bloco *target* (informação obtida nesta mesma mensagem MSGCODE_IP).

Novo cliente entra no sistema: neste caso a reação do sistema é idêntica à do caso em que um cliente do sistema interage portanto recai em um dos dois casos anteriores.

As 3 situações acima estão inalteradas e não sofreram modificações neste trabalho. Os campos extras adicionados nas mensagens trocadas nas mesmas não foram utilizados, como comentado no texto. Abaixo apresentamos outras situações que não existiam antes ou que existiam mas tiveram seus tratamentos alterados devido às modificações que nossas contribuições fizeram necessárias.

Líder l de um grupo G interage: um outro grupo é escolhido para l ainda que ele interaja para um lugar dentro de δ_{after} ou δ_{before} de G[24] e o próximo membro de G assume a liderança no lugar de l. Se não houver mais clientes em G além de l então G será extinto no exato momento da interação. Caso o próximo

[24]Isto evita problemas como o caso de um líder de um grupo com apenas ele de membro que interage e cai em seu próprio grupo. Como o grupo é extinto assim que ele interage, ele não pode ser realocado em seu grupo de origem; mas como isto seria semelhante a criar um novo grupo para ele então esta é a solução adotada pois é simples além de evitar problemas.

membro de G seja um sublíder s, então G será extinto e o subgrupo de s voltará a ser um grupo, bem como s voltará a ser líder. Neste caso todos os submembros de G[25] serão avisados, através da mensagem MSGCODE_IP, que deverão parar de escutar o fluxo de G pois ele será extinto e s será avisado de que foi promovido de sublíder a líder através da mensagem MSGCODE_MODE.

Sublíder s de um grupo interage: neste caso o próximo cliente do subgrupo, caso exista, é eleito sublíder, sendo informado disto pela mensagem MSGCODE_MODE, e s recai nas duas primeiras situações aqui apresentadas.

Cliente passivo ultrapassa seu líder: esta situação parece difícil de acontecer, pois todos os clientes passivos necessariamente estão tocando um bloco anterior ao do líder do grupo. Considere a seguinte situação: Um cliente c interage para o bloco 20 e é alocado no grupo G, de um único membro l que obviamente é o líder deste grupo. Se l estiver, neste momento, requisitando o bloco 100 para G, então tem-se que o *target* de c é 100. Imagine que c receba os blocos 100, 101, 102, ..., 110 enquanto toca os blocos 20, 21, 22 ..., 30 e então l faça uma interação e saia de G. Considere ainda que enquanto c recebia tais blocos um novo cliente d entrou no sistema requisitando o bloco 90 e foi alocado em G. Nesta situação, como c é o próximo da lista, é eleito líder de G e, por não ter os blocos 31, 32, ..., 99, continuará fazendo o *patch* dos mesmos além de fazer as requisições dos blocos 111, 112, ... para G (o que significa dizer que c é "líder forçado de G"). Sabendo-se que o *player* do cliente d, por diferenças de processador, consome os dados ligeiramente mais rápido que o *player* de c, é perfeitamente possível que, em algum momento, d precise de blocos que c ainda não tenha requisitado para o grupo[26]. Considere que isto aconteça no bloco 150, ou seja, enquanto c solicita o bloco 150 para o grupo, d já exibiu este bloco e precisa agora do bloco 151. Como d não tem este bloco no disco

[25] Isto inclui o sublíder s.

[26] Lembre-se de que como o sistema é de vídeo-sob-demanda, c requisita um novo bloco para o grupo a cada bloco que consome. Como seu consumo é mais lento do que o de d, tem-se que em algum momento d não terá recebido os blocos de vídeo a tempo.

ele faz uma requisição *unicast* para o PL, que por sua vez sabe que d está em G. Analisando a situação, o PL verifica que d solicitou um bloco posterior ao que seu grupo está recebendo e que isto se deve a uma ultrapassagem. Para evitar que d fique fazendo estas requisições via *unicast* (já que elas chegarão, de qualquer forma, via *multicast* quando c requisitá-las no futuro) então o PL elege d o novo líder de G e destitui c da liderança. Isto é feito enviando-se uma mensagem MSGCODE_MODE para c e d informando a um que não é mais líder e a outro que passou a ser líder. É importante, ainda, que d não deixe de solicitar nenhum bloco para o grupo.Se d consome os dados muito mais rápido que c, pode ser que antes que ele seja notificado de que foi eleito líder ele já tenha solicitado vários blocos via *unicast*. Para evitar isto, o PL informa a d qual o bloco inicial i do pedido para o grupo G, isto é, qual o bloco que os membros de G esperam receber pelo fluxo *multicast*. Caso d esteja requisitando x blocos a frente de i então ele faz uma mini rajada de requisições de todos os blocos entre i e $i + x$, assegurando que nenhum bloco deixará de ser pedido por conta desta mudança de líder.

Esta situação não era tratada na implementação anterior do PL mas foi solucionada ao ser percebida durante a execução deste trabalho.

Cliente passivo ultrapassa seu sublíder: neste caso o cliente, que era um submembro, passa automaticamente para o grupo principal, deixando de ser um submembro tornando-se membro e, conseqüentemente, deixa de escutar o fluxo enviado para o subgrupo. A estrutura é modificada de forma que este membro não esteja mais à direita do sublíder do grupo e que passe a se posicionar à sua esquerda. Caso o subgrupo ainda não tenha alcançado o grupo principal então o cliente fará o *patch* destes blocos.

Na figura 2.27 mostramos como é reorganizada a estrutura dos grupos neste tipo de ultrapassagem. Se o cliente 4, submembro do grupo 3, ultrapassa o cliente 2, sublíder do mesmo grupo, então ele é realocado na estrutura de forma que fique à esquerda do sublíder (o que indica que ele para de escutar o

fluxo *multicast* do cliente 2) e à direita do líder (o que indica que ele continua escutando o fluxo *multicast* do cliente 7) no caso do cliente 4, submembro do cliente 2, após um sublíder ser ultrapassado por um submembro.

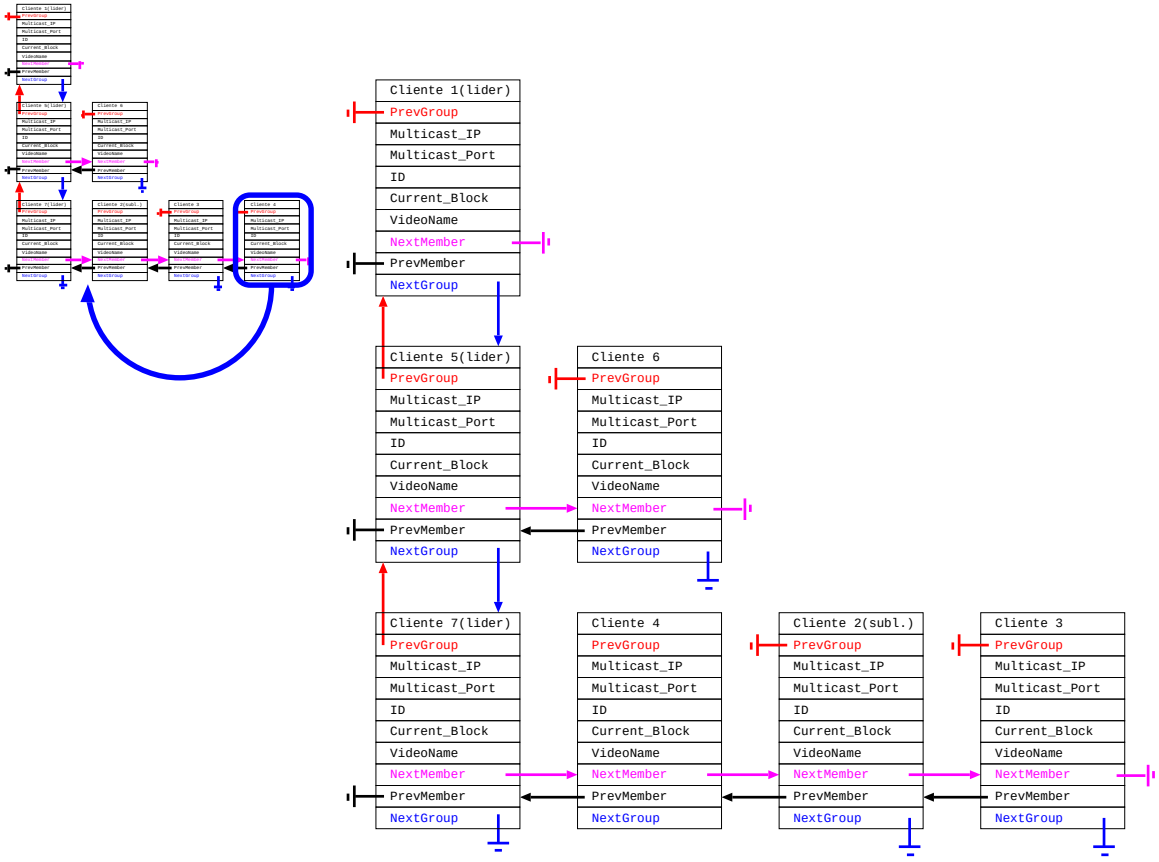


Figura 2.27: Estrutura dos subgrupos após um submembro tornar-se membro.

Líder requisita último bloco do vídeo: nesta situação temos um grupo que não faz mais requisições, está prestes a se desfazer e que, portanto, não deveria aceitar nenhum novo membro. Qualquer cliente que interagir e for alocado a este grupo não receberá bloco algum dele pois as requisições *multicast* já foram finalizadas e ficaria em *patching* até o final do vídeo ou até que interagisse novamente. Neste caso, é mais vantagem que seja criado um novo grupo para este cliente que interagiu pois desta forma este fluxo, que na situação anterior seria de *patching*, será um fluxo *multicast* e, portanto, poderá ser compartilhado com outros clientes que vierem a interagir para uma região próxima a dele.

Implementamos esta vantagem da seguinte forma: quando um líder de um grupo requisita o último bloco do vídeo, ele é destituído da liderança mas nenhum outro membro é eleito líder. Nesta situação temos um grupo ativo sem um líder[27]. Com isto nenhum cliente que interagir para uma área dentro de δ_{after} deste grupo será alocado nele pois o PL só aloca um cliente em um grupo ativo que tenha líder.

Quanto a este grupo sem líder, conforme seus membros chegarem à posição final do vídeo e forem dando *stop*, o grupo vai se desfazendo. Quando seu último membro terminar sua execução ou sair do grupo por outro motivo, este grupo será finalmente extinto do sistema.

Subgrupo alcança seu grupo: este caso é o mesmo que um sublíder alcançar seu *target*. Quando isto acontece o sublíder informa ao PL que já alcançou o grupo e que a fusão dos grupos já pode ser feita e que o fluxo do subgrupo pode ser extinto, mas o sublíder só sabe se alcançou o grupo principal se lhe for informado seu *target* no momento em que for transformado em um sublíder[28].

Uma vez que o sublíder informa o PL de que alcançou seu *target*, o PL envia uma mensagem MSGCODE_IP com o parâmetro **action** igual a LEAVE_THIS para cada submembro deste subgrupo. Esta mensagem informa aos seus destinatários que eles devem parar de ouvir o fluxo enviado ao subgrupo, o que significa que devem sair deste subgrupo, pois o mesmo será extinto. Estes clientes passam a desempenhar o papel de clientes passivos[29] no sistema e não têm mais um *target* a alcançar.

[27] Note que apesar de não ter um líder, este grupo pode, ainda, ter um sublíder.

[28] Lembre-se de que "tornar-se sublíder" é conseqüência de uma interação cuja iniciativa não partiu deste cliente, logo ele precisa ser informado de qual grupo escutará o segundo fluxo *multicast* e até quando deve fazê-lo (seu *target*).

[29] O conceito de cliente passivo foi definido na seção 2.4.1.

2.4.2 Contribuições para o cliente de vídeo

Durante a execução deste trabalho algumas falhas e vulnerabilidades foram identificadas no cliente de vídeo. Cada uma das soluções foi cuidadosamente estudada quanto aos seus aspectos práticos e de impacto na estrutura original do cliente. Em alguns casos as mudanças foram superficiais mas em outros foram significativas e tiveram como pré-requisito a reestruturação de parte da arquitetura do cliente. Nesta seção apresentamos as falhas encontradas e as soluções implementadas.

Cliente possui blocos não contíguos na sua *cache*: Quando um cliente entra em um grupo, recebe uma mensagem que informa seu *target*. O comportamento normal do cliente nesta situação é fazer seu *patch*, que são as requisições via *unicast*, até que alcance seu grupo (seu *target*). Neste momento ele envia uma mensagem ao PL informando que alcançou seu grupo e que passará a escutar passivamente o fluxo *multicast* do mesmo.

Considere a seguinte situação: um cliente c entra no sistema e salta imediatamente para o bloco 100. Após tocar 50 blocos ele interage saltando para o bloco 80 e é alocado no grupo G cujo líder é o cliente l. Considere que l está requisitando o bloco 160 para G. Nesta situação, c tem em sua *cache* os blocos 100, 101, ..., 150 e, enquanto toca os blocos 80, 81, ... (recebidos via *patch*), recebe os blocos 160, 161, ... de G. Quando c chegar ao centésimo bloco, perceberá que já o tem em sua *cache*. Neste ponto entra a otimização feita neste trabalho.

Na implementação anterior o cliente enviava uma mensagem ao PL informando que encontrou os dados de que precisava em sua *cache* e que, portanto, não necessitava mais receber os dados vindos de seu grupo *multicast*[30]. Com isto o cliente era transferido para o grupo de inativos.

Durante a execução deste trabalho reparamos que esta situação desperdiçava

[30]Para o PL esta situação era a mesma da de o cliente interagir para uma região cujos blocos que precisa já estão em sua *cache*.

uma excelente oportunidade de economia de banda pois na maioria dos casos este cliente, ao terminar de passar pelo trecho de blocos que tinha na *cache* voltava a fazer requisição *unicast* e acabava sendo alocado no mesmo grupo ao qual já fazia parte, só que com uma defasagem: durante o período de tempo em que ficou na lista de inativos ele deixou de receber os blocos enviados para este grupo.

Ao percebermos isto tratamos esta situação da seguinte forma: sempre que um cliente que ainda não alcançou seu *target*, isto é, está em *patching* percebe que tem os blocos do *patch* na *cache*, ele verifica se tem todos os blocos da posição atual até seu *target*. Caso tenha, ele envia ao PL uma mensagem informando que alcançou seu líder, o que significa que não fará mais requisições *unicast*.

Cliente já recebeu fragmentos de um bloco: Na seção 2.3.1 explicamos que um bloco de vídeo é composto por noventa fragmentos e que a finalidade disto é minimizar o efeito de perdas na rede. Quando um cliente solicita um bloco de vídeo e percebe que o mesmo não chegou por completo no momento de sua exibição então ele tenta recuperar todos os fragmentos disponíveis daquele bloco e os envia ao *player* imediatamente. Esta liberação de fragmentos é feita pelo método *FreeBlock*, descrito sucintamente na seção 2.3.4.

Na implementação anterior o método *FreeBlock* só era chamado em uma situação específica: quando o cliente termina de exibir um bloco de vídeo, ele libera a posição que este ocupa no *PlayOutBuffer*, requisita um novo bloco ao servidor (caso não tenha tal bloco na *cache*) e registra no PlayOutBuffer que a posição recentemente liberada destina-se a receber o bloco que acabou de requisitar. Se no momento de exibir tal bloco o cliente perceber que o mesmo ainda não se encontra no PlayOutBuffer, então a *FreeBlock* é chamada para que sejam liberados os fragmentos que já estiverem disponíveis do referido bloco pois sua exibição será feita naquele exato momento.

Nos deparamos com outra situação, além desta, em que o uso do método *FreeBlock* seria valioso: diminuição considerável de desperdício de memória

pelo cliente, descrita a seguir.

Detectamos, ao longo deste trabalho, uma situação bastante comum que gerava um desperdício de memória: sempre que o cliente interage o *PlayOutBuffer* é limpo. Quando a interação é um salto para trás ou para frente, além de se limpar este *buffer*, uma requisição inicial de preenchimento do mesmo é feita. Com isto em mente percebemos que quando o *PlayOutBuffer* é limpo mas nele estava registrada uma requisição ainda não atendida de um bloco que teve alguns fragmentos perdidos, então estes fragmentos jamais serão usados e ficarão alocados na memória até que o aplicativo seja finalizado, pois se acontecer de o cliente precisar daquele bloco em um outro momento, ele fará uma nova requisição ao servidor pois não o encontrará em sua *cache*.

A solução para este problema foi bem simples: implementamos um método de nome *thereAreFragments* que verifica se existem fragmentos disponíveis para um determinado bloco. Toda vez que um cliente precisar solicitar um bloco ao servidor, além de verificar se o mesmo já existe na *cache*, ele chamará este método e verificará se existem fragmentos disponíveis e, se existirem, chamará a *FreeBlock* para que os mesmos sejam gravados no *PlayOutBuffer* para exibição.

Cliente com muitos blocos pendentes: Quando um cliente *unicast* que se encontra em um ambiente com muitas perdas faz muitas interações, muitos blocos ficam pendentes pois o *PlayOutBuffer* é limpo constantemente. Esta situação se agrava para clientes de um fluxo *multicast* pois os fragmentos dos blocos que chegam por este fluxo começam a ocupar muita memória e somente são liberados para a *cache* ou o *PlayOutBuffer* quando o cliente precisa exibí-los. Como o cliente interage muito, então muitos blocos poderão ficar pendentes indefinidamente, nunca serão usados e só serão liberados da memória quando o aplicativo for finalizado. Quanto maior for o vídeo, mais crítica é a situação pois há um maior número de blocos, e conseqüentemente de fragmentos, a ser alocado na memória.

A solução dada para liberar a memória foi a seguinte: sempre que o primeiro fragmento de um bloco chega ao cliente, registramos a hora em que isto acontece. Toda vez que a *FreeBlock* é chamada nós percorremos a lista de requisições pendentes[31] e chamamos a *FreeBlock* para cada requisição pendente que tenha excedido um tempo máximo pré-determinado para a chegada do bloco completo. Para isto implementamos um método cujo nome é *FreePendentBlocks*, que é responsável por fazer a varredura dos blocos pendentes na tabela *hash* de requisições pendentes e chamar a *FreeBlock* para todos os blocos incompletos que ultrapassaram o tempo máximo tolerado para serem recebidos.

O valor escolhido para este tempo máximo foi empírico e igual ao quíntuplo do tamanho do *PlayOutBuffer*. Para um *PlayOutBuffer* de tamanho quatro, por exemplo, a tolerância é de 5*4=20 segundos.Para se ter uma idéia do impacto positivo desta solução, o uso de memória não ultrapassa mais de 10Mbytes. Em um teste, antes desta implementação, chegávamos facilmente a mais de 40Mbytes de memória por cliente disparado.

Um cliente de vídeo ultrapassa o líder de seu grupo: Esta situação acontece quando um cliente exibe o vídeo a uma taxa mais alta que a do líder de seu grupo. Ela já foi descrita e sua solução apresentada na página 61.

[31] Chamamos de requisição pendente um bloco que ainda não tenha sido inteiramente recebido, isto é, foram recebidos menos de noventa fragmentos, seja este bloco fruto de uma requisição *unicast*(feita pelo próprio cliente) ou *multicast*(feita pelo líder do grupo).

Capítulo 3

Ambiente dos experimentos

Inicialmente apresentaremos o ambiente onde foram realizados os experimentos,em seguida descreveremos a carga usada nos testes e a metodologia utilizada para realização dos experimentos.

3.1 Descrição do Ambiente de Emulação

A emulação consiste em disparar todos os elementos do sistema RIO, deixá-lo em operação durante um certo intervalo de tempo e coletar suas estatísticas.

Para o cumprimento do objetivo do trabalho é necessário reproduzir uma situação em que centenas ou milhares de clientes interagem durante a exibição de seus respectivos vídeos. De forma a viabilizar experimentos com um número expressivo de clientes interativos optamos por executar diversos clientes em uma mesma máquina. O único fator limitante da quantidade de clientes em uma mesma máquina é o desempenho pretendido. Quanto mais clientes são disparados, mais recursos (principalmente memória e CPU) são utilizados. Quando a utilização dos recursos da máquina é muito alto o desempenho dos clientes cai e isto é um indicativo de que a máquina não está agüentando a carga a ela submetida.

Desenvolvemos um emulador de clientes que é capaz de ler de um arquivo de log as ações do usuário ao longo de uma sessão. O log de ações representa o comportamento do cliente durante uma sessão. Na figura 3.1 temos um exemplo de um log. A primeira coluna indica o tempo em que a ação deve ser executada, a segunda coluna indica a interação a ser feita e a terceira coluna, no caso da interação JUMP, indica o bloco destino da operação, isto é, para onde deve ser dado o salto. Para as demais instruções esta terceira coluna informa qual bloco estava sendo exibido no momento em que a ação foi executada.

Em síntese o emulador comporta-se exatamente como o cliente com as seguintes diferenças:

1. Suas interações provém de um arquivo de ações. O emulador lê uma linha l_0 do arquivo, armazena o valor da primeira coluna em t_0, executa a instrução da segunda coluna (usando a terceira coluna caso necessário), lê a próxima linha l_1, armazena o valor de sua primeira coluna em t_1, calcula o intervalo de tempo entre t_0 e t_1 e dorme este período de tempo, isto é, dorme por um intervalo de tempo igual a $t_{sleep} = t_1 - t_0$ e, depois disto, repete o procedimento para a próxima linha (l_1).

2. Os blocos recebidos não são armazenados no disco ainda que isto seja informado ao emulador. No lugar disto é usado um vetor que informa se o emulador já recebeu o bloco. Isto é feito deste modo pois se disparássemos muitos clientes em uma mesma máquina e os mesmos armazenassem os blocos recebidos, o uso do disco seria tão intenso que em pouco tempo seu espaço livre seria esgotado[1]. Com este vetor emula-se o *BufferStream*, que é a *cache* do cliente de vídeo.

As estatísticas calculadas para cada experimento foram obtidas a partir dos logs gerados pelos aplicativos do sistema RIO, uma ferramenta valiosa de monitoramento, análise e geração de estatísticas de tráfegos de rede: o NeTraMeT (*Network Traffic*

[1] Um vídeo de 30 minutos tem, em média, 500Mb de tamanho.

```
0 PLAY 0
25 JUMP 87
42 JUMP 111
45 JUMP 147
61 JUMP 4254
69 JUMP 740
75 JUMP 147
77 JUMP 218
84 JUMP 422
129 JUMP 147
131 JUMP 218
134 JUMP 422
139 JUMP 740
142 JUMP 422
178 PAUSE 465
204 PLAY 465
217 JUMP 422
221 JUMP 2717
227 JUMP 218
235 JUMP 218
244 JUMP 422
246 JUMP 2717
310 JUMP 2898
336 JUMP 4254
342 JUMP 3397
354 JUMP 3559
389 JUMP 280
1096 STOP 1141
1108 PLAY 0
1123 JUMP 94
1156 JUMP 857
1179 JUMP 996
1182 JUMP 1119
1214 QUIT -1
```

Figura 3.1: Log de ações usado pelo emulador de cliente de vídeo.

Flow Measurement Tool) [Group 2008b]. O funcionamento e características mais detalhadas desta ferramenta serão apresentados adiante, na seção 3.3.

Daqui por diante não usaremos mais o termo "emulador" para nos referirmos ao emulador do cliente de vídeo. Ao fazê-lo diremos simplesmente "cliente" para evitar que se confunda o emulador do cliente com os *scripts* que executam os experimentos (disparam o servidor, o PL e os clientes). Ao dizermos "emulador" estaremos, doravante, nos referindo ao conjunto de *scripts* desenvolvidos para gerenciar o ambiente de emulação, ou seja, que disparam o servidor, o PL e os clientes. Este emulador executa o conjunto de ações descrito a seguir.

Prepara o ambiente: analisa cada uma das máquinas escolhidas para ser usada na emulação; compila o código de todos os elementos ativos do sistema RIO (*dispatcher*, *storage*, PL e cliente) em cada uma das máquinas[2];

Sincroniza versão em todas as máquinas: é importante que todas as máquinas tenham a mesma versão dos *scripts*, logs de ação do cliente e arquivos de configuração, por isso tudo o que está na máquina de onde é disparado o emulador é reproduzido fielmente nas demais usadas no processo;

Dispara o servidor RIO e o PL: além de dispará-los ele verifica se os mesmos foram carregados com sucesso;

Dispara todos os clientes: os clientes são disparados nas máquinas pré-determinadas sendo que, em uma mesma máquina, há um intervalo aleatório entre 0 e 10 segundos entre um disparo e outro. Os disparos não são todos feitos ao mesmo tempo para evitar uma rajada de pedidos no servidor;

Monitora a emulação: durante a execução os *scripts* do emulador verificam periodicamente se o servidor e o PL estão em operação e se todas as máquinas

[2]Isto é necessário pois por vezes o ambiente é composto de máquinas com distribuições *Linux* diferentes, e esta situação impossibilita compilarmos o código em uma máquina e copiarmos o binário para as demais por questões de incompatibilidade dos sistemas operacionais.

envolvidas na emulação estão acessíveis e em funcionamento. No caso do servidor (*dispatcher* ou *storage*) ou do PL terem terminado antes da hora por alguma falha (como travamento da máquina, espaço indisponível em memória ou em disco, rede indisponível, desligamento da máquina., dentre outros), a emulação é abortada e um *email* é enviado ao usuário que a disparou. No caso de uma ou mais máquinas que executam clientes não estarem funcionando ou acessíveis, a emulação continua mas um alerta é inserido no arquivo de log de eventos da emulação;

Finaliza a emulação: em um arquivo de configuração dos *scripts* de emulação é estabelecido se a mesma deve durar um período máximo de tempo ou se ela deve durar até o término da execução de todos os clientes. De acordo com a opção escolhida, a emulação é finalizada no devido tempo e o servidor, PL e clientes são forçados a terminar a execução caso ainda não o tenham feito;

Coleta os logs gerados: os clientes, o PL, o *dispatcher* e o *storage* geram logs durante sua execução. Todos estes logs são centralizados na máquina de onde foi disparada a emulação;

Envia um *email* ao usuário: ao término da emulação um *email* é enviado ao usuário que a disparou. Neste *email* são informados detalhes do andamento da mesma.

Os *scripts* que compõem o emulador estão apresentados a seguir:

- variaviesDeAmbiente.cfg: arquivo destinado a configurar o emulador e o ambiente de emulação. Nele são definidas as variáveis de ambiente usadas pelos *scripts* onde são informados o modo de operação dos clientes (com ou sem *cache*, qual o tamanho do *PlayOutBuffer*, etc.), onde se encontra o código fonte do sistema RIO e do NeTraMeT, onde estão instalados o servidor RIO e o PL, quantas vezes cada emulação deve ser repetida, para onde deve ser enviado o email com os resultados da emulação, etc;

- startApplication.sh: *script* usado para disparar o servidor RIO, o PL, o NeTraMeT e os clientes. No caso do disparo não ser dos clientes, o *script* verifica, após executar o devido aplicativo, se o mesmo foi carregado com sucesso e se continua executando;

- checkServers.sh: *script* usado periodicamente para verificar se o servidor RIO, o PL e o NeTraMeT estão em execução;

- loadCheck.sh: *script* usado periodicamente para coletar informações de uso de memória e cpu do PL, *dispatcher* e *storage*;

- postMorten.sh: *script* usado para finalizar os clientes, servidor, storage, PL e NeTraMeT que ainda estiverem rodando após o término da emulação e/ou fazer tratamentos nos logs destes após sua finalização;

- rodaExperimentoCompleto.sh: *Script* principal que faz uso dos outros *scripts* na administração da emulação. Este *script* oferece as funcionalidades de compilação distribuída, sincronização do ambiente de emulação, centralização dos logs distribuídos, análise sucinta dos logs coletados e verificação do ambiente de emulação para saber se o mesmo está apto e pronto a disparar os aplicativos.

Há ainda algumas ferramentas que foram criadas e que são indispensáveis para execução dos experimentos e coleta de estatísticas. Todas as ferramentas apresentadas a seguir, com exceção da primeira (movietime), foram desenvolvidas neste trabalho.

- movietime.c: Este programa feito em linguagem C tem por finalidade gerar um arquivo que informa o tempo necessário para tocar cada bloco de um determinado vídeo. Para isto o arquivo do vídeo é dividido em blocos cujo tamanho é o mesmo do usado no sistema RIO. Cada bloco é enviado ao *player* e contabiliza-se o tempo que foi necessário para a sua exibição. Como saída é gerada uma lista contendo o tempo de exibição de cada um dos blocos do vídeo. Esta lista, armazenada em um arquivo, é útil ao emulador do cliente de vídeos pois devido ao fato de ele não executar o *player*, não sabe quanto tempo

cada bloco leva para ser tocado[3]. No momento de emular a exibição de um determinado bloco, o emulador do cliente dorme o mesmo período de tempo que o cliente real levaria para exibir tal bloco, e o período que leva dormindo é obtido neste arquivo, chamado de *cactimes*.

- ExtractCactimes.sh: Alguns vídeos não puderam ser tratados pela ferramenta movietime por terem sido codificados com um padrão diferente do esperado pelo aplicativo, o que gerou incompatibilidade entre ambos. Para resolver este problema desenvolvemos um *script* que gera o arquivo de *cactimes* com base no tempo médio que cada bloco de um determinado vídeo leva para ser executado[4] um bloco e o tempo total que o vídeo leva para ser exibido, esta ferramenta utiliza o arquivo de XML (*eXtensible* **M***arkup* **L***anguage*) [W3C 2008] do vídeo em análise. Este arquivo contém várias informações sobre o objeto que são úteis ao *script*.

  Uma vez obtidos estes dados, a lista é gerada aplicando-se alguma variação neste tempo médio, para mais ou para menos, aleatoriamente[5]. Apesar de não representar a situação real, o arquivo de *cactimes* gerado desta forma mostrou ser tão eficiente quanto o real dadas as análises dos resultados finais.

- ExtractCederjLogs.sh: Este *script* converte o log de comportamento de um cliente [Botelho 2005], conforme figura 3.3 a ser explicada adiante, em um log de ações no formato de entrada do emulador de cliente de vídeo, apresentado na figura 3.1. Esta ferramenta trabalha de forma recursiva, percorrendo toda uma árvore de diretórios à procura dos logs de comportamento extraído dos

[3] Devido ao método de compactação utilizado no arquivo de vídeo, dois blocos distintos, independente de terem o mesmo tamanho, não levam o mesmo tempo para serem exibidos.

[4] A forma de codificação utilizada trabalha com um mecanismo de compactação no qual blocos de vídeo com um mesmo tamanho em *bytes* não levam, necessariamente, o mesmo tempo para serem exibidos. O tempo médio é obtido com o cálculo da média aritmética dos tempos de exibição de cada bloco de vídeo de um objeto.

[5] Os números aleatórios foram gerados pelo comando *rand* do *Linux*, e são uniformemente distribuídos dentro do intervalo $[0.95x, 1.05x]$, onde x é o tempo médio que um bloco de vídeo demora para ser exibido.

pólos do CEDERJ e pode converter, de uma só vez, todos os logs de um formato para outro;

- getlostfragments.sh: Este *script* procura, todos os logs dos clientes gerados no final da emulação e, para cada um deles, contabiliza e exibe o total de perdas ocorridas durante sua execução;

- getNetMachines.sh: Esta ferramenta faz uma varredura na rede local para tentar descobrir todas as máquinas que estão ligadas e conectadas à rede local. Para isto, o *script* envia um ping para todas as máquinas (via *broadcast*), coleta todas as respostas recebidas e, para cada máquina que respondeu ele verifica se ela está acessível via SSH (**S***ecure* **SH***ell*), verifica qual a sua distribuição *Linux* e informações úteis de *hardware*, como capacidade do(s) processador(es), total de memória RAM (**R***andom* **A***ccess* **M***emory*) instalada e espaço livre em disco;

- histogramPerdaPlot.sh: Este *script* analisa os logs de clientes gerados em uma emulação e, extraindo informações sobre as perdas experimentadas por cada cliente, gera um histograma cujo objetivo é informar quantos clientes experimentaram uma certa quantidade de perdas de fragmentos;

- loadPlotPL.sh: Esta ferramenta analisa informações de uso de memória e cpu do PL coletadas pelo *script* loadCheck.sh, previamente descrito, e gera um gráfico "consumo × tempo" para cada uma das duas análises;

- loadPlotRIO.sh: Semelhante à ferramenta anterior, esta ferramenta gera os mesmos gráficos para o *dispatcher* e para o *storage*;

- netrametPlot.sh: Esta ferramenta filtra as estatísticas coletadas pelo NeTraMeT e gera um gráfico de consumo de banda para os protocolos TCP e UDP;

- storagePlot.sh: Esta ferramenta analisa os logs gerados pelo *storage* em uma emulação e traça um gráfico de *bytes* enviados ao longo do tempo;

- streamsPlot.sh: Esta ferramenta analisa os logs gerados pelo *storage* e gera dois gráficos: um que informa o número de fluxos abertos ao longo do tempo e outro que indica a probabilidade de um certo número de fluxos estar aberto ao longo do tempo;

- plotAll.sh: Este *script* gera todos os gráficos descritos acima fazendo uso dos respectivos *scripts*. Passando os parâmetros coerentemente ele gera uma coleção destes gráficos agrupados em uma pasta;

- playPlot.sh: Esta ferramenta analisa os logs dos clientes extraindo informações de exibição dos blocos de vídeo e gera um gráfico que diz quantos blocos são exibidos ao longo do tempo. Este gráfico é útil para fins de testes pois nos diz quantos clientes estão em atividade (nem estão em pausa nem parados e nem finalizados).

3.2 Configuração das máquinas

Nas tabelas 3.1 e 3.2 estão listadas as *workstations* utilizadas neste trabalho bem como informações pertinentes sobre as mesmas: quantidade de memória instalada, capacidade do(s) processador(es) e o número máximo de clientes que conseguimos disparar sem que detectássemos perda local nos mesmos. Consideramos perda apenas uma quantidade igual ou superior a noventa fragmentos perdidos, não consecutivos, que equivalem, quantitativamente, a um bloco de vídeo como visto na seção 1.

O ambiente dos experimentos era composto em sua maioria por equipamentos com capacidade de transmissão de 1Gbps. Os testes iniciais mostraram que a utilização de equipamentos com capacidade de transmissão de apenas 100Mbps é inviável neste ambiente pois a diferença da velocidade da transmissão de uma interface Giga para a de 100Mbps causa muitas perdas nesta última.

A partir das tabelas 3.1 e 3.2 podemos perceber que em alguns casos temos

Máquina	Memória (KB)	Processador(es) (2)=core duo, (1)=single core	Máx. clientes
itapema	2074000	(2)Core 2 6300 1.86GHz	100
recreio	2075752	(2)Xeon 3050 2.13GHz	150
forte	2075200	(2)Pentium 4 3.00GHz	100
prainha	2065596	(2)Core 2 6400 2.13GHz	150
botafogo	1807220	(1)Pentium 4 3.00GHz	75
copa	2064632	(2)Pentium 4 3.40GHz	100
icarai	1035488	(1)Pentium 4 2.80GHz	100
armacao	2075432	(1)Pentium 4 3.00GHz	100
salinas	2064664	(2)Pentium 4 3.40GHz	75
paraty	514496	(1)Pentium III	20
meaipe	2074000	(1)Core 2 6300 1.86GHz	100
dunas	1033224	(1)Pentium 4 3.00GHz	125
pipa	2064656	(2)Pentium 4 3.80GHz	125
barra	2067356	(1)Pentium 4 2.80GHz	125
urca	2066420	(1)Pentium 4 3.60GHz	100

Tabela 3.1: Máquinas do LAND usadas nas emulações (parte 1/2).

Máquina	Memória (KB)	Processador(es) (2)=core duo, (1)=single core	Máx. clientes
ramos	1285320	(1)Pentium 4 2.80GHz	75
itaipu	2326472	(2)Pentium 4 3.20GHz	100
trindade	1026324	(2)Pentium 4 3.20GHz	150
flamengo	1033132	(1)Pentium 4 1.80GHz	75
pontal	1026016	(1)Pentium 4 2.40GHz	100
ipanema	1025080	(2)Pentium D 2.80GHz	125
grumari	766848	(2)Pentium 4 3.20GHz	50
torres	2066420	(1)Pentium 4 3.60GHz	100
arpoador	2075752	(2)Xeon 3050 2.13GHz	150
dhcp36	2075752	(2)Xeon 3050 2.13GHz	150
dhcp37	2075752	(2)Xeon 3050 2.13GHz	150
dhcp42	2075752	(2)Xeon 3050 2.13GHz	150
dhcp45	2075752	(2)Xeon 3050 2.13GHz	150
itapuan	1027260	(2)Pentium 4 3.20GHz	125

Tabela 3.2: Máquinas do LAND usadas nas emulações (parte 2/2).

máquinas com configuração muito semelhante mas com desempenho bem diferente. Temos até mesmo máquinas menos potentes com resultados melhores que máquinas mais potentes. Este resultado se deve às diferenças de uso das máquinas bem como os processos que cada uma estava executando durante os testes[6]. Para evitar problemas em uma fase adiante, fizemos os testes três vezes para cada máquina e ficamos com o pior resultado de cada uma.

Nas máquinas mais potentes, como a arpoador e as dhcpXX, o limitante não foi nem o processador, nem a memória e nem a interface de rede. Todos estes estavam sendo subutilizados de acordo com nossas monitorações. Acreditamos que as perdas foram fruto de gargalo do SO. Este argumento está baseado no fato de que, para uma mesma máquina (pontal), fizemos testes exaustivos com duas distribuições: *Scientific Linux* IV e Ubuntu 7.10. Na segunda distribuição o desempenho foi substancialmente melhor. Conseguimos disparar apenas vinte e cinco clientes na primeira e setenta e cinco na segunda. É válido, ainda, comentar que o teste com Ubuntu nesta máquina foi feito com Live CD; já o *Scientific Linux* estava instalado e em uso. Era esperado que os testes com Ubuntu instalado tivesse desempenho ainda melhor comparado ao teste com o *Live* CD. Esta expectativa foi confirmada: aumentamos de setenta e cinco para 100 a quantidade de clientes disparados simultaneamente. Todas as máquinas das tabelas 3.1 e 3.2 estavam com a distribuição Ubuntu 7.10 instaladas na execução deste trabalho.

Outro fator que limita o número de clientes por máquina é um ambiente com muitas perdas. Quando um cliente perde fragmentos ele chama o método *FreeBlock* que libera os fragmentos que chegaram do bloco a ser exibido, conforme apresentado na seção 2.3.4. Este método envolve muitas chamadas de sistema e é muito custoso ao processador. Quando o cliente está em um ambiente de perdas este método é constantemente chamado, o que aumenta consideravelmente o consumo de CPU por cliente, diminuindo a quantidade de clientes que podem ser disparados em uma

[6]O simples uso do *browser* Mozilla-Firefox consome tanta CPU da máquina que prejudica consideravelmente os resultados.

mesma máquina.

Para conectar as máquinas usamos quatro *switches* DELL, dos quais dois, gerenciáveis, implementam corretamente a técnica IP *multicast*. Os outros dois, não gerenciáveis, a implementam com o *broadcast*, isto é, quando chega um bloco cujo destino é um grupo *multicast* então estes *switches* aguardam o recebimento completo do bloco e o replicam em todas as portas. Este é um dos fatores que ocasionou as perdas no cenário *multicast*, como veremos mais adiante no capítulo de experimentos e resultados.

Os *switches* gerenciáveis são o "Dell PowerConnect 2708" e o "Dell PowerConnect 2724" e os não gerenciáveis são o "Dell PowerConnect 2616" [Inc. 2006]. Todos estes equipamentos operam a 1Gbps e estão conectados de forma que o número de saltos não ultrapasse uma unidade no cascateamento dos mesmos. Esta medida foi tomada pois percebemos em nossos testes preliminares que quando estes saltos são mais de um as perdas aumentam consideravelmente: em um teste com duração de cinco minutos feito entre duas máquinas com um salto e com dois saltos tivemos, respectivamente, perda de 80 e 1200 fragmentos, em média, por cliente.

Para determinar a quantidade máxima de clientes a serem disparados em uma máquina foi considerado o seguinte critério: não queríamos que as perdas detectadas nos clientes tivessem a contribuição da máquina local, isto é, queríamos ter certeza de que todas as perdas ocorreram por fatores exclusivamente externos à máquina que executa o cliente. Isto impede que a comparação entre as técnicas de transmissão *unicast*, PI e PIE seja injusta com uma delas. Se a máquina local for um fator que contribui para a perda então a comparação entre as técnicas de transmissão estará prejudicada pois a técnica em si não será a única a influenciar no desempenho dos clientes.

Fizemos emulações usando apenas uma máquina com os clientes e mantivemos a estrutura do servidor RIO e do PL inalteradas. Começamos os testes disparando $x = 30$ clientes na máquina a ser experimentada e fomos aumentando este número

em incrementos de 10 ($x = x + 10$) até que as perdas começassem a surgir nos clientes. Quando isto ocorria, parávamos o experimento e tomávamos o x da rodada anterior ($x - 10$) como o valor ideal para o máximo de clientes a serem disparados naquela máquina.

Trabalhamos, ainda, com uma certa tolerância a perdas: só consideramos o evento perda como real quando perdia-se o equivalente a um bloco de vídeo (equivalente a quase um segundo de vídeo[7]), logo, enquanto o cliente não perdia noventa fragmentos não considerávamos os fragmentos faltantes como perda (para uma sessão com duração de apenas cinco minutos, por exemplo, um segundo de vídeo é o equivalente a 0.3333% do tempo total e por isto uma perda desta magnitude foi considerada desprezível).

3.3 O NeTraMeT: A ferramenta usada para coleta de tráfego

O NeTraMeT (*Network Traffic Flow Measurement Tool*) [Group 2008b] é uma ferramenta de medição de tráfego de pacotes de dados na rede implementado pela universidade de *Auckland*. A ferramenta possui código-aberto e de acordo com as especificações da arquitetura definida pelo grupo de trabalho da IETF (**I***nternet* **E***ngineering* **T***ask* **F***orce*), o RTFM (**R***ealtime* **T***raffic* **F***low* **M***easurement*).

As ferramentas que compõem o NeTraMeT são: medidores (NeTraMeT e NetFlowMet), leitor de medidores/gerenciador (NeMaC e nifty), compilador e alguns utilitários (fd_filter e fd_extract) que tratam os arquivos que contém as informações obtidas sobre os fluxos.

Escolheu-se esta ferramenta para a análise dos fluxos por diversos motivos [Group 2008b], entre eles:

[7] Cada bloco de vídeo leva entre 95 e 98 centésimos de segundo, em média, para ser consumido.

1. O NeTraMeT implementa o padrão sugerido pelo IETF;
2. Pode ser instalado em um PC, não sendo necessário um equipamento à parte;
3. Permite fazer estudos em um ambiente controlado de rede;
4. É um programa de código aberto sob a licença GPL;
5. Tem sido usado em outros trabalhos e seus resultados têm se mostrado satisfatórios.

No NeTraMeT especificamos, através de um arquivo de configuração que é compilado para uma linguagem própria da ferramenta, os tipos de fluxos que queremos analisar (se TCP ou UDP), os fluxos de quais máquinas queremos analisar (isto é feito analisando-se o endereço IP da máquina de origem e/ou de destino), dentre outras opções. Estas duas são as que se aplicam ao nosso caso, pois queremos analisar os fluxos referentes à troca de mensagens entre as máquinas (esta troca é feita com TCP) e o fluxo de vídeos transmitidos (estes fluxos usam o protocolo com UDP).

Uma dificuldade que tivemos deve-se ao fato de que a máquina que roda o NeTraMeT está conectada a um *switch* de nosso laboratório. Como este equipamento isola o tráfego entre duas máquinas das demais, seria impossível capturar o fluxo da rede entre o servidor e cada uma das máquinas clientes pois tal fluxo não seria transmitido à porta à qual a máquina que executa o NeTraMeT está conectada. A solução para isto foi espelhar as portas onde se encontram o *dispatcher*, os *storages* e o PL na porta onde se encontra o NeTraMeT. Desta forma, tudo o que chega às interfaces de rede das máquinas cujas portas foram espelhadas é perceptível à máquina que roda o NeTraMeT, o que viabiliza a análise do tráfego.

Note que não é necessário fazer o espelhamento das portas das máquinas clientes na porta da máquina do NeTraMeT pois os clientes não trocam mensagem entre si. Todo o fluxo, seja TCP ou UDP, de vídeo ou de troca de mensagens, necessariamente passa pela máquina do PL, do *dispatcher* e/ou do *storage*.

3.4 A topologia do ambiente de emulação

As emulações foram feitas em uma rede local cuja capacidade de transmissão é de 1Gbps. Nos experimentos iniciais tentamos utilizar alguns clientes com interfaces de rede a 100Mbps, mas ao analisarmos o desempenho dos mesmos percebemos que as perdas foram altíssimas. Isto se explica pelo fato de que muitos dados chegam à estas interfaces com uma taxa muito maior do que elas podem receber; com isto os dados ainda não recebidos são armazenados no *buffer* da placa de rede. Como este *buffer* é extremamente pequeno, há um *overflow* e os dados que não encontram espaço no *buffer* são descartados. Este problema acontece com tamanha intensidade que os clientes nesta situação começam a deixar o sistema por *timeout*, isto é, ficam esperando blocos que nunca chegam e acabam sendo finalizados por excesso no tempo de espera.

As máquinas estão conectadas entre si por um *switch* usado exclusivamente pelos equipamentos envolvidos nos experimentos, isto é, ficaram a ele conectadas apenas 5 máquinas, onde uma executava o servidor RIO (*dispatcher* + *storage*), outra executava o módulo PL, outra executava o NeTraMeT e as outras duas executavam, cada uma, 125 clientes. Esta topologia, apresentada na figura 3.2, mostrou-se a ideal para nossos experimentos por isolar os fluxos de dados de nossos experimentos dos demais fluxos de nossa rede local.

3.5 Os logs dos usuários

Para a confecção deste trabalho três tarefas foram fundamentais: a preparação do ambiente de emulação, a implementação da técnica PIE no cliente e no PL e a obtenção da carga de logs de interações a ser usada nos clientes durante a emulação. As duas primeiras tarefas já foram apresentadas neste texto. Agora apresentaremos a terceira tarefa.

Os logs podem ser obtidos de duas formas: (1) através de logs reais, obtidos

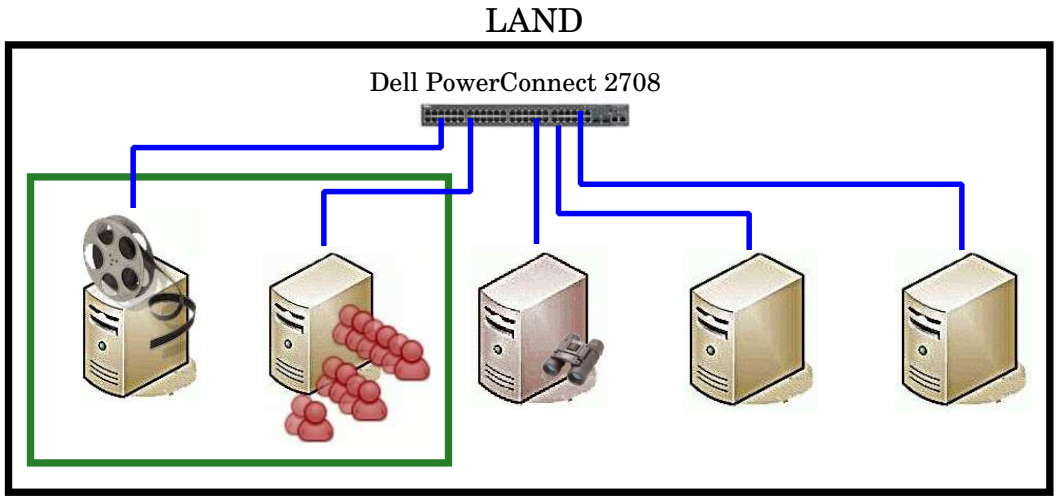


Figura 3.2: Topologia do ambiente de experimentos.

na execução dos clientes pelos alunos do CEDERJ e (2) através de logs gerados artificialmente.

3.5.1 Carga real e carga sintética

O cliente do servidor RIO armazena em um arquivo todas as interações realizadas pelo usuário assim como outras informações de configuração do mesmo. No final da execução do cliente este arquivo é enviado ao servidor. Com isto, quanto maior o número de usuários acessando os vídeos disponibilizados, maior será a quantidade de logs disponíveis para serem usados e analisados posteriormente. Um exemplo de um destes logs coletados no cliente é apresentado na figura 3.3. Note que algumas informações sobre a situação em que o usuário se encontrava e os parâmetros de configuração do cliente que estavam sendo usados também constam neste log. Estas informações existem pois podem ser necessárias dependendo do estudo que se quer fazer sobre o comportamento de um ou mais usuários do sistema RIO[Botelho 2005].

Os logs sintéticos são obtidos através de um gerador de carga sintética [Vielmond 2007, Carolina C. L. B. de Vielmond]. Este gerador se baseia nos logs reais para gerar os logs sintéticos. Em síntese, este gerador estuda o comportamento

```
Log date: Sat Aug 20 11:09:42 2005
User login: a20051305060
Object path: /cederj/sistemas_comp/ead05002/Aula_010.mpg
Block size: 131072
Client IP Adrress: 127.0.0.1
Connected: Server
Index status: success
--------------------Config file-------------------
[riommclient]
RioMMClient/Buffers=2
RioMMClient/ConnectPI=NULL
RioMMClient/MainWindow=0
RioMMClient/PIHost=NULL
RioMMClient/PluginPort=0
RioMMClient/Sync=false
RioMMClient/TgifWindow=0
RioMMClient/UseCache=false
RioMMClient/UsePlugins=true
RioMMClient/User=guest
RioMMClient/Version=1.1
RioMMClient/VideoWindow=0
RioMMVideo/MPlayer/arguments=-quiet -vfm ffmpeg -cache 128 -
                    .
                    .
                    .
RioMMVideo/makeLog=true
------------------Actions log--------------------
1124546976297664 bplay 0
1124546976318476 reqserv 0
1124546976319172 reqserv 1
1124546976340216 arrives 0
1124546976709547 arrives 1
1124546976712357 plays 0 131072
1124546976868718 reqserv 2
1124546976871323 plays 1 131072
1124546976924050 jmptop 239
1124546976936374 reqserv 239
1124546976936562 reqserv 240
1124546977216025 arrives 240
1124546977290602 reqserv 241
1124546977711407 arrives 241
1124546977715975 plays 240 131072
1124546978107063 arrives 240
1124546978184541 reqserv 242
1124546978620047 arrives 241
1124546979108947 arrives 242
1124546979117063 plays 241 131072
1124546979416170 reqserv 243
1124546979520867 arrives 243
1124546979522920 plays 242 131072
1124546980253198 reqserv 244
1124546980290764 arrives 244
1124546980293831 plays 243 131072
1124546980971196 reqserv 245
1124546980998669 arrives 245
1124546981001695 quit
```

Figura 3.3: Log de comportamento de um cliente.

dos usuários acessando um servidor de ensino a distância em operação criando um modelo para a geração da carga sintética. O modelo do gerador usado baseia-se na abordagem clássica de modelos de Markov ocultos (*hidden Markov models* - HMM), onde restringe-se a distribuição das observações dentro de um estado oculto. Por não ser o objetivo deste estudo, o gerador não será detalhado aqui, mas pode ser melhor compreendido em [Carolina C. L. B. de Vielmond].

Entre a carga real e a carga sintética optamos pela sintética pois não temos logs reais suficientes para emularmos o número alto de clientes que queremos emular; com o gerador podemos gerar quantos logs forem necessários. Por outro lado, confiamos que a carga sintética representa satisfatoriamente o comportamento dos usuários reais pois em estudos anteriores [Valle 2007, Carolina C. L. B. de Vielmond] compararam-se os resultados utilizando-se uma e outra, donde se concluiu que o uso da carga sintética deste gerador traz resultados semelhantes ao uso da carga real.

3.5.2 Classificação dos logs

De forma a podermos avaliar as técnicas para clientes com características distintas de interatividade separamos os logs em três categorias: baixa, média e alta interatividade. Com isto, faremos as emulações para cada um destes grupos e, no momento de compararmos as políticas PI, PIE e *unicast* entre si, evitaremos injustiças.

Quanto a métrica usada para a classificação dos logs nos baseamos no trabalho de [Rocha et al. 2005, Almeida et al. 2001, Costa et al. 2004]. A primeira métrica que consideramos foi o número de interações de um usuário durante uma sessão, no entanto esta não parecer ser uma métrica adequada. Suponha o seguinte cenário: considerem um log com duas horas de duração e vinte interações no total e outro de dez minutos e cinco interações. Embora o primeiro log tenha o quádruplo de interações do segundo, o intervalo entre interações do primeiro é três vezes maior do que o intervalo do segundo. Isto significa que o primeiro usuário é menos interativo. Por esta razão, o fator decisivo na classificação de um log não foi a quantidade de

interações, mas o tempo entre interações.

Decidido isto, separamos os logs reais nestas três categorias e parametrizamos o modelo para cada grupo de logs, gerando três cargas sintéticas distintas.

Na fase de planejamento, algumas metas foram traçadas: emular por volta de 2000 clientes com sessões que durasse ao menos trinta minutos. Uma vez estabelecidas estas metas não podíamos simplesmente utilizar a carga real por dois motivos:

1. Precisávamos de pelo menos 2000 logs já que esta era a meta inicial de clientes a serem emulados;
2. Precisávamos de logs com duração mínima de trinta minutos pois este foi considerado um tempo razoável para obtermos os resultados confiáveis. Para tempos muito curtos de emulações poderíamos ter resultados que refletissem o comportamento transiente do sistema.

Atualmente temos disponível um total de 7664 logs reais, os quais divididos em categorias nos dá 2554 para cada classe. Destes logs somente 2878 são maiores que 30 minutos o que nos dá em torno de 960 logs para cada categoria. Portanto optamos pelo uso de logs sintéticos.

Outras duas categorias de logs foram gerados: logs seqüenciais e logs com altíssima interatividade. Estes logs representam os dois extremos de nível de interatividade que um cliente pode ter. Nos logs seqüenciais os clientes iniciam a exibição do vídeo e não interagem em momento algum até que o vídeo termine. Nos logs de altíssima interatividade os clientes executam saltos a cada vinte segundos e nunca interagem com pausas ou suspensão da execução do vídeo.

3.6 Metodologia de Experimentos

A motivação deste conjunto de experimentos é avaliar o desempenho do sistema RIO com a técnica PIE e comparar os resultados do uso desta técnica com a técnica PI

e a de transmissão *unicast*.

Analisamos diferentes cenários para evitar injustiça nas comparações de desempenho das técnicas em estudo. Desta forma poderemos determinar qual técnica é mais adequada a um cenário específico. Isto se torna necessário pois elimina dúvidas tais como: "em um cenário onde os clientes têm um elevado grau de interatividade a técnica PI ou PIE não melhora o desempenho do servidor e, portanto, deve-se utilizar a transmissão *unicast*." Esta é apenas uma das muitas dúvidas que surgem antes mesmo da implementação da técnica. Estas e outras eventuais situações serão avaliadas e terão seus resultados apresentados no capítulo 4.

Para cada cenário emularemos tantos clientes quantos forem possíveis de serem disparados. De acordo com as tabelas 3.1 e 3.2 temos um limite máximo de 3235 clientes a serem disparados. Estamos limitados a este número por não termos mais máquinas disponíveis para uso em nossas emulações e por estarmos exaurindo a capacidade de cada máquina rodando o número máximo de clientes sem que haja perda local. Este limite, no entanto, não nos preocupa pois antes de conseguirmos disparar esta grande quantidade de clientes esbarraremos no recurso de banda, pois visto que cada cliente consome entre 1.1 e 1.3 Mbps, o canal de 1Gbps permite entre 768 e 909 fluxos ativos simultaneamente. Na prática não conseguimos ter mais de 600 fluxos ativos. Não conseguimos chegar ao intervalo em questão pois estes números pressupõem um ambiente utópico onde utilizaríamos 100% do recurso de banda; deixando de considerar eventos comuns que causam desperdício de banda como colisões, perdas ou *overflow* de *buffers*[Kurose e Ross 2000].

3.6.1 Os cenários testados

Para avaliação da técnica montamos diversos cenários nos quais variamos o grau de interatividade do usuário e o uso ou não de *cache* (*buffer* em disco).

Os graus de interatividade estão descritos a seguir:

Seqüencial: Neste cenário os clientes iniciam a exibição do vídeo a partir do seu primeiro bloco e sem fazer interação alguma assistem o vídeo do início ao fim;

Baixa interatividade, média interatividade e alta interatividade: Nestes cenários utilizamos os logs sintéticos obtidos como descrito na seção 3.5.2. Para o cenário de alta interatividade o tempo médio entre duas interações nestes logs varia de 0 a 501 segundos; já para o cenário de média interatividade este valor fica entre 502 e 1594. Finalmente, para o cenário de baixa interatividade este tempo médio assume valores entre 1595 e 4762. Nota-se, que o cenário com baixa interatividade tem alguns clientes cujo tempo médio entre interações é mais dez vezes maior que o tempo médio entre interações de alguns clientes do cenário de alta interatividade;

Altíssima interatividade: Neste cenário os clientes estão em constante interação. Todos os clientes iniciam a exibição do vídeo a partir do primeiro bloco (zero) e interagem a cada vinte segundos com saltos aleatórios[8] para uma outra parte do vídeo. Suas interações nunca são do tipo *pause* ou *stop*.

Em uma fase mais adiantada do trabalho nos deparamos com uma limitação nova que surgiu com a substituição do sistema operacional das máquinas do LAND: o PL não conseguia gerenciar mais de 250 clientes pois o *kernel* do SO utilizado não permitia que um processo manipulasse mais de 1024 *sockets* na função de escuta destes sockets, o *select()*. Como para o gerenciamento de cada cliente o PL faz uso de quatro *sockets*[9] e precisa de mais 12 sockets para outros fins, não temos como disparar mais de 250 clientes[10]. Devido a esta limitação todos os nossos experimentos

[8]Os números aleatórios foram gerados pelo comando *rand* do *Linux*, e são uniformemente distribuídos dentro do intervalo especificado.

[9]Quando um cliente se conecta ao PL, este cria quatro *sockets* que são utilizados para as trocas de mensagem do cliente com o PL e do PL com o *dispatcher*, sendo que nos dois casos abre-se um *socket* para troca de mensagens e outro para envio de dados.

[10]A solução para este problema seria recompilarmos o *kernel* do sistema operacional ou reestruturarmos o gerenciamento de *sockets* no PL usando uma função alternativa ao *select()* (a função *poll()* seria esta alternativa), mas como a troca do SO no LAND foi próxima à conclusão deste

foram feitos com 250 clientes.

Cada um dos cenários descritos foi emulado seis vezes e seus resultados comparados entre si para termos certeza de que o comportamento das emulações de cada um deles era consistente. Nesta comparação verificamos a semelhança no consumo de banda média e de perdas percebidas pelos clientes, pois acreditamos que estas duas métricas representam satisfatoriamente o desempenho do sistema nas duas extremidades: cliente e servidor.

Combinando-se os cinco graus de interatividade à utilização ou não da *cache*, temos dez cenários para cada uma das três técnicas PI, PIE e transmissão *unicast*, o que nos dá trinta cenários distintos. Como cada um destes cenários foi emulado seis vezes, totalizamos cento e oitenta emulações. Cada emulação dura, em média, 40 minutos, onde 30 minutos são de emulação e 10 minutos de procedimentos pré e pós emulação, como preparação e verificação do ambiente e coleta dos logs nas máquinas envolvidas na emulação e geração de relatório dos mesmos. No total foram 120 horas de emulações.

Quanto aos parâmetros utilizados no PL, nos valemos do estudos feitos sobre valores ideais para δ_{after}, δ_{before} e δ_{merge} em [Netto 2004] e [Rodrigues 2006] onde se analisou diferentes cenários variando-se os valores destas janelas. No primeiro destes estudos conlui-se que a janela δ_{after} não deve ser muito curta pois isto diminuiria muito o compartilhamento de fluxos. Por outro lado esta janela também não pode ser muito longa pois haveria o agrupamento de clientes muito distantes do grupo e isto acarretaria em *patches* muito longos. Quanto ao valor de δ_{before}, o critério principal a ser levado em conta é a tolerância do cliente em receber blocos de vídeo que não solicitou. Por último, nos apoiamos na conclusão de [Rodrigues 2006] de que δ_{merge} pode ser aproximado pelo valor de δ_{after}. Para estes parâmetros usamos os seguintes valores:

trabalho não houve tempo hábil para aplicarmos uma destas soluções dada a complexidade das mesmas.

- $\delta_{\text{after}} = 150$ blocos de vídeo
- $\delta_{\text{before}} = 25$ blocos de vídeo
- $\delta_{\text{merge}} = \delta_{\text{after}} = 150$ blocos de vídeo

O *storage* utilizado nos experimentos era formado de dois discos em arquivo.

Obs:	O *storage* foi implementado de forma a poder utilizar um disco físico ou de simular o disco com o uso de arquivo. Testes iniciais de ambiente mostraram que o sistema se comportou tão bem no uso de arquivos quanto no uso de disco físico. Como a utilização de um disco em arquivo é muito mais simples e facilita migrar o servidor de uma máquina para outra, optamos pelo disco em arquivo.

No *dispatcher* utilizamos um *buffer* de tamanho nove para cada cliente e conexão com apenas um *storage*[11]. Os testes na fase de definição do ambiente dos experimentos nos mostraram que, para o nosso caso, não faz diferença usar um único *storage* ou múltiplos.

3.6.2 As medidas coletadas

As medidas coletadas concentraram-se em diversos elementos do sistema, a saber:

- Medidas coletadas no cliente: No lado do cliente coletamos as informações relativas às perdas que o cliente experimentou. Para cada bloco de dados enviado pelo *storage*, o cliente analisa para ver se tem os noventa fragmentos e caso não tenha informa no log a quantidade de fragmentos perdidos para aquele bloco.

  Não faremos distinção de dados perdidos e dados atrasados. Toda informação que não estiver disponível ao cliente no momento em que ele precisar dela será considerada perda.

[11] Na seção 2.3.1 vimos que o *dispatcher* pode se conectar a um ou mais *storage servers*

Alguns blocos poderão ser recebidos mas não tocados. Estes também entrarão no cômputo de blocos perdidos pois ainda que o cliente não precise mais dele, já precisou em algum momento e o requisitou ou o recebeu do fluxo *multicast* de seu grupo.

- Medidas coletadas no servidor RIO:

  No *dispatcher*: No *dispatcher* coletaremos informações de uso dos recursos da máquina (memória e CPU).

  No *storage*: No *storage* coletaremos informações de uso dos recursos da máquina (memória e CPU), dos dados enviados aos clientes e do número de fluxos abertos ao longo de cada emulação.

- Medidas coletadas pelo NeTraMeT:
  No NeTraMeT foram coletados os dados transmitidos pelo conjunto do servidor RIO (*dispatcher*, *storage* e PL) tanto para os dados TCP quanto UDP.

Capítulo 4

Resultados Experimentais

Neste capítulo descrevemos os principais experimentos realizados e fazemos uma análise sobre cada resultado obtido. Começamos apresentando os resultados obtidos nos cenários testados e terminamos comparando o desempenho de cada um deles.

Para facilitar comparações entre gráficos inserimos um código que, sucintamente, referencia o cenário ao qual o gráfico descreve, a saber:

SEQ: Emulação onde os clientes acessam o vídeo de forma seqüencial;

LOW: Emulação onde o grau de interatividade dos clientes é baixo;

MED: Emulação onde o grau de interatividade dos clientes é médio;

HIGH: Emulação onde o grau de interatividade dos clientes é alto;

STRESS: Emulação onde o grau de interatividade dos clientes é altíssimo, com saltos a cada vinte segundos e nenhuma interação de parada (*pause* ou *stop*) deste o início até a finalização da emulação;

PI: Emulação onde os clientes compartilhavam fluxos através da transmissão *multicast* do servidor RIO utilizando a técnica PI (*patching* interativo);

PIE: Emulação onde os clientes compartilhavam fluxos através da transmissão *multicast* do servidor RIO utilizando a técnica PIE (*patching* interativo eficiente);

UC: Emulação onde cada cliente é atendido exclusivamente por seu fluxo *unicast*;

BUF: Emulação onde os clientes armazenam em disco os blocos recebidos para uso posterior;

NOB: Emulação onde os clientes descartam os blocos após sua exibição.

Para compararmos as técnicas faremos uso de gráficos, dentre os quais o mais comum é o de número de fluxos ativos no servidor. Cabe aqui dizer que um fluxo é considerado ativo no instante t se o cliente associado a ele está fazendo requisição de blocos neste mesmo instante. O fluxo de um cliente *multicast* passivo e que não esteja fazendo *patching* não é considerado ativo apesar de ainda estar aberto e pronto para enviar e receber dados.

Indicaremos nos gráficos de número de fluxos ativos e nos de probabilidade do número de fluxos ativos a média destes valores por uma linha pontilhada e mais espessa que a do gráfico em questão. Esta métrica será usada nas comparações das técnicas PI, PIE e *unicast* nos diversos cenários sob análise e foi calculada da seguindo a equação 4.1, onde x_i é a quantidade de fluxos abertos no instante i do experimento e n é o tempo total deste mesmo experimento. Para calcularmos o valor de cada x_i, verificamos, para cada instante de tempo, quantos clientes estavam fazendo requisições de blocos ao servidor.

$$\sum_{i=1}^{n} \frac{x_i}{n} \tag{4.1}$$

4.1 Experimentos

4.1.1 Clientes com acesso seqüencial

Iniciamos nossa análise com clientes que fazem acesso exclusivamente seqüencial, ditos clientes com grau nulo de interatividade, onde as únicas interações que ocorrem são a inicial (*play*) e a final (*stop*).

Quanto à compação das técnicas de PIE e PI entre si, era esperado, neste cenário, que ambas tivessem resultados muito semelhantes com pouca ou nenhuma vantagem para a técnica PIE. Declaramos ser este o esperado pois a maioria dos clientes tendem a entrar em um único grupo, o grupo criado para o primeiro cliente que entrou no sistema. Como a quantidade de grupos neste cenário foi ínfima, a fusão de grupos não trouxe melhorias sensíveis, embora não possamos dizer que elas não existiram[1]. De fato isto é comprovado pelas figuras 4.2 e 4.3, onde mostra-se a quantidade de *bytes* enviados pelo *storage* ao longo do tempo.

Vale, ainda, comentarmos um detalhe sobre a igualdade dos resultados da PI e PIE neste cenário. Nos experimentos em questão, todos os clientes requisitam o vídeo desde o seu primeiro bloco. Com isto, novos grupos não podem se juntar a grupos já existentes, pois todos os grupos no sistema estão, necessariamente, em um estágio mais avançado e, portanto, nunca estarão dentro da janela δ_{merge} de um grupo novo. Para entender melhor, considere a seguinte situação: um novo cliente c entra no sistema. Se houver um grupo G tal que c esteja dentro de seu δ_{before} ou δ_{after}, então c será membro de G e não será criado um novo grupo. Caso contrário, será criado o novo grupo N. Neste momento o PL verifica se existe algum grupo atrás de N a uma distância máxima de δ_{merge} para ser agrupado a N tornando-se subgrupo do mesmo. Como o fato de c não ser agrupado em G significa que ele está a uma distância de G maior que δ_{after}, então isto implica que G está distante do

[1]No gráfico da figura 4.1 vimos, após análise minuciosa dos logs dos clientes e do PL, que realmente não tivemos nenhuma fusão de grupos, o que justifica o fato de seu resultado ter sido exatamente igual ao da técnica PI da figura 4.2.

bloco inicial a uma distância superior a δ_{after} e, conseqüentemente, a uma distância superior a δ_{merge}, pois $\delta_{\text{after}} = \delta_{\text{merge}}$. Com isto nenhum grupo pode ser agrupado a G pois novos grupos sempre iniciam as requisições a partir do primeiro bloco.

Já quanto a compararmos as técnicas PI e PIE com a transmissão *unicast*, esperamos, dado o comportamento seqüencial dos clientes, que o compartilhamento de banda seja maximizado e que a quantidade de dados enviadas pelo *storage* seja consideravelmente reduzida. Esta tese fica comprovada ao compararmos os gráficos das figuras 4.1, 4.2 e 4.3.

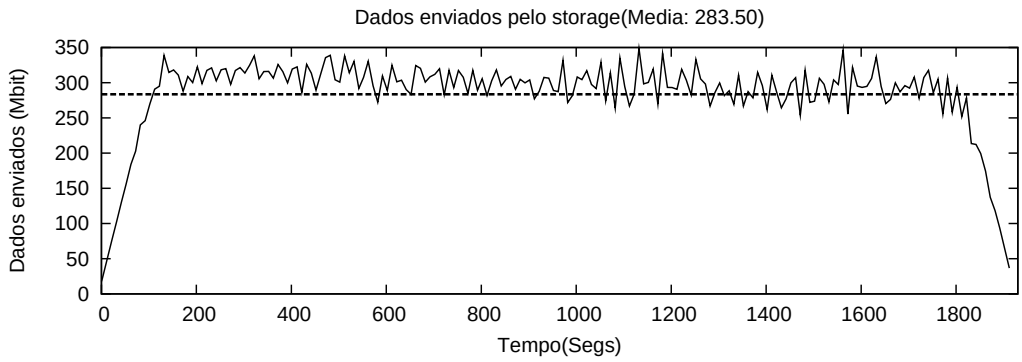


Figura 4.1: Dados enviados pelo *storage* (SEQ-UC-BUF).

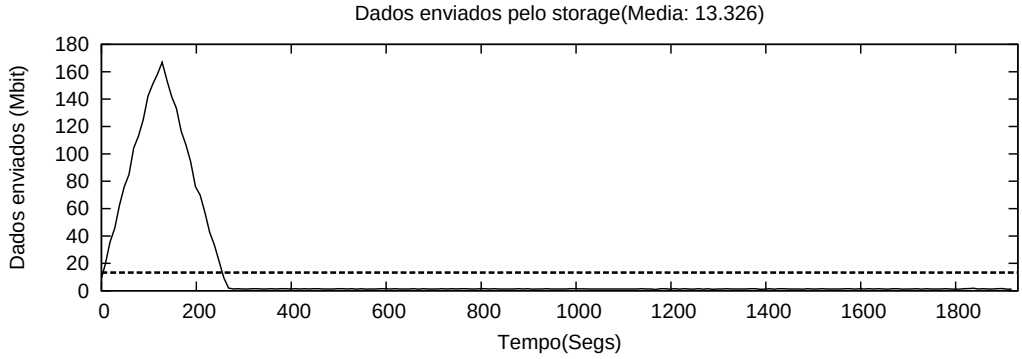


Figura 4.2: Dados enviados pelo *storage* (SEQ-PI-BUF).

Um gráfico também interessante de se analisar é o consumo de CPU pelo *storage* nos dois cenários (*unicast* e *multicast*). Vimos nas figuras 4.2 e 4.3 que após o centésimo segundo temos, aparentemente, poucos clientes no ambiente *multicast*, o que contrasta substancialmente com a quantidade de clientes no caso *unicast*. Nas

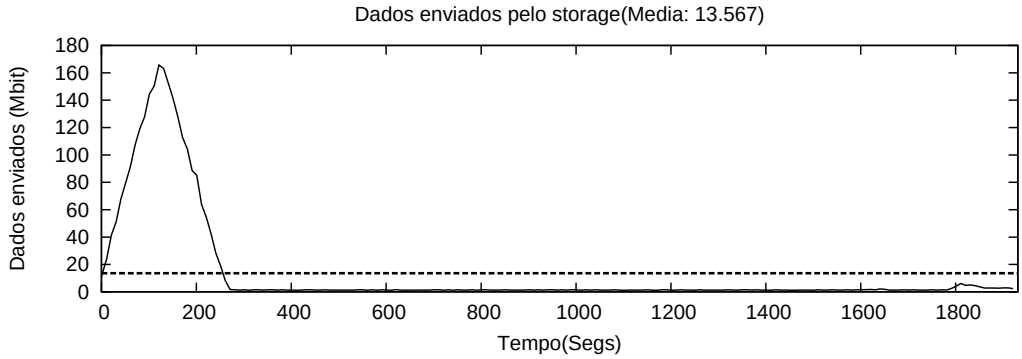


Figura 4.3: Dados enviados pelo *storage* (SEQ-PIE-BUF).

figuras 4.4, 4.5 e 4.6, vê-se o reflexo deste resultado no consumo do processador da máquina que executa o *storage*. Pelo gráfico das figuras 4.2 e 4.3 tínhamos a expectativa de que o gráfico de utilização da CPU pelo *storage* acompanhasse a curva destas figuras, e isto de fato ocorreu.

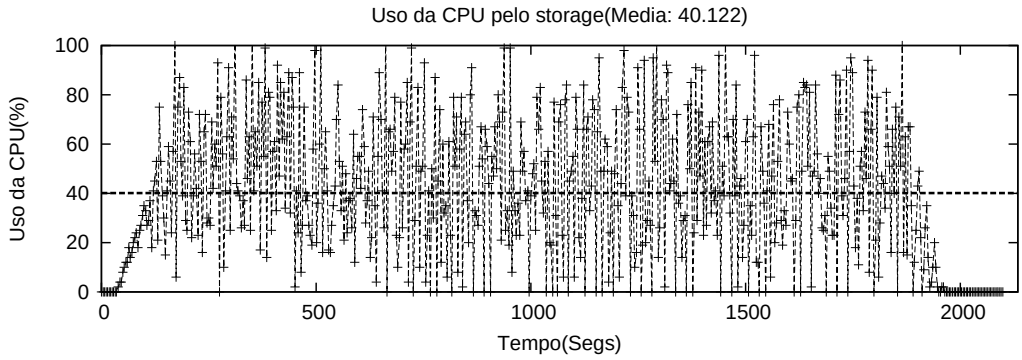


Figura 4.4: Consumo do processador pelo *storage* (SEQ-UC-BUF).

4.1.2 Clientes com baixa interatividade

Este cenário é bem mais interessante que o anterior pois representa um terço dos usuários típicos do CEDERJ[2], em especial o grupo que faz menos interações ao longo da execução do cliente de vídeo.

[2]Considerando que ao gerar a carga sintética fizemos uso de todos os logs gerados no CEDERJ até então e que cada um terço destes logs foi usado para gerar os logs sintéticos de baixa, média e alta interatividade respectivamente.

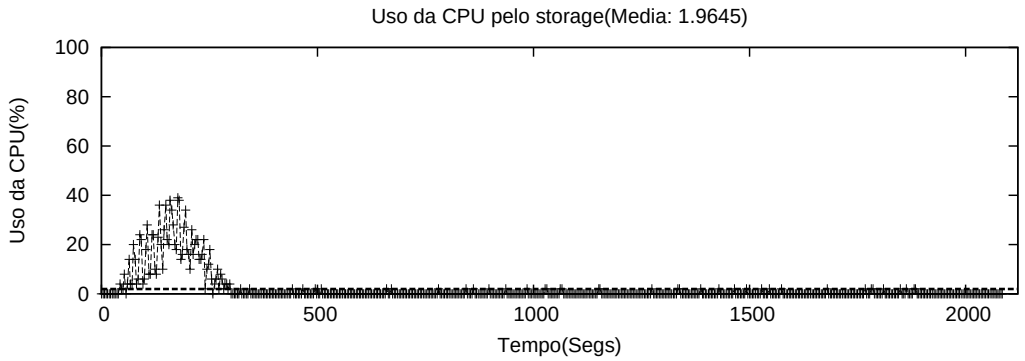


Figura 4.5: Consumo do processador pelo *storage* (SEQ-PI-BUF).

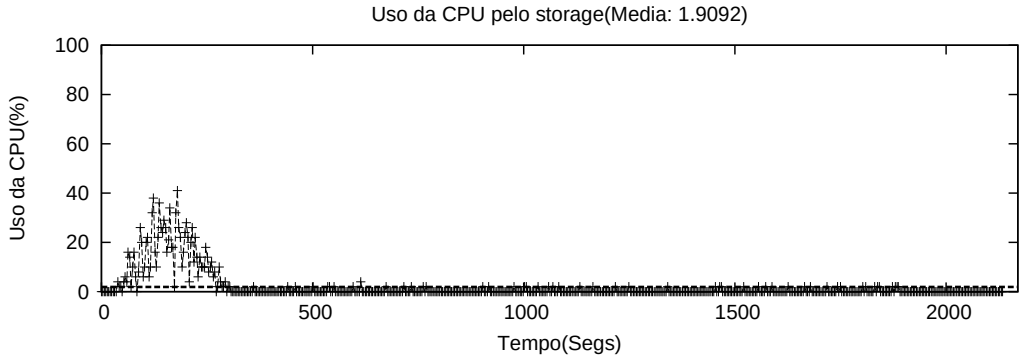


Figura 4.6: Consumo do processador pelo *storage* (SEQ-PIE-BUF).

Nota-se nas figuras 4.7 e 4.8 que a economia de banda, ainda muito valiosa e perceptível, cai consideravelmente quando comparada ao cenário anterior. Isto se explica pelo fato de os clientes interagirem para pontos distintos no vídeo. O agrupamento de fluxos, conforme descrito na página 17, acontece ou não dependendo da distância entre os mesmos. Quando a distância é maior que um certo valor, cria-se um novo grupo e, conseqüentemente, um novo fluxo é aberto, o que explica um maior número de requisições chegando ao *dispatcher* e de dados enviados pelo *storage*.

Um resultado que pode parecer diferente do esperado é estarmos com 250 clientes em execução e, conforme a figura 4.7, termos em média 74 fluxos abertos(cada um representando um cliente por se tratar de transmissão *unicast*.). Ao vermos este gráfico fomos averigüar a causa e a apontamos como conseqüência da alta incidência de longas pausas nos clientes que compõem este grupo. Diversas pausas excedem

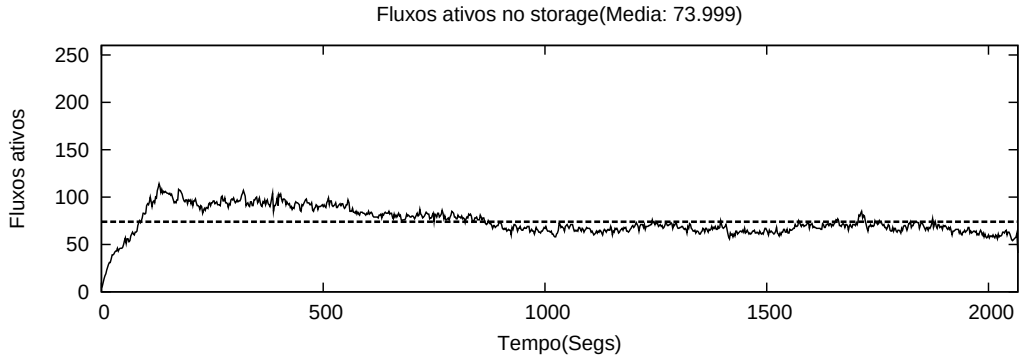


Figura 4.7: Fluxos ativos no *storage* (LOW-UC-BUF).

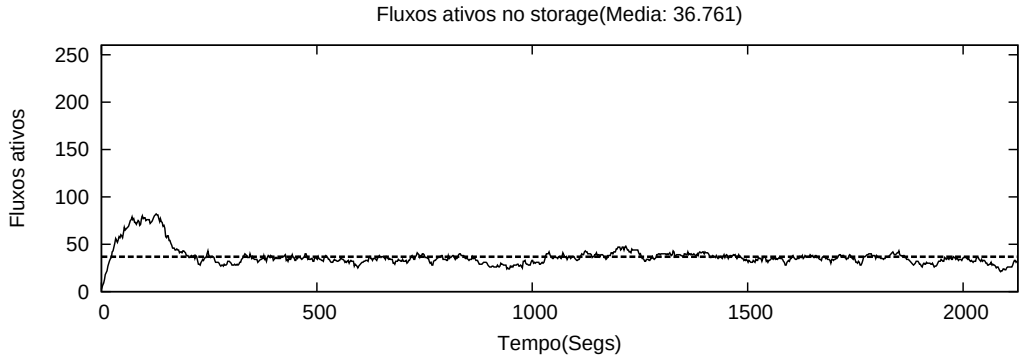


Figura 4.8: Fluxos ativos no *storage* (LOW-PIE-BUF).

cinco minutos.

A partir dos gráficos das figuras 4.9 e 4.10 pode-se observar a economia de banda proporcionada pela técnica PIE. Por exemplo, a probabilidade de termos mais de 50 fluxos ativos é aproximadamente 1 para a técnica *unicast* (Figura 4.9), enquanto que para a técnica PIE é menos de 0.1 (Figura 4.10). Neste cenário temos uma economia de banda média de aproximadamente 50,3%. A banda média para PIE é aproximadamente 40Mbps e para *unicast* é aproximadamente 81Mbps[3], conforme mostram as linhas marcadas nas figuras 4.7 e 4.8, respectivamente.

[3]Estes valores foram obtidos dos gráficos de número de fluxos ativos multiplicando-se o valor médio de fluxos ativos pela taxa de envio de dados de cada um deles, que é de aproximadamente 1.1Mbps.

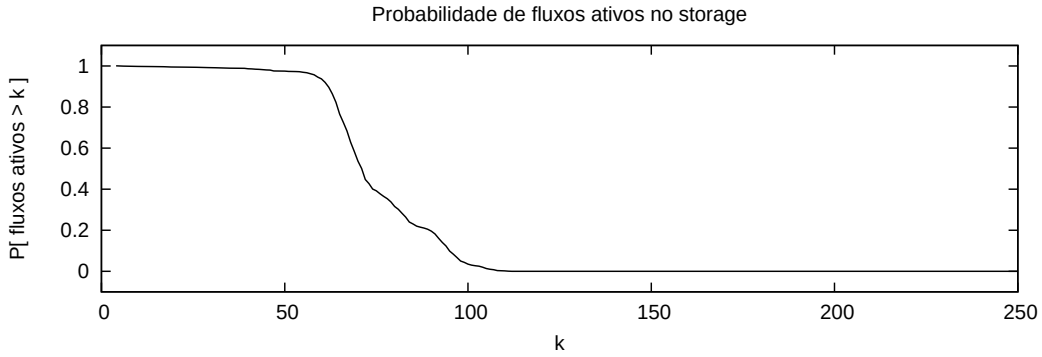


Figura 4.9: Probabilidade de fluxos ativos no *storage* (LOW-UC-BUF).

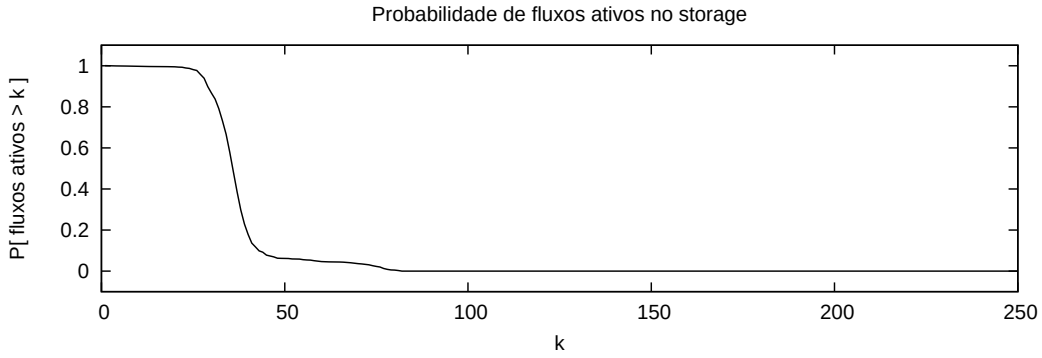


Figura 4.10: Probabilidade de fluxos ativos no *storage* (LOW-PIE-BUF).

4.1.3 Clientes com média interatividade

Os clientes com média interatividade tiveram resultados muito semelhantes aos do cenário anterior e, portanto, só apresentaremos o gráfico com a distribuição de probabilidade do número de fluxos ativos.

Como era esperado, para este cenário, os requisitos de banda média aumentaram com relação ao cenário anterior. Vemos nas figuras 4.11 e 4.12 que a banda média para o caso *unicast* é igual a aproximadamente 118Mbps e para o caso da técnica PIE é aproximadamente 50Mbps. Este resultado é explicado pelo fato de uma maior interatividade dos clientes aumentar a probabilidade de abertura de novos fluxos (*patch*, *unicast*, *multicast*). A economia de banda é aproximadamente igual a 57,8%.

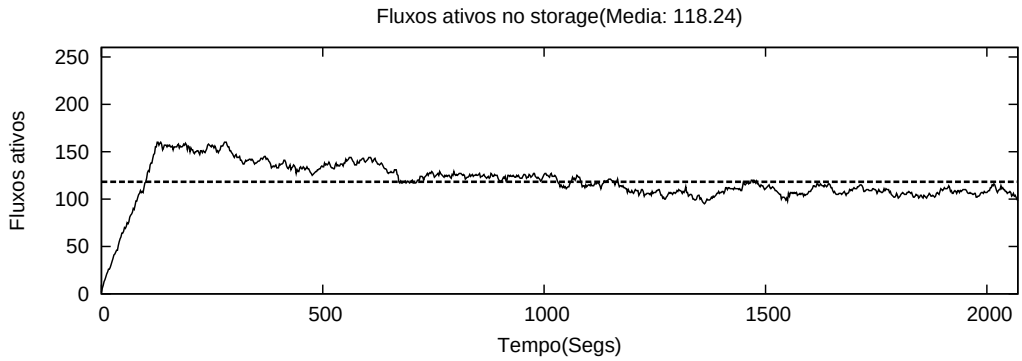


Figura 4.11: Fluxos ativos no *storage* (MED-UC-BUF).

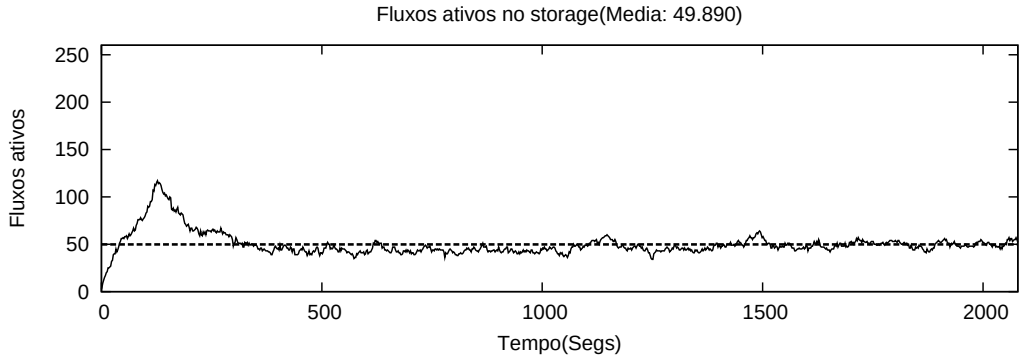


Figura 4.12: Fluxos ativos no *storage* (MED-PIE-BUF).

Podemos perceber que a economia de banda neste cenário foi maior que no cenário anterior. Este resultado se deve às características intrínsecas aos logs usados na emulação, características estas que permitiram uma maior economia de banda. Entre estas características podemos mencionar a maior incidência de interações que levam clientes a entrarem em grupos já existentes comparada às interações que geram novos grupos, e conseqüentemente novos fluxos, e uma maior incidência de interações para trechos que o cliente já tem em disco e, portanto, não precisará fazer requisição alguma.

4.1.4 Clientes com alta interatividade

Neste cenário, pelo fato da interatividade dos clientes ser alta, temos clientes entrando e saindo a todo instante de grupos e, mesmo que ele seja alocado em um grupo já existente, ele pode precisar fazer *patching*. Portanto esperávamos que o cliente estivesse quase que constantemente em *patching* ou fazendo requisições *multicast* para um novo grupo criado para ele após uma interação.

Note nas figuras 4.13 e 4.14 como a economia de banda foi bem satisfatória a despeito do esperado. Explicamos este resultado com o argumento de que os clientes *unicast* estão quase que constantemente enviando ao servidor requisições para preenchimento do *PlayOutBuffer* quando saltam para uma região para a qual eles não têm os blocos em disco. Isto também acontece com os clientes *multicast*, mas não com tanta freqüência pois quando o cliente faz uma interação e cai no δ_{before} de um grupo não é necessário fazer nenhuma requisição ao servidor, ele apenas escuta o fluxo *multicast* sendo enviado para o grupo que ele se juntou.

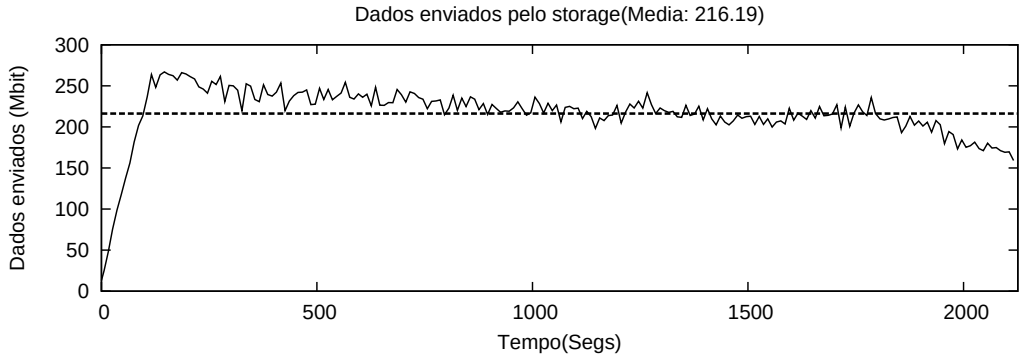


Figura 4.13: Dados enviados pelo *storage* (HIGH-UC-BUF).

Os gráficos das figuras 4.15 e 4.16 reforçam a idéia de que os clientes estejam realmente sendo, em grande parte dos casos, alocados em grupos já existentes. O número de fluxos *multicast* abertos chega, na maior parte do tempo, a ser por volta de um terço dos fluxos *unicast*.

A distribuição de probabilidade do número de fluxos ativos pode ser vista nas figuras 4.18 e 4.17. Através das figuras podemos notar que com a introdução da

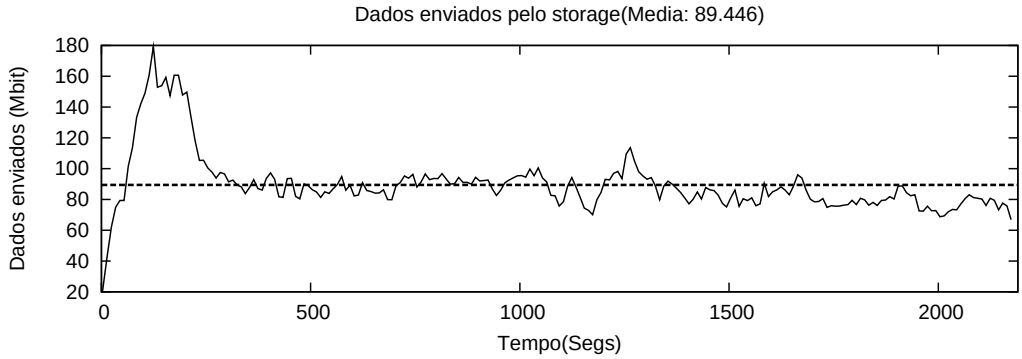


Figura 4.14: Dados enviados pelo *storage* (HIGH-PIE-BUF).

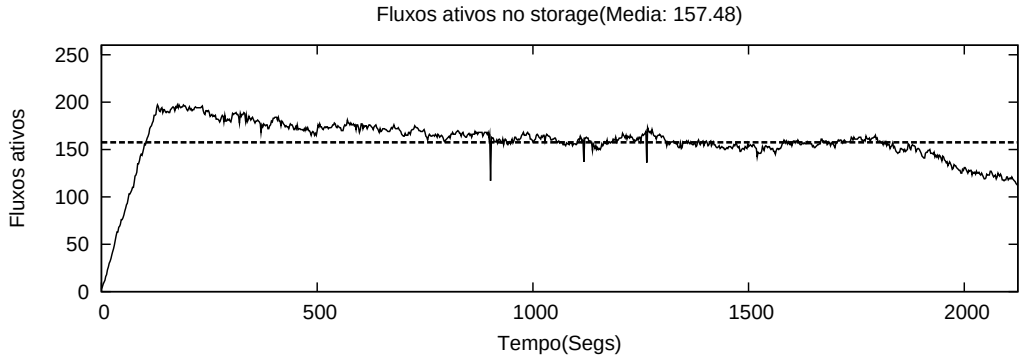


Figura 4.15: Fluxos ativos no *storage* (HIGH-UC-BUF).

técnica PIE a probabilidade de termos no sistema mais de 120 fluxos ativos é próxima de zero enquanto que para o cenário *unicast* esta probabilidade é próxima a 0.9.

A economia de banda média é de aproximadamente 58,6% para este cenário: a banda média para usuário *unicast* é aproximadamente 173Mbps e para a técnica PIE é aproximada a 66Mbps, conforme assinalado nos gráficos das figuras 4.15 e 4.16 respectivamente.

Temos no gráfico da figura 4.15 por volta de 157 fluxos ativos durante todo o experimento, apesar de termos disparado 250 clientes. Este resultado sugere que temos uma parte dos clientes em estado de pausa ao longo de todo o experimento[4].

[4]Escolhemos 15 logs de ações aleatórios, analisamos o número de clientes em pausa e percebemos que entre 3,5 e 4,5 deles, em média, estavam pausados ao mesmo tempo.

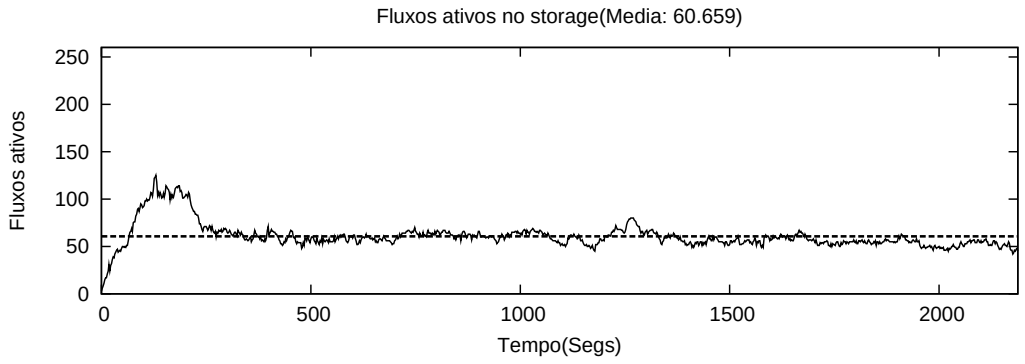


Figura 4.16: Fluxos ativos no *storage* (HIGH-PIE-BUF).

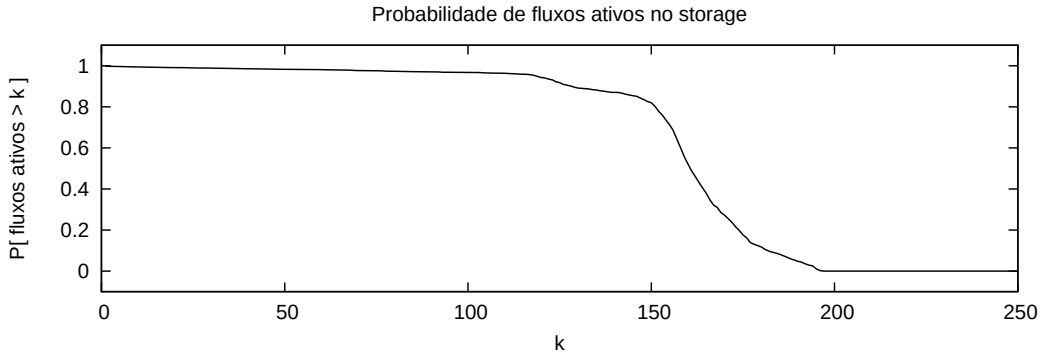


Figura 4.17: Probabilidade de fluxos ativos no *storage* (HIGH-UC-BUF).

4.1.5 Clientes com altíssima interatividade

Neste cenário, dados os logs de comportamento utilizados pelos clientes (explicado na seção 3.5.2), temos um ambiente de pouca estabilidade pois os clientes estão sempre saltando de uma parte para outra do vídeo. Com isto as expectativas são de que o uso do PL não se mostrará tão vantajoso em comparação à transmissão *unicast*, já que os clientes não terão tempo de se unir aos seus grupos (provavelmente farão uma nova interação antes que a junção ao grupo ocorra[5]) e têm grandes chances de sempre estar fazendo *patching*. Note nas figuras 4.19 e 4.20 que, como esperado, o uso do PL não foi tão vantajoso como vinha sendo nos cenários anteriores, mas ainda tem-se economia de banda média de aproximadamente 27,7% com o seu uso.

[5]Uma junção a um grupo leva, aproximadamente, tanto tempo quantos forem os blocos que ainda faltam receber no fluxo de *patching* para se alcançar o fluxo *multicast*.

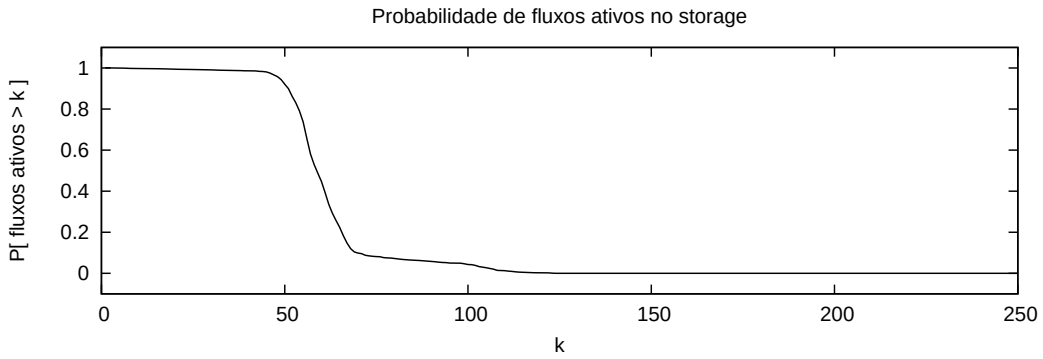


Figura 4.18: Probabilidade de fluxos ativos no *storage* (HIGH-PIE-BUF).

Isto fica mais claro nas figuras 4.21 e 4.22, onde percebe-se que a probabilidade de encontrarmos mais de 150 fluxos abertos no cenário *multicast* é próximo da metade da mesma probabilidade no cenário *unicast*.

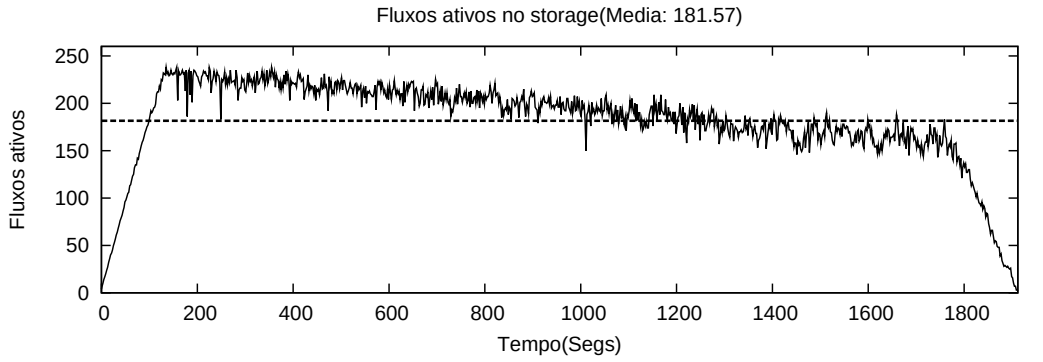


Figura 4.19: Fluxos ativos no *storage* (STRESS-UC-BUF).

4.1.6 Comparando os cenários quanto ao uso de *cache* em disco

Um fator que por mais simples que seja é um grande aliado à economia de banda do cliente interativo é o uso de uma *cache*. Como visto anteriormente na seção 2.3.4, chamamos de *cache* o *buffer* de armazenamento em disco e de *PlayOutBuffer* o *buffer* de armazenamento em memória, usado para exibição imediata de blocos de vídeo.

O uso da *cache* impede que um bloco seja requisitado mais de uma vez ao servidor,

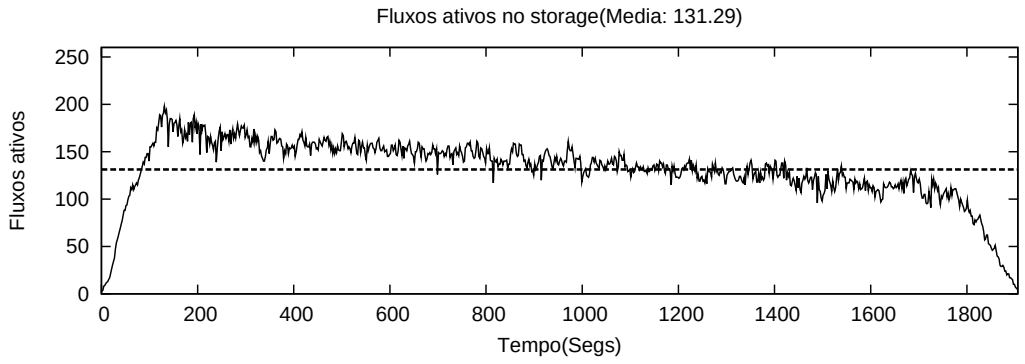


Figura 4.20: Fluxos ativos no *storage* (STRESS-PIE-BUF).

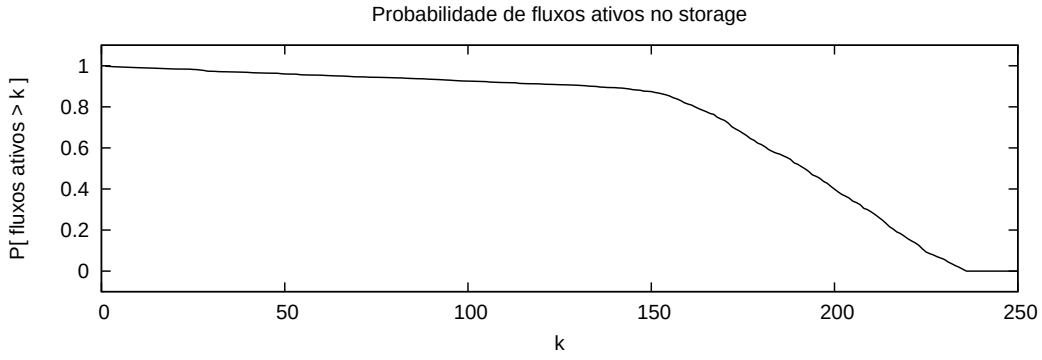


Figura 4.21: Probabilidade de fluxos ativos no *storage* (STRESS-UC-BUF).

e este evento é muito comum em clientes interativos pois uma das interações mais praticadas é a de saltos para trás, para trechos do vídeo já assistidos.

Uma análise do comportamento dos clientes interativos [Vielmond 2007, Tomimura et al. 2006] pode comprovar a teoria de que trechos de aula cujo conteúdo é mais difícil geralmente são assistidos mais de uma vez. Em sala de aula é menos comum os alunos pedirem ao professor para repetir uma explicação por mera timidez, mas este fator inibitório não existe quando o aluno está só, diante de um vídeo. Neste ambiente ele tem total liberdade de "pedir ao professor que repita o tópico" tantas vezes quantas ele quiser, bastando saltar para o ponto do vídeo que quer assistir novamente; e é exatamente isto que ele faz.

Se o cliente de vídeo fizer uso da *cache*, o servidor será bastante poupado cada vez que blocos já solicitados forem requisitados novamente pelo aluno. Estes blocos terão

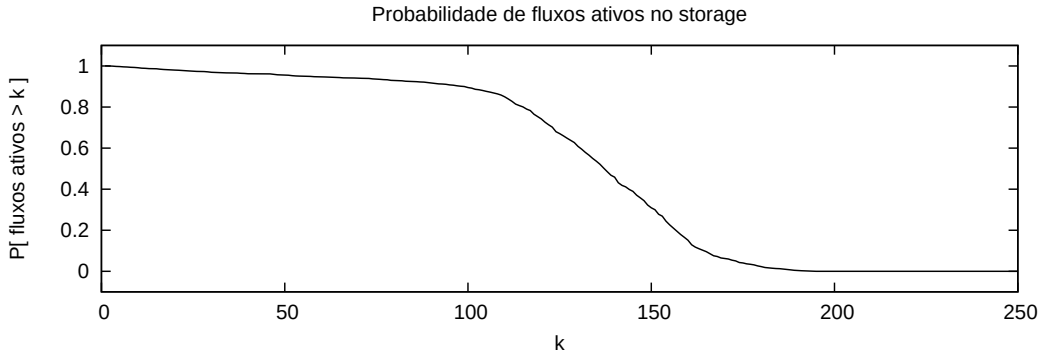


Figura 4.22: Probabilidade de fluxos ativos no *storage* (STRESS-PIE-BUF).

suas requisições atendidas sem gerar pedidos ao servidor: serão obtidos diretamente da *cache* do cliente. Nas figuras 4.23 e 4.24 percebemos que ocorre economia de banda de 11,7% quando se usa a *cache*.

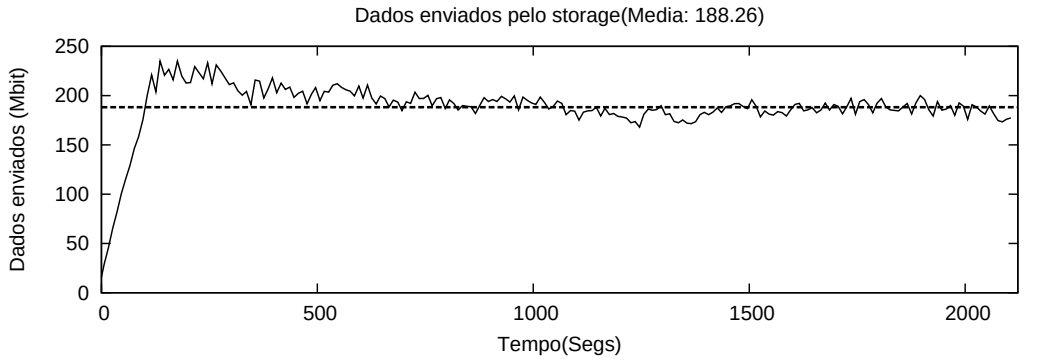


Figura 4.23: Dados enviados pelo *storage* (MED-UC-NOB).

4.1.7 Comparando as políticas PI e PIE

Na seção 4.1.1, quando consideramos clientes com acesso seqüencial, vimos que não houve nenhuma diferença significativa nos resultados das políticas PI e PIE. Analisemos agora, então, o cenário com grau de interatividade médio. Temos nas figuras 4.25 e 4.26 o número de fluxos ativos no *storage*. Note que a técnica de PIE não trouxe economia de banda alguma em relação à banda média usada pela técnica PI.

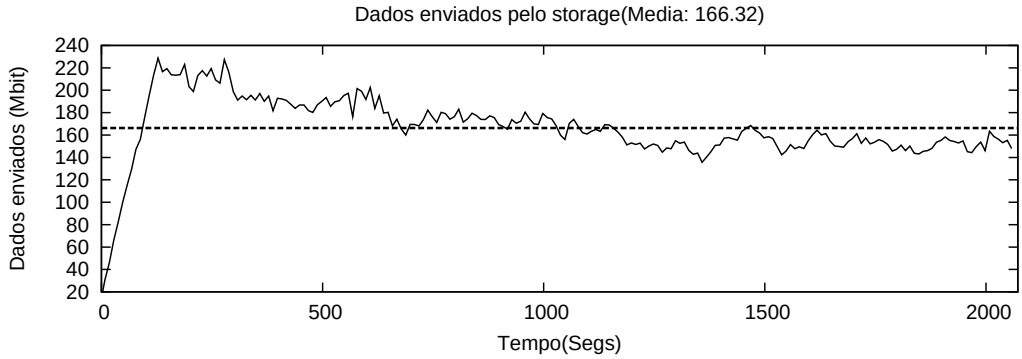


Figura 4.24: Dados enviados pelo *storage* (MED-UC-BUF).

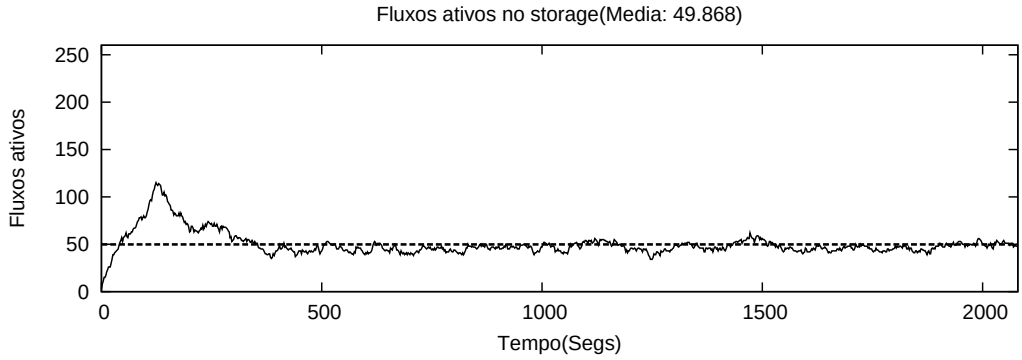


Figura 4.25: Fluxos ativos no *storage* (MED-PI-BUF).

Quando o cenário de comparação das técnicas é o de baixa interatividade, percebemos, conforme mostram as figuras 4.27 e 4.28, uma tímida economia de banda de aproximadamente 4% da técnica PIE sobre a PI.

A partir dos gráficos das figuras 4.29 e 4.30 fica mais evidente a pouca diferença entre as técnicas.

Para ambientes cujo nível de interatividade é altíssimo, esperamos recair em uma situação parecida com a do cenário onde o acesso é seqüencial. Isto porque não haverá (ou quase não haverá) fusões de grupos. Como os clientes estão dando saltos a cada vinte segundos, então mesmo que estejam como subgrupos de um grupo, a probabilidade de os membros do subgrupo interagirem e saírem deste subgrupo para um outro grupo ou subgrupo é bem alta, o que diminui a ocorrência de fusão de grupos, que é o único diferencial da PIE frente à PI. Os resultados das figuras 4.31

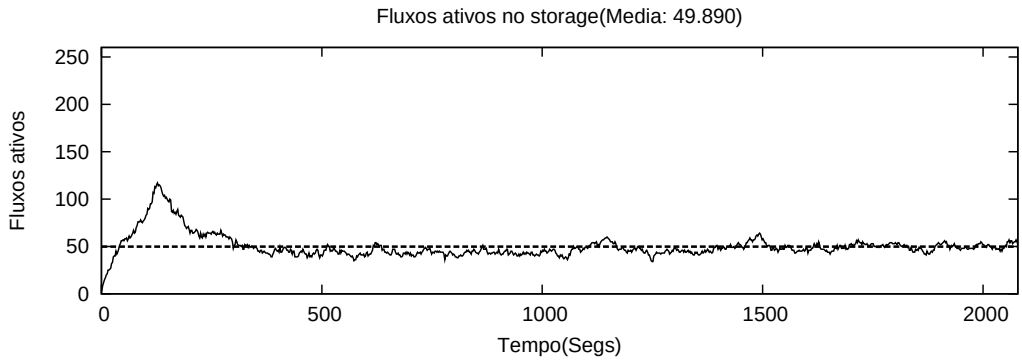


Figura 4.26: Fluxos ativos no *storage* (MED-PIE-BUF).

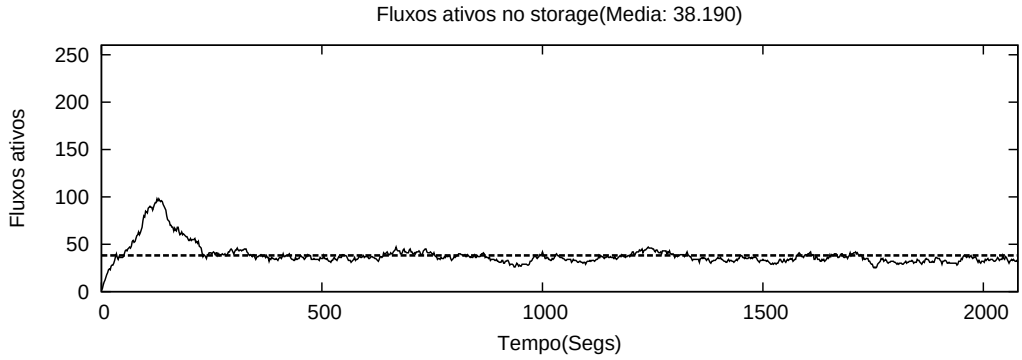


Figura 4.27: Fluxos ativos no *storage* (LOW-PI-BUF).

e 4.32 mostram que existe muito pouca diferença entre as técnicas para o cenário de altíssima interatividade .

Um último cenário de testes foi feito utilizando-se uma outra topologia, que será apresentada com mais detalhes na seção 4.1.9, onde a taxa de perdas é alta. Neste cenário, em nossos experimentos com nível de interatividade médio, tivemos como resultado uma superioridade de 7.7% de economia de banda da técnica PIE sobre a PI. Acreditamos que essa diferença se deva ao fato de que clientes agrupados sob a técnica PIE, por terem a possibilidade de participar de dois fluxos *multicast*, têm uma menor probabilidade de re-solicitar um bloco já recebido anteriormente que foi descartado após consumo por causa das perdas experimentadas.

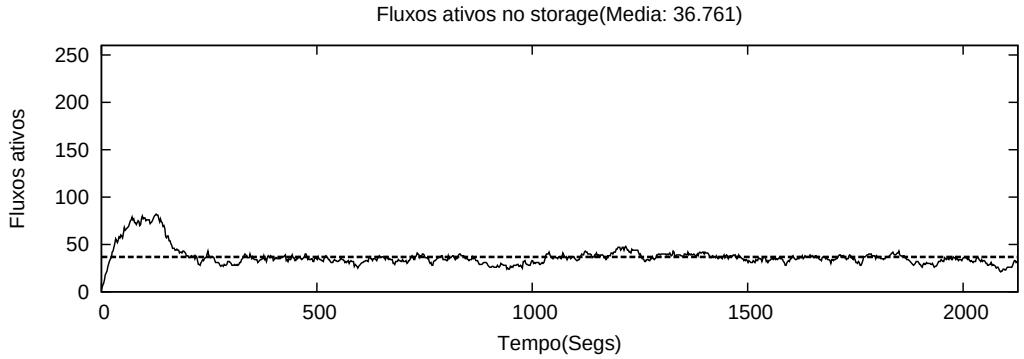


Figura 4.28: Fluxos ativos no *storage* (LOW-PIE-BUF).

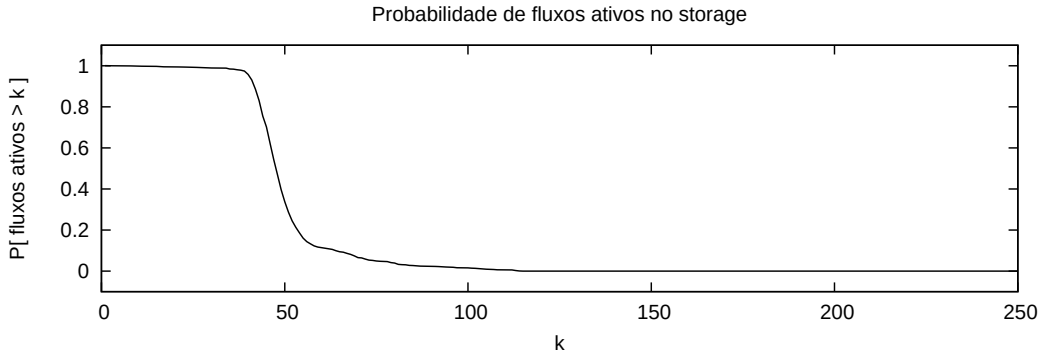


Figura 4.29: Probabilidade de fluxos ativos no *storage* (MED-PI-BUF).

4.1.8 Consumo dos recursos da máquina pelo *storage*, *dispatcher* e PL

Quando comparamos os gráficos de consumo dos recursos da máquina por parte do *dispatcher* e do *storage* nos diferentes cenários, não nos surpreendeu o fato de que geralmente o servidor RIO apresenta uma sobrecarga bem inferior quando trabalha com fluxos *multicast*.

Nas figuras 4.33 e 4.34 percebemos que o uso da CPU do *storage* é bem menor quando este tem o apoio do PL na administração dos diferentes grupos do sistema.

Quanto ao uso de CPU pelo *dispatcher*, confirmamos, com o auxílio das figuras 4.35 e 4.36, que a atuação do PL intermediando as requisições do cliente com o *dispatcher* faz com que este não faça tanto processamento pois o PL atua como um

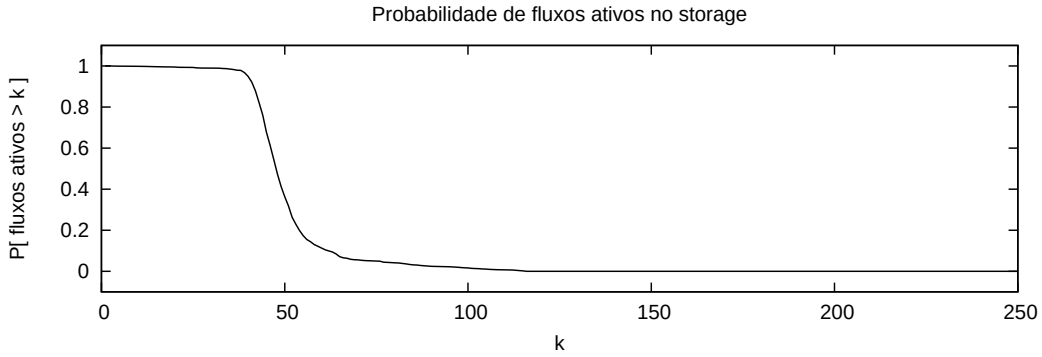


Figura 4.30: Probabilidade de fluxos ativos no *storage* (MED-PIE-BUF).

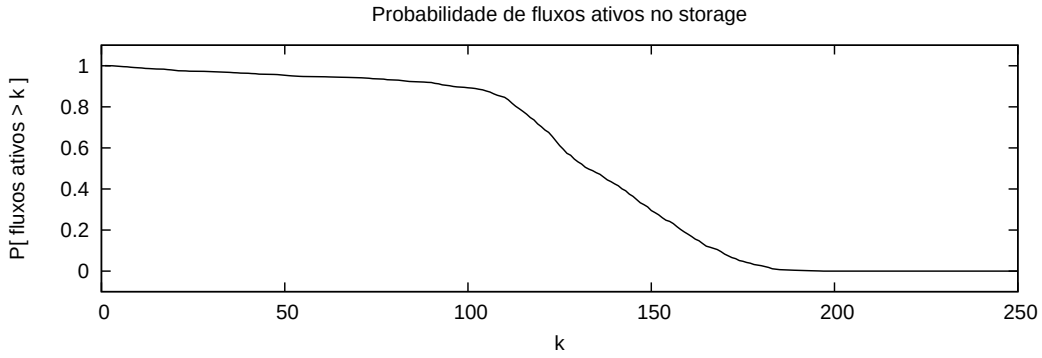


Figura 4.31: Probabilidade de fluxos ativos no *storage* (STRESS-PI-BUF).

filtro, onde só chegam ao *dispatcher* as requisições de *patching* e de fluxos *multicast*.

Obviamente que esta menor demanda de processador pelo *dispatcher* tem um custo que é um processamento extra feita pelo PL. Na figura 4.37 é exibido o consumo de CPU do PL, responsável pela diferença entre os gráficos das figuras 4.33 e 4.34 e das figuras 4.35 e 4.36.

4.1.9 As perdas nos clientes

O ambiente totalmente isolado, apresentado na figura 3.2, foi ideal para coletarmos resultados com o mínimo de influência de agentes externos aos nossos experimentos, como outros fluxos de dados, cascateamentos de *switches*, etc.

Antes de chegarmos à conclusão de que esta topologia era a ideal para nossos

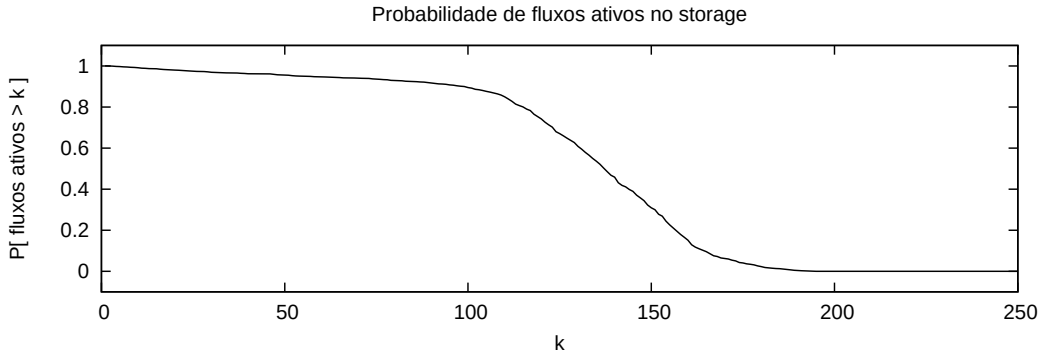


Figura 4.32: Probabilidade de fluxos ativos no *storage* (STRESS-PIE-BUF).

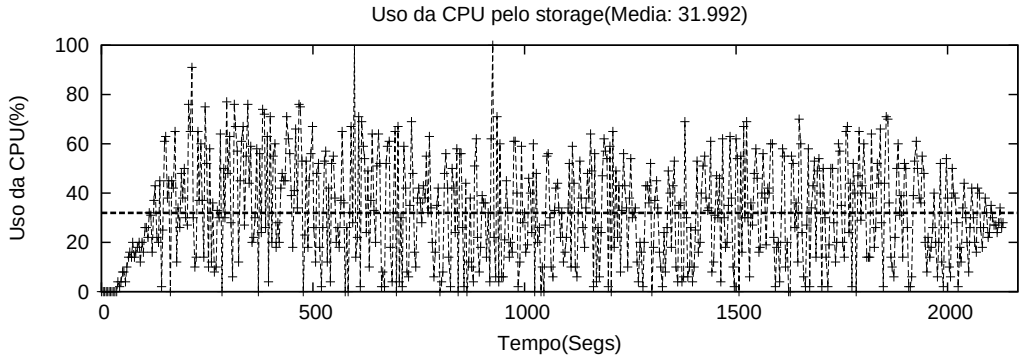


Figura 4.33: Consumo do processador pelo *storage* (HIGH-UC-BUF).

experimentos, usávamos uma outra, onde os resultados obtidos eram ligeiramente diferentes, reflexo de altíssimas perdas de fragmentos de blocos nos clientes. A topologia usada, até então, era a apresentada na figura 4.38. Em todos os experimentos realizados nesta configuração, o índice de perda era consideravelmente maior quando utilizávamos transmissão *multicast*. Para investigarmos o motivo das perdas, diversos testes foram feitos com os *switches* que interconectavam as máquinas, bem como experimentos onde os mesmos não eram utilizados. Nestes experimentos, conectamos as máquinas diretamente com um cabo *cross*. Isto obviamente nos limitou a um cenário com duas máquinas **A** e **B**, onde em **A** disparávamos o servidor e o PL e em **B** os clientes. Disparamos 50 clientes na máquina B e não tivemos perda alguma. Repetimos o experimento só que interligando as duas máquinas com o *switch*.

Quando disparamos um único cliente *multicast* neste ambiente envolvendo es-

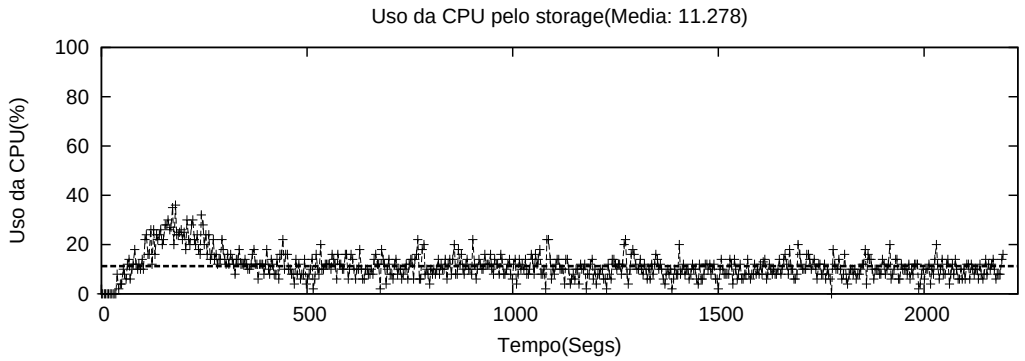


Figura 4.34: Consumo do processador pelo *storage* (HIGH-PIE-BUF).

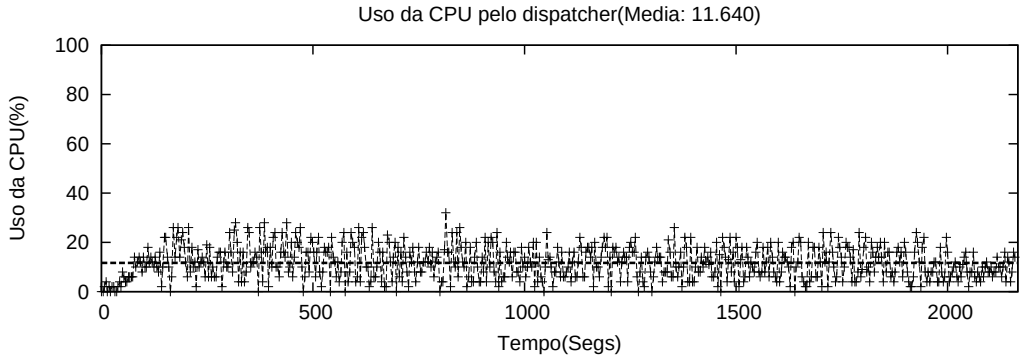


Figura 4.35: Consumo do processador pelo *dispatcher* (HIGH-UC-BUF).

tas mesmas máquinas, mas passando pelo *switch*, detectamos perdas. Com isto concluímos que as perdas acontecem nos *switches*.

Em [Inc. 2006] confirmamos a suspeita de que os *switches* descartam pacotes voluntariamente. Estes equipamentos implementam um algoritmo de controle de tráfego que descarta pacotes *broadcast* e *multicast* quando estes excedem uma determinada taxa, que é menos de um quarto da que nosso sistema necessita para funcionar. Uma vez desabilitado este limitador de tráfego tivemos uma melhora significativa no desempenho dos clientes, mas ainda ocorriam muitas perdas. Suspeitamos que o fato das perdas no *switch* acontecerem para fluxos *multicast* mas não para fluxos *unicast* se deve à alguma diferença na forma de processamento na hora de repassar os pacotes de dados para as portas de destino.

Na figura 4.39 exibimos um histograma do número de fragmentos perdidos dos

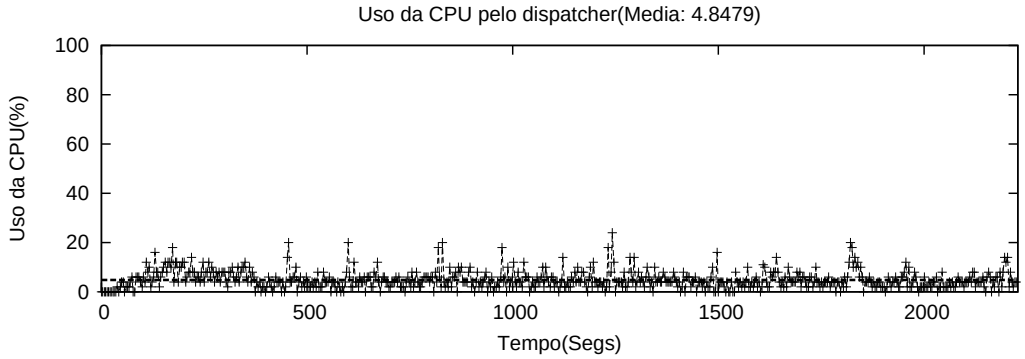


Figura 4.36: Consumo do processador pelo *dispatcher* (HIGH-PIE-BUF).

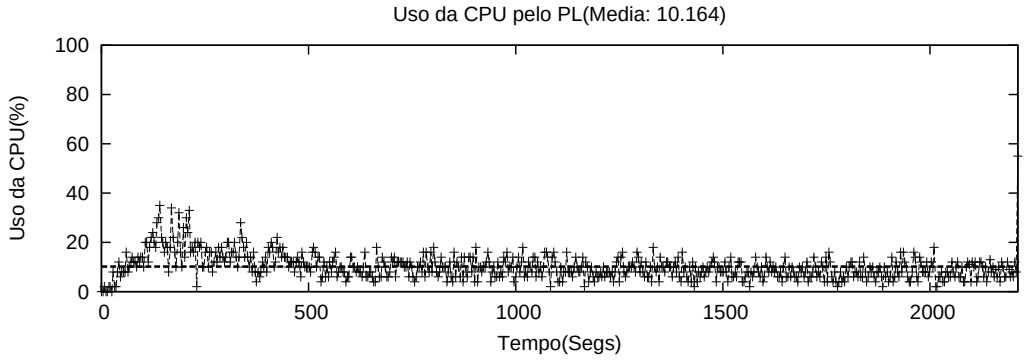


Figura 4.37: Consumo do processador pelo PL (HIGH-PIE-BUF).

duzentos e cinqüenta clientes disparados em uma emulação com duração de apenas 5 minutos configurada com a topologia da figura 4.38. Cada barra deste histograma agrupa uma perda de novecentos fragmentos, o que equivale a dez blocos de vídeo. Note que alguns clientes perderam mais de vinte mil fragmentos, no entanto, quando estes fragmentos não pertencem ao mesmo bloco, esta perda não causa tanto impacto para o cliente pois ele pode assistir o vídeo com uma qualidade inferior (os blocos, neste caso, são exibidos usando um subconjunto dos fragmentos).

De forma a avaliarmos o percentual de fragmentos perdidos em um mesmo bloco, obtivemos o histograma mostrado na figura 4.40. Nesta figura temos a média de fragmentos perdidos por bloco e o número de clientes correspondente. Pode-se notar que grande parte dos clientes perdia, em média, entre vinte e trinta fragmentos de cada bloco neste experimento.

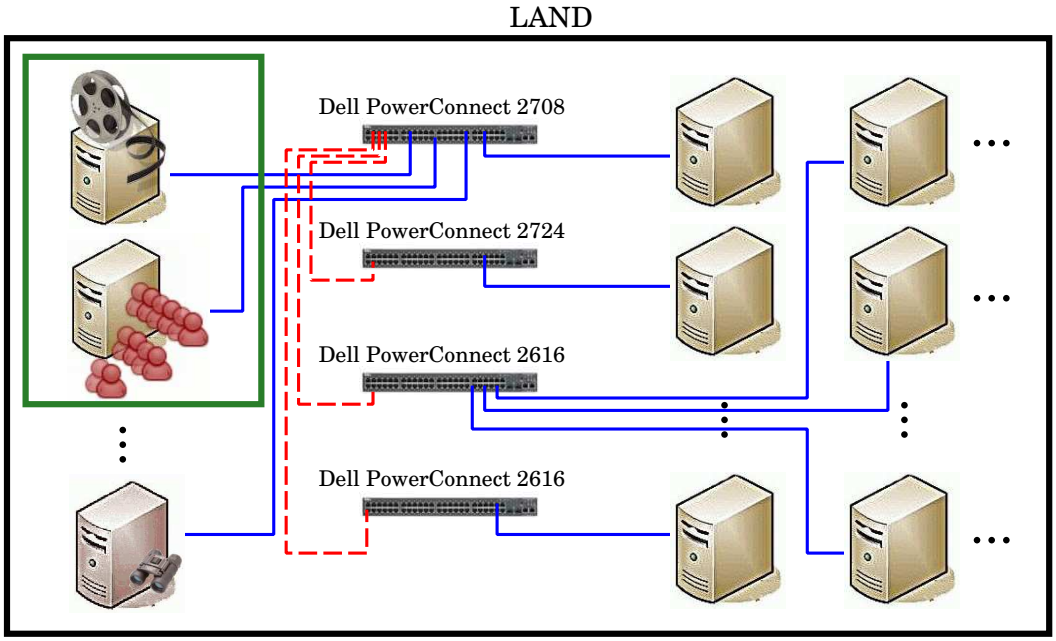


Figura 4.38: Topologia do ambiente de experimentos: Teste de perdas.

Um dos principais efeitos negativos das perdas é o de que blocos que não chegaram completos (todos os 90 fragmentos) não são armazenados na *cache* e, portanto, caso o cliente precise novamente dele em algum momento posterior, o mesmo será re-solicitado ao servidor. Isto faz com que os gráficos de uso de banda no cenário sem perdas seja mais atrativo.

No que tange às técnicas PI e PIE, entre si, a principal diferença foi a perda de vantagem no que diz respeito à economia de banda de rede da técnica PIE face à técnica PI. No ambiente com perdas altas a técnica PIE proporcionou, em um cenário de média interatividade, 7.7% de economia comparada à banda utilizada pela técnica PI.

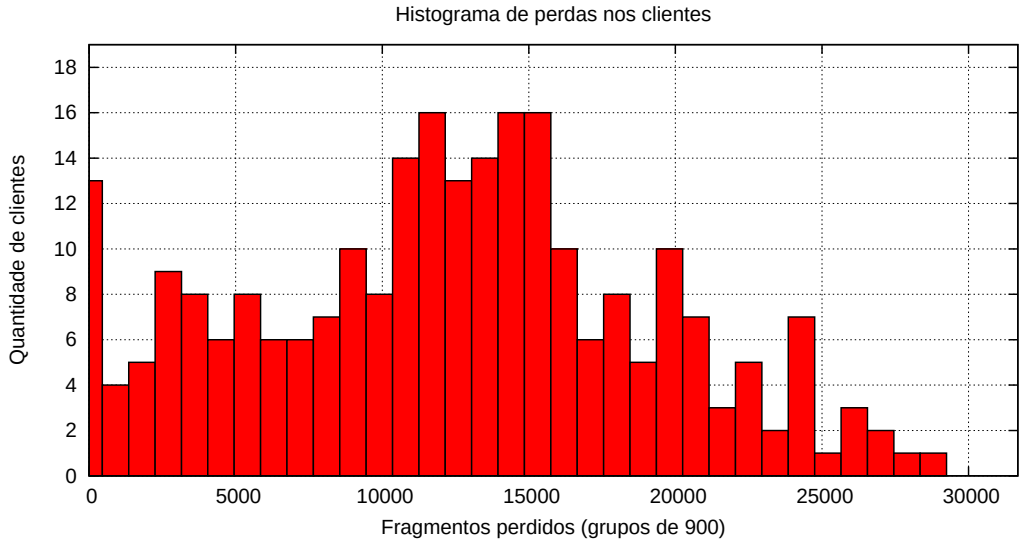


Figura 4.39: Histograma do número de fragmentos perdidos (HIGH-PIE-BUF).

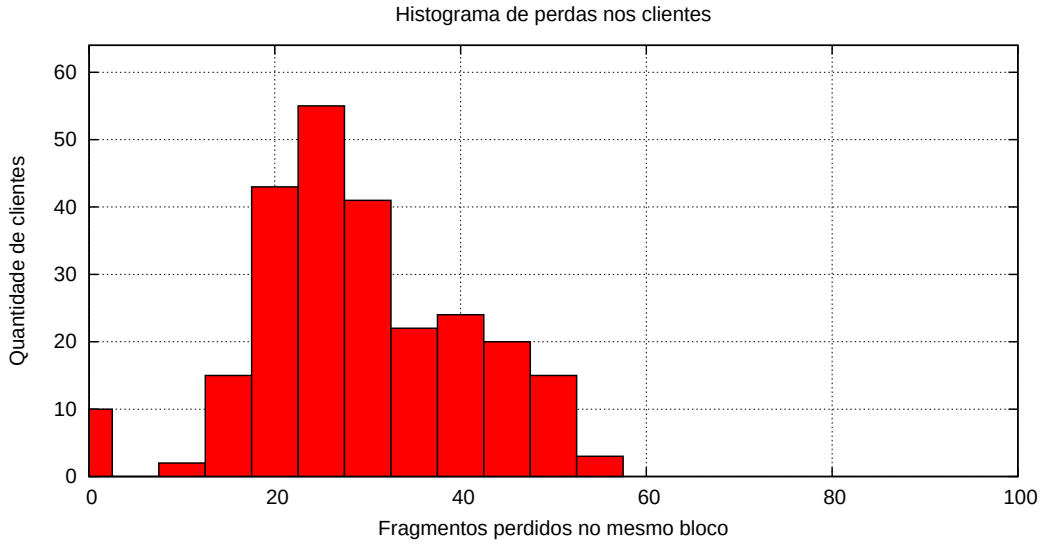


Figura 4.40: Histograma da média de fragmentos perdidos no mesmo bloco (HIGH-PIE-BUF).

Capítulo 5

Contribuições e Trabalhos Futuros

5.1 Contribuições

As contribuições deste trabalho podem ser divididas em dois grupos. O primeiro diz respeito à implementação e análise de desempenho da técnica PIE no servidor RIO e o segundo ao ambiente desenvolvido para experimentos e coleta de logs do servidor RIO.

Como visto nos resultados dos experimentos, o desempenho da técnica PIE foi superior ao da técnica PI, no entanto esta superioridade não foi expressiva e em alguns casos o uso da técnica PI pode ser mais indicado devido à sua maior simplicidade. Notamos, ainda, que há uma leve vantagem da técnica PIE sobre a PI quando o ambiente experimenta uma alta taxa de perdas.

Com relação a transmissão *unicast*, a técnica PIE obteve menores requisitos de banda em todos os cenários considerados. A maior economia de banda foi para o cenário de acesso seqüencial, onde a economia com relação à transmissão *unicast* foi de 95,2%. Todos os experimentos mostraram que o compartilhamento da banda é expressivo para todos os níveis de interatividade, embora não tenha sido detectada uma relação direta, em termos de proporção, do nível de interatividade com a

economia de banda.

Para cenários onde consideramos o nível de interatividade altíssimo a economia não foi expressiva. Este resultado pode ser explicado pois dado que os clientes interagem a cada vinte segundos, é muito baixa a probabilidade de haver junção de grupos. Para jogos, por exemplo, em que cada jogador tem uma visão, a transmissão *multicast* não traria economia de banda pois o nível de interatividade é tão alto que dois jogadores compartilharão pouca ou nenhuma transmissão de dados. Nos nossos experimentos as interações são a cada vinte segundos; em um aplicativo de jogo o grau de interatividade é consideravelmente mais alto que isto e os resultados seriam ainda menos satisfatórios se usássemos transmissão *multicast*.

Durante a confecção deste trabalho muitos experimentos foram feitos envolvendo cada um dos integrantes do sistema RIO: *dispatcher*, *storage*, PL e cliente de vídeo. Com isto todos os módulos, exceto o cliente administrativo (riosh), foram testados exaustivamente.

Com o uso das ferramentas valgrind[Valgrind-3.3.0 2008], gdb[GDB 2008] e *insure++*[Parasoft 2008] diversos problemas de concorrência e de gerenciamento de memória foram resolvidos, muitos dos quais só puderam ser percebidos ao fazermos experimentos com mais de quinhentos clientes.

Algumas classes sofreram reestruturação para uma maior legibilidade do código e para deixá-lo em conformidade com as regras de programação orientada a objeto. Primou-se, todo o tempo, por um código de fácil entendimento. Eliminamos diversas redundâncias de código nas várias reestruturações, redefinições e criação de classes internas de bibliotecas usadas por todo o sistema RIO.

Por se tratar de um software voltado também para pesquisa e uso acadêmico, foi montada uma estrutura de depuração das classes com uso de logs e mensagens obtidas em tempo de execução, o que facilita o programador no entendimento do sistema RIO e do relacionamento dos módulos e classes que o compõem. Desta forma esperamos diminuir o tempo que os próximos alunos e pesquisadores que vierem a

trabalhar com o sistema RIO levarão para entendê-lo.

Para testes com muitos clientes criamos um ambiente auto administrável, onde o usuário informa no início os parâmetros dos testes a serem executados (tempo mínimo, tempo máximo, quantidade de clientes em cada máquina, se os clientes são *unicast* ou *multicast*, o tamanho do *PlayOutBuffer* dos mesmos, etc). Uma vez ajustados os parâmetros, o ambiente será montado automaticamente: os *scripts*, arquivos de configuração e logs de comportamento dos clientes são replicados nas máquinas envolvidas no experimento, é verificado se cada uma das máquinas está apta para participar do experimento[1] e o experimento é iniciado. Periodicamente é verificado se todas as máquinas continuam aptas e se os módulos do servidor RIO (*dispatcher*, *storage* e PL) continuam sendo executados. Em caso de falha o experimento é finalizado e todos os clientes, em cada uma das máquinas utilizadas, bem como os módulos do servidor RIO, são finalizados, os seus logs agrupados em uma pasta e um email é enviado ao usuário informando do término forçado da emulação. Todo este ambiente foi montado de forma a ser autorecuperável tanto quanto possível e de evitar ao máximo que um experimento seja disparado se o ambiente não está totalmente preparado para isto. Isto evita resultados inválidos e o usuário ao término da emulação não precisa analisar log a log para ver se todos os clientes de todas as máquinas, bem como cada um dos componentes do servidor RIO, estiveram executando durante todo o experimento ou se algum deles abortou por algum motivo.

5.2 Trabalhos futuros

Um dos fatores que mais prejudicou a performance dos clientes foram as perdas de fragmentos. Sabemos que perdas que acontecem no caminho entre o cliente e o

[1] Uma máquina é considerada apta se ela está sincronizada com a máquina principal (que dispara a emulação), se está acessível via ssh e, no caso das máquinas que executam o PL e o servidor RIO, se tem as pastas destes módulos.

servidor, como por exemplo, *overflow* no *buffer* de um *switch*, são mais difíceis de serem controladas. No entanto, o descarte de 89 fragmentos de um bloco devido a perda do fragmento zero deste bloco poderia ser minimizado se, por exemplo, fosse adotada a solução de envio das mesmas informações que constam no fragmento zero em outro (além deste) ou outros fragmentos que compõe o bloco.

Nenhuma variação foi feita nos valores das janelas δ_{before}, δ_{after} e δ_{merge} das técnicas PIE e PI e no tamanho do *PlayOutBuffer* dos clientes. Seria interessante um estudo da influência destes parâmetros no desempenho do sistema RIO e relações que os mesmos têm entre si.

Uma outra sugestão para continuação deste trabalho é um estudo envolvendo redes heterogêneas para analisar desempenho das técnicas aqui estudadas em *links* de capacidades diferentes. Com isto teremos uma análise do sistema RIO em ambientes não tão ideais quanto o dos experimentos aqui apresentados e as estatísticas obtidas estariam mais próximas de um ambiente como o da internet, onde os dados trafegam além da rede local. Para este estudo clientes deveriam ser executados em máquinas remotas, conectadas a diferentes redes de acesso, inclusive redes sem fio.

Referências Bibliográficas

[A. Dan e Shahabuddin] A. Dan, D. S. e Shahabuddin, P. Dynamic batching policies for an on-demand video server. In *ACM Multimedia Systems, 1996*.

[Almeida et al. 2001] Almeida, J. M., Krueger, J., Eager, D. L., e Vernon, M. K. (2001). Analysis of educational media server workloads. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pp. 21–30, New York, NY, USA. ACM.

[Botelho 2005] Botelho, E. R. (2005). Análise da interatividade dos alunos acessando um servidor multimídia em um ambiente de ensino a distância. Technical report, UFRJ - Universidade Federal do Rio de Janeiro.

[Botelho 2006] Botelho, V. M. (2006). Experimentos com o servidor RIO em um ambiente distribuído e heterogêneo. Tese de Mestrado, Universidade Federal do Rio de Janeiro - COPPE - Programa de Engenharia de Sistemas e Computação.

[C. C. Aggarwal e Yu] C. C. Aggarwal, J. L. W. e Yu, P. S. On optimal piggyback merging policies for video-ondemand systems. In *ACM SIGMETRICS, 1996*.

[Cardozo 2002] Cardozo, A. Q. (2002). Mecanismos para garantir qualidade de serviço de aplicações de vídeo sob demanda. Tese de Mestrado, Universidade Federal do Rio de Janeiro - COPPE - Programa de Engenharia de Sistemas e Computação.

[Carolina C. L. B. de Vielmond] Carolina C. L. B. de Vielmond, Rosa M. M. Leão, E. d. S. e. S. Um modelo HMM hierárquico para usuários interativos acessando um servidor multimídia. In *Simpósio Brasileiro de Redes de Computadores, 2007*.

[CEDERJ 2008] CEDERJ (2008). Centro de Educação superior a Distância do Estado do Rio de Janeiro. URL http://www.cederj.edu.br/fundacaocecierj/.

[Cheng 2008] Cheng, W. C. (2008). Tangram Grafics Interface Facility. URL http://bourbon.usc.edu/tgif/.

[Costa et al. 2004] Costa, C. P., Cunha, I. S., Borges, A., Ramos, C. V., Rocha, M. M., Almeida, J. M., e Ribeiro-Neto, B. (2004). Analyzing client interactivity in streaming media. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pp. 534–543, New York, NY, USA. ACM.

[D. L. Eager e Zahorjan] D. L. Eager, M. K. V. e Zahorjan, J. Optimal and efficient merging scheduling for video-on-demand servers. In *ACM SIGMETRICS, 1999*.

[Dan et al.] Dan, A., Sitaram, D., e Shahabuddin, P. Scheduling policies for an on-demand video server with batching. In *ACM Multimedia Systems, 1996*.

[GDB 2008] GDB (2008). GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/.

[Gereoffy 2008] Gereoffy, A. (2008). MPlayer - Movie player for linux. URL http://www.mplayer.org.

[Gorza 2003] Gorza, M. L. (2003). Uma técnica de compartilhamento de recursos para transmissão de vídeos com alta interatividade e experimentos. Tese de Mestrado, Universidade Federal Fluminense - Curso de Pós Graduação da UFF.

[Group 2008a] Group, N. R. (2008a). Nemertes Research Group. http://www.nemertes.com.

[Group 2008b] Group, R. W. (2008b). NeTraMeT. URL http://www.caida.org/tools/measurement/netramet/.

[Hua e Sheu] Hua, K. A. e Sheu, S. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demans systems. In *SIGGCOM, 1997*.

[Inc. 2006] Inc., D. (2006). Dell(tm) powerconnecttm 27xx - systems user's guide. Technical report, DELL Inc.

[iPoque 2008] iPoque (2008). iPoque - Insight & Control. URL http://www.ipoque.com/media/internet_studies/internet_study_2007.

[K. A. Hua e Sheu] K. A. Hua, Y. C. e Sheu, S. Patching: A multicast technique for true video-on-demand services. In *ACM SIGMETRICS, 1998*.

[Kurose e Ross 2000] Kurose, J. F. e Ross, K. W. (2000). *Computer Networking - A Top-Down Approach Featuring the Internet*. Addison-Wesley.

[Land 2008] Land (2008). Laboratório land. URL http://www.land.ufrj.br.

[MediaWiki 2008] MediaWiki (2008). Wikipédia. URL http://pt.wikipedia.org.

[Mozilla 2008] Mozilla (2008). Mozilla firefox. URL http://www.mozilla.org.

[Netscape 2008] Netscape (2008). Netscape NetCenter. URL http://wp.netscape.com/pt/.

[Netto 2004] Netto, B. C. M. (2004). Patching interativo: um novo método de compartilhamento de recursos para transmissão de vídeo com alta interatividade. Tese de Mestrado, Universidade Federal do Rio de Janeiro - COPPE - Programa de Engenharia de Sistemas e Computação.

[Pacheco 2007] Pacheco, A. (2007). Modelagem e Análise do Comportamento do Servidor RIO em Ambientes Reais e Heterogêneos . Tese de Mestrado, Universidade Federal do Rio de Janeiro - COPPE - Programa de Engenharia de Sistemas e Computação.

[Parasoft 2008] Parasoft (2008). Insure++. http://www.parasoft.com.

[RNP 2008] RNP (2008). RNP - Rede Nacional de Ensino e Pesquisa. http://www.rnp.br.

[Rocha et al. 2005] Rocha, M., Maia, M., Ítalo Cunha, Almeida, J., e Campos, S. (2005). Scalable media streaming to interactive users. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pp. 966–975, New York, NY, USA. ACM.

[Rodrigues 2006] Rodrigues, C. K. S. (2006). *Mecanismos de compartilhamento de recursos para aplicações de mídia contínua na internet.* PhD thesis, Universidade Federal do Rio de Janeiro - COPPE - Programa de Engenharia de Sistemas e Computação.

[Stevens 1997] Stevens, W. R. (1997). *Unix Network Programming - Networking APIs: Sockets and XTI - Vol1.* Prentice Hall.

[Tomimura et al. 2006] Tomimura, D., Leão, R. M. M., e Silva, E. A. S., e Filho, F. S. (2006). Caracterização do comportamento de usuários acessando um vídeo de ensino à distância. Tese de Mestrado.

[UCLA 2008] UCLA (2008). University of California, Los Angeles. http://www.ucla.edu.

[Valgrind-3.3.0 2008] Valgrind-3.3.0 (2008). Valgrind. http://www.valgrind.org.

[Valle 2007] Valle, M. (2007). Um Estudo do Servidor RIO em Uma Rede de Alta Velocidade. Tese de Mestrado, Universidade Federal do Rio de Janeiro - COPPE - Programa de Engenharia de Sistemas e Computação.

[Vielmond 2007] Vielmond, C. C. L. B. (2007). Um modelo HMM hierárquico para usuários interativos acessando um servidor multimídia. Tese de Mestrado, Universidade Federal do Rio de Janeiro - COPPE - Programa de Engenharia de Sistemas e Computação.

[W3C 2008] W3C (2008). XML - eXtensible Markup Language. URL http://www.xml.org.

[Youtube 2008] Youtube, L. (2008). Youtube. URL http://www.youtube.com.

[Zafalão 2004] Zafalão, R. M. (2004). Protocols de difusão periódica de vídeo sob limitação de banda passante. Tese de Mestrado, Universidade Estadual de Campinas - UNICAMP.