EXPERIMENTOS COM O SERVIDOR RIO EM UM AMBIENTE DISTRIBUÍDO E HETEROGÊNEO

Vinicius Martins 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. Paulo Henrique de Aguiar Rodrigues, Ph. D.

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

RIO DE JANEIRO, RJ - BRASIL

JULHO DE 2006

BOTELHO, VINICIUS MARTINS

Experimentos com o Servidor RIO em um ambiente distribuído e heterogêneo [Rio de Janeiro] 2006

XVII, 100 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2006)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Balanceamento de Carga
2. Vídeo sob Demanda
3. Redes Gigabit
4. Diversidade de Caminhos

I. COPPE/UFRJ II. Título (série)

Aos meus pais, pedra angular de minha vida.

Agradecimentos

Em primeiro lugar, devo agradecer a Deus, por ter pensado em mim desde a eternidade, ter me criado e me sustentado até hoje. Agradeço também aos meus pais, que são co-responsáveis, em parceria com Deus, pela minha existência. Além disso, é o seu amor por mim que me faz capaz de compreender, ainda que palidamente, a infinidade do Amor divino. E também devo agradecer os seus incentivos constantes e ajuda sem limite que me prestaram durante toda a minha vida.

Quero agradecer muitíssimo aos professores Edmundo e Rosa, que sempre confiaram e acreditaram em mim, mesmo nos momentos em que ninguém o faria. Espero que este trabalho seja minha retribuição pela confiança depositada em mim, desde 2001, quando fui aluno do professor Edmundo em TP.

Um agradecimento muito especial está reservado para a secretária mais querida da UFRJ: a Carol. Ela foi calma quando eu era intempestivo, ria quando eu chorava e ouvia quando eu precisava desabafar.

Tenho que agradecer a todos meus amigos e amigas que, de alguma forma, contribuíram para a conclusão deste trabalho: Primeiramente devo agradecer a Juliana Cunha, pelo incentivo, paciência e correção do *abstract*. Um agradecimento muito importante deve ser feito ao Bernardo Netto, tamanha foi sua ajuda na implementação no RIO e outro ao Flávio, por me ajudar a destrinchar o bisonho comportamento do *kernel* do Linux. Quero agradecer em especial os administradores do LAND, Boechat, Hugo e Carolzinha, por sempre concederem os *sudos* que eu pedia prontamente. E a todos do LAND: GD, Guto, Fabianne, Ana Paula, Allyson, Bene, Fernando, Sadoc, Diana, Isabela, Watanabe e demais, pelo companheirismo e prontidão no auxílio!

Agradeço aos "garotos e garota"que trabalharam comigo durante o Projeto Giga/RNP: Bruno, Marcello e Ariadne. Este trabalho foi uma conseqüência do

Projeto Giga/RNP: agradeço a RNP e a FINEP pelo financiamento e pela realização do projeto agradeço a COPPE/UFRJ, DCC/UFF, DCC/UFMG e ao Canal Saúde/FIOCRUZ. Deste projeto veio parte dos recursos de minha bolsa e agradeço ao FAPERJ, que fez a complementação na etapa final. Também agradeço a Fundação Coppetec, que viabilizou financeiramente meus estudos desde a metade de minha graduação e foi a forma pela qual recebi a bolsa do Projeto Giga/RNP.

Por fim, mas não menos importante, agradeço ao Bernardo Miranda, que me deu uma carona vital para cumprir um prazo. Também agradeço aos amigos da UFF, professora Anna Dolejsi e Sandoval, aos amigos da UFMG, professora Jussara e Alex Borges e aos amigos do Canal Saúde, Angélica, Odir, Mirela e Neide.

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

EXPERIMENTOS COM O SERVIDOR RIO EM UM AMBIENTE DISTRIBUÍDO E HETEROGÊNEO

Vinicius Martins Botelho

Julho/2006

Orientador: Edmundo Albuquerque de Souza e Silva

Programa: Engenharia de Sistemas e Computação

Nesta dissertação, realizamos uma série de experimentos no servidor multimídia RIO em um ambiente real que incluiu a Internet e uma rede paralela com velocidades de gigabits por segundo. Assim, foi possível verificar possíveis gargalos de desempenho e estudar maneiras de contorná-los. Como o ambiente de experimentação inclui uma rede com caminhos heterogêneos, com taxas de perda e tempo de propagação claramente distintos, também foi possível traçar um paralelo com um problema conhecido na literatura como diversidade de caminhos. Utilizamos métricas para avaliar o desempenho do sistema, como a taxa de *goodput* para o cliente. Assim, avaliamos qual o impacto, para a qualidade de serviço (QoS), da adição de nós extras de armazenamento e comparamos os resultados da adição em cada rede. Também investigamos o impacto da replicação dos vídeos na QoS do sistema, além do aumento de confiabilidade. Concluímos que a adição de um nó de armazenamento na rede contribui positivamente para a capacidade total do sistema, mesmo que esta introduza alta latência e taxa de perda.

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

EXPERIMENTS WITH THE RIO SERVER IN A DISTRIBUTED AND HETEROGENEOUS ENVIRONMENT

Vinicius Martins Botelho

July/2006

Advisor: Edmundo Albuquerque de Souza e Silva

Department: Systems Engineering and Computer Science

In this dissertation, we have done experiments with the RIO multimedia server in a real environment that includes the Internet and an additional network with gigabit per second bandwidth. We investigated the potential performance bottlenecks and we studied ways to circunvent them. As the experimentation environment had heterogeneous paths, with clearly distinct loss rates and propagation delays, it was also possible to relate our scenario with the problem known in the literature as path diversity. We used known metrics to measure the system's performance, such as the client's goodput rate. We measured the impact on the Quality of Service (QoS) of adding extra storage nodes both in the gigabit network and in the Internet. We also investigated the possible performance improvements of block replication in the system's QoS. We conclude that the addition of a storage node to the system can have a positive impact to the overall system performance, even if this node is connected to the server through a network with higher packet loss rate and delay.

Lista de Acrônimos

VoD :	*Video on Demand*;
DVoD :	*Distributed Video on Demand*;
P2PVoD :	*Peer-to-Peer Video on Demand*;
RIO :	*Randomized I/O Multimedia Server*;
Mbps :	*Megabits per second*;
KB :	*Kilobytes*;

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução e Motivação

SERVIDORES de vídeo sob demanda possuem diversas peculiaridades inerentes à aplicação, que os distinguem de outros sistemas computacionais. Por exemplo, podemos citar: a baixa tolerância a perdas de dados, a sensibilidade a variações de atrasos na rede, a alta taxa de dados para cada fluxo de vídeo, a sensibilidade a latência, etc.

Devido a estas características, sistemas multimídia para transmissão de vídeo sob demanda são sistemas complexos e não é trivial prever o seu comportamento em cenários distintos. Por exemplo, determinar o gargalo de desempenho do sistema depende de muitos parâmetros que devem ser avaliados em conjunto. Por isso, além da modelagem, a experimentação é fundamental para a determinação destes parâmetros, os quais também influenciam a qualidade de serviço, uma métrica de extrema importância em sistemas multimídia. Como exemplo destes parâmetros, podemos citar as limitações do sistema operacional, o consumo de memória, limitações do equipamento utilizado, etc. Por conseguinte, a prototipagem e experimentação tornam-se indispensáveis para determinar problemas, muitas vezes desconsiderados por ser assumidos irrelevantes ou por se tratar de simplificações realizadas durante a análise teórica, por questões de tratabilidade do modelo gerado.

O objetivo deste trabalho é o de realizar uma série de experimentos em um servidor multimídia em um ambiente real que inclua a Internet e uma rede paralela com velocidade de gigabits por segundo. Assim, podemos verificar possíveis gargalos de desempenho e estudar maneiras de contorná-los. Como o ambiente de experimen-

tação inclui uma rede com caminhos heterogêneos, com taxas de perda e tempo de propagação claramente distintos, também pode-se traçar um paralelo com um problema conhecido na literatura como diversidade de caminhos.

1.1 Principais objetivos

O principal objetivo deste trabalho é realizar experimentações usando o servidor RIO, descrito no Capítulo 3, em um cenário real e em alta carga. Além disso, são estudados pontos de gargalo neste cenário e são propostas maneiras de contorná-los, para aumentar a capacidade do sistema. Para tal, foi implementado no servidor a redundância de blocos para mais de uma réplica, além da correção de erros, otimização de desempenho e utilização de memória. Também foram criados diversos *scripts* para a execução de todos os experimentos, desde a inicialização do ambiente até a execução dos clientes e confecção dos gráficos resultantes. O tipo de serviço que desejamos experimentar é um servidor de vídeo sob demanda com aplicação para ensino a distância, por este serviço ser de extrema relevância por sua aplicabilidade no Brasil. Isto é devido ao fato do servidor RIO estar sendo utilizado desde 2005 no curso de Tecnologia de Sistemas de Computação do consórcio CEDERJ de universidades públicas do Estado do Rio de Janeiro. Os servidores em uso atendem alunos dos pólos do CEDERJ, dentro de uma rede local. É nosso objetivo a realização de experimentos para determinar o desempenho do servidor em um ambiente distribuído e a sua escalabilidade.

Também é estudado o comportamento de um servidor multimídia em uma rede heterogênea e é traçado um paralelo entre este comportamento e o problema de diversidade de caminhos.

1.2 Principais contribuições

A contribuição deste trabalho foi a construção e elaboração de um ambiente distribuído e heterogêneo de experimentação para o servidor RIO e subsequente realização de experimentos em alta carga para verificar os limites de desempenho do servidor. Como parte da elaboração do ambiente, o código do servidor foi modificado

para atender as necessidades dos experimentos propostos e corrigir falhas.

O ambiente é composto por nós de armazenamento distribuídos em diferentes instituições, tais como UFRJ, UFF, UFMG e Fiocruz. Três destes locais estão conectados a rede Giga da RNP e um deles está conectado somente a Internet. Este ambiente heterogêneo permitiu a realização de experimentos de desempenho e confiabilidade.

Como resultado dos experimentos sobre o servidor multimídia distribuído, pode-se verificar que, mesmo que um dos nós de armazenamento esteja conectado através de um caminho com perdas e latências muito maiores que os demais caminhos, este nó poderá contribuir positivamente para a capacidade total do sistema. Uma conclusão importante foi obsevar que o servidor RIO, descrito no Capítulo 3, usado nos experimentos, possui muito mais espaço para ser escalado. Em outras palavras, foi verificar que o nó servidor não é o gargalo. Portanto, distribuindo-se os nós de armazenamento, poder-se-ia conseguir um sistema robusto, tolerante a falhas e de alto desempenho, permitindo acesso a milhares de usuários.

1.3 Por que estudar vídeo sob demanda?

O inegável sucesso da Internet pode ser percebido pelo incessante crescimento de pessoas conectadas, no Brasil e no mundo. O sucesso da Internet deve ser atribuído, em grande parte, às inúmeras aplicações existentes e ao imenso volume de dados disponíveis aos usuários.

Com a evolução da tecnologia, passou a ser possível a veiculação de vários tipos de mídia além do texto puro. Imagem foi a primeira nova mídia, seguida por áudio e, posteriormente, vídeo. Aplicações que combinam em parte ou no todo estas mídias passaram a ser chamadas de aplicações multimídia. Estas aplicações são numerosas. Como exemplos, podemos citar, dentre outras: rádio e televisão online, ensino a distância, locadoras de filmes online, cirurgias à distância, treinamento e simulação em um mundo virtual, tele e videoconferências, etc;

Para a implantação de um serviço de transmissão de conteúdo multimídia para ensino a distância, temos duas abordagens naturais: podemos transmitir, armazenar

e reproduzir todo o conteúdo ao final de sua recepção ou então podemos reproduzí-lo de acordo com sua chegada. A primeira abordagem é a mais simples, mas extremamente limitada, especialmente no contexto de uma aplicação para ensino a distância. A segunda abordagem é a ideal por prover a interatividade e flexibilidade que este tipo de serviço exige, apesar de possuir diversos problemas inerentes que devem ser tratados. Falaremos deles mais a frente nesta seção.

Portanto, este trabalho utiliza-se da segunda categoria supracitada: aplicações multimídia que viabilizam conteúdo multimídia exibido sob demanda de um usuário. Como este conteúdo multimídia, em geral, é composto de vídeos, chamaremos esta categoria de vídeo sob demanda, ou VoD (*Video On Demand*). Exemplos desta categoria são locadoras de filmes online, mundo virtual, ensino a distância, etc.

Em nosso serviço de vídeo sob demanda para ensino a distância, temos:

- o início assíncrono dos pedidos aos objetos que os clientes desejam;
- um repositório de objetos multimídia;
- cada cliente possui um *buffer* para armazenar blocos recebidos antes de seu momento de exibição, ao qual chamaremos de *playout buffer*;
- a possibilidade de se alterar o ponto de exibição do vídeo através de operações de VCR como avançar, retroceder, parar, etc;
- um nó que funcionará como índice do conteúdo multimídia disponível;
- um ou mais nós que irão, efetivamente, armazenar todo conteúdo multimídia disponível.

O cliente necessita ter um *playout buffer* conforme descrito acima por causa da variabilidade dos atrasos existentes nas redes de pacotes. O tamanho do *playout buffer* é decidido através do compromisso entre a menor probabilidade do *playout buffer* esvaziar e o maior tempo de espera inicial para o começo da reprodução do vídeo (*startup delay*).

1.4 Interativo x Ao Vivo: Problemas inerentes a cada tipo de serviço

As aplicações que utilizam vídeo e áudio como principal mídia de veiculação são sensíveis a retardo, mas exibam tolerância a perda e a latência. A garantia determinística de retardos e outras métricas implica na garantia de banda. Porém isto iria de encontro com os quatro princípios da arquitetura da Internet, que são: autônomo e minimalista (uma rede deve ser capaz de operar sem alterações quando se conectar em outra rede), serviço sem garantias para entrega de pacotes (*best-effort*), roteadores que não guardam o estado dos fluxos de dados e controle descentralizado. Portanto, a entrega do pacotes na Internet opera sem todas as garantias que seriam necessárias para um serviço de *streaming* de vídeo. Desta forma, as aplicações devem implementar algum mecanismo que forneça a qualidade mínima exigida em uma rede com estas características.

Como a Internet fornece, em geral, serviços básicos de entrega de pacotes, as técnicas de qualidade de serviço (QoS) devem se situar nas bordas do sistema. Ou seja, a mudança deve estar nas aplicações, no cliente e no servidor, e no protocolo de comunicação entre eles, de acordo com o tipo de serviço que se deseja prover: interativo, com vídeo sob demanda (VoD) ou uma transmissão ao vivo.

Um serviço de vídeo sob demanda consiste, de maneira geral, em um servidor multimídia que contenha um repositório de material multimídia e um índice do mesmo. Então é possível que clientes cheguem, examinem o índice e comecem a ver o conteúdo de seu interesse de forma totalmente aleatória e independente. Um serviço de transmissão ao vivo é composto por um servidor que transmite um conteúdo multimídia específico durante um tempo específico. Os clientes que chegam em tempos aleatórios estão interessados apenas no conteúdo que está sendo transmitido a partir do momento de sua chegada.

É importante ressaltar as diferenças entre um serviço VoD interativo e um serviço de transmissão de vídeo ao vivo. Na transmissão ao vivo, quanto mais curto for o atraso fim-a-fim, mais "real"a transmissão parecerá para seus usuários. Isto é definido como *liveness* por [1]. Já na transmissão de vídeo sob demanda, *liveness*

Tabela 1.1: Diferenças entre um serviço de vídeo sob demanda (VoD) e um serviço de transmissão ao vivo.

	VoD	Ao Vivo
Atraso fim-a-fim (*liveness*)	Sensível	Sensível
Tolerância a períodos de QoS baixo	Baixa, clientes deixam o sistema	Alta
Compartilhamento de banda	Complexo	Simples

é irrelevante, pois o vídeo já foi pré-gravado. Por outro lado, o atraso fim-a-fim em um serviço VoD afetará o atraso inicial (*startup delay*) para execução do vídeo e também o atraso para comandos de tipo VCR, como avançar, retroceder, mover para algum outro ponto do vídeo, etc. Um alto atraso fim-a-fim diminui a percepção de interatividade pelo usuário, pois estas operações terão um grande atraso para serem executadas.

Uma segunda diferença entre estes dois tipos de serviço consiste em que um novo cliente de uma transmissão ao vivo só estará interessado no fluxo multimídia a partir do momento em que se conecta. Já em um serviço de vídeo sob demanda, um novo cliente estará interessado no vídeo inteiro. Portanto, o compartilhamento de banda no serviço de transmissão ao vivo é bem mais simples quando comparado com um serviço VoD. Por exemplo, em um serviço VoD, um novo cliente poderá armazenar o fluxo de um cliente anterior, mas por outro lado deverá existir alguma maneira de entregar a parte inicial faltosa a este cliente.

Por fim, as correlações entre as diversas variáveis existentes são diferentes para cada tipo de serviço. Por exemplo, quando a qualidade de serviço (*Quality of Service* - QoS) degrada, um cliente de vídeo sob demanda é mais provável de interromper o serviço do que um cliente de um vídeo ao vivo. Isso porque ele ou ela não terá a opção de assistir ao vídeo novamente no futuro, o que aumenta a sua tolerância a períodos de baixa QoS. Assim, ao cair a QoS de um serviço VoD, muito mais clientes vão deixar o sistema do que em um serviço de transmissão ao vivo. Isto foi observado em [2]. Portanto, a necessidade de se ter um servidor VoD tolerante a falhas e com algum tipo de controle de admissão de usuários é muito mais importante do que no serviço ao vivo, pois é muito mais importante, neste caso, manter uma QoS aceitável para todos os usuários. Algumas das diferenças podem ser sumarizadas na tabela 1.1.

1.5 Organização do trabalho

Este trabalho está organizado da seguinte maneira: O Capítulo 2 possui um breve apanhado geral sobre arquiteturas propostas na literatura para *streaming* de vídeo na Internet. No Capítulo 3 encontra-se uma visão geral da arquitetura do RIO, o servidor utilizado neste trabalho. A descrição dos experimentos, bem como a descrição mais pormenorizada da arquitetura de redes utilizada, os testes realizados e seus respectivos resultados estarão no Capítulo 4. Por fim, a conclusão e os trabalhos futuros relacionados se encontram no Capítulo 5.

Capítulo 2

Principais Arquiteturas para *Streaming* Multimídia

NESTE capítulo revisaremos de forma geral vários estudos publicados na literatura de arquiteturas para *streaming* multimídia. A forma de categorização das propostas constitui parte original deste trabalho e está sumarizada na tabela 2.2. Ao apresentar as idéias propostas, iremos discorrer brevemente sobre algumas delas para tentar apontar os seus principais problemas e indicaremos qual a arquitetura que seria mais apropriada para um serviço de educação a distância.

2.1 Quanto a função dos agentes

Aqui descreveremos sobre as arquiteturas propostas no tocante a função de cada agente do sistema e como os recursos disponíveis são utilizados pelos mesmos.

2.1.1 Arquiteturas VoD

A primeira arquitetura proposta para se fazer *streaming* multimídia é termos um único servidor para indexar o material multimídia, atender os pedidos dos clientes e enviar os dados para cada um, como visto em [3] ou em [4], por exemplo. Esta arquitetura é muitas vezes chamada de servidor de vídeo sob demanda ou VoD (*Video on Demand*) e pode ser ilustrado na Figura 2.1.

O primeiro grande problema de arquiteturas VoD é a escalabilidade. Por ex-

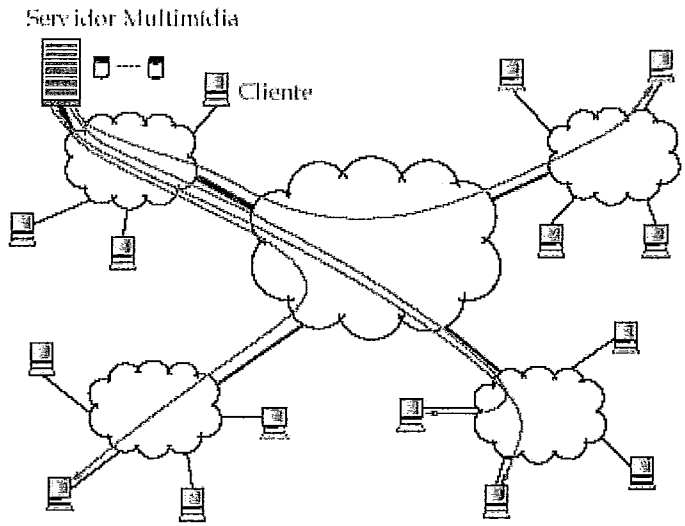

Figura 2.1: Exemplo de uma rede com um servidor multimídia único.

emplo, tal qual em [5], consideremos que um servidor VoD possua apenas filmes codificados em MPEG-2, com uma tela de 320 x 240 *pixels*. Em geral, filmes neste formato precisam de uma banda de 1.5 Mbps (mega bits por segundo). Assim, como também analisado em [6], considerando que temos um canal de aproximadamente 1 Gbps, seria possível servir, aproximadamente, 666 clientes, dado que consigamos 100% de utilização do canal, o que já seria impossível. Mesmo assim, neste cenário teórico, o custo de implantar este sistema, repassado a tão pequena quantidade de usuários simultâneos servidos, o tornaria economicamente inviável. Entretanto, existem técnicas de compartilhamento de banda que podem ser utilizadas para solucionar este problema, conforme veremos na seção 2.2.

Além disso, outro problema inerente às arquiteturas VoD é existir um ponto único de falha: se este servidor falhar, todo o sistema também falhará. Dado que, para ser minimamente efetivo, um servidor VoD necessita de dezenas a milhares de discos rígidos, o tempo médio até a falha ou MTTF (*Mean Time To Failure*) de um destes discos torna-se alto. Por exemplo, segundo [3], com um sistema com 1000 discos, o tempo médio até a falha de um deles é de 300 horas ou aproximadamente 12 dias.

2.1.2 Arquiteturas DVoD

Para resolver o problema de escalabilidade de servidores VoD anteriormente mencionado, a solução mais evidente é particionar e distribuir este servidor pela rede, separando a função de armazenamento de conteúdo da função de indexação do mesmo. Em [7] é visto que esta abordagem torna possível ter uma arquitetura para *streaming* multimídia escalável. Estas arquiteturas com algum grau de descentralização são comumente chamadas de servidores VoD distribuídos ou DVoD (*Distributed Video on Demand*). Agrupamos, nas próximas subseções, as propostas na literatura que utilizam tal abordagem.

Proxies: descentralização parcial

Uma das sugestões encontrada mais freqüentemente na literatura é a utilização de proxies, como encontrado em [8] [9] [10] [11] [12]. A principal idéia é assumir que os clientes estejam localizados em grupos concentrados, com gargalo único. Assim, é possível colocar um proxy na "borda"destas redes, neste gargalo, a fim de mitigar o acesso ao servidor VoD, conforme ilustrado na Figura 2.2.

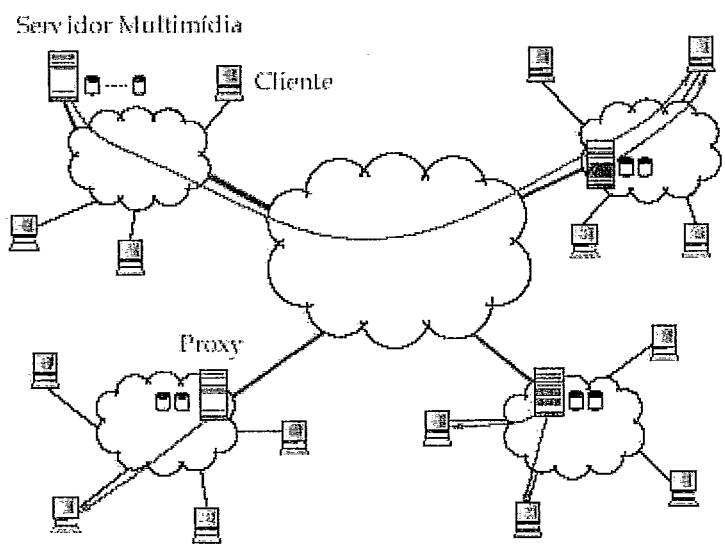

Figura 2.2: Exemplo de uma rede com proxies perto dos grupos de clientes.

Todavia, novos problemas surgem com a utilização de proxies. A primeira pergunta que surge ao ser sugerida a adoção de proxies é de que maneira irá ser

preenchido o seu cache, uma vez que não seria efetivo em termos de custo termos proxies tão robustos quanto um servidor VoD [7].

As sugestões na literatura são extremamente díspares no tocante a este problema. Diversas sugestões são analisadas e comparadas em [12]. Podemos categorizar as sugestões no armazenamento:

- do prefixo dos filmes, a fim de diminuir a latência inicial [9];
- das partes dos filmes que possuam maior taxa de informações e que, portanto, causem rajadas na rede por possuirem uma taxa de consumo maior (*staging cache*) [10];
- de uma seleção não contígua de blocos intermediários, para auxiliar operações de VCR como avanço rápido (*fast forward*) ou retrocesso rápido (*rewind*), por exemplo [11];
- de todos os filmes, perfazendo um *mirror* distribuído. Dividem-se os proxies em uma árvore binária, em que cada nível representa uma hierarquia. Os proxies de um mesmo nível devem ou armazenar os filmes mais populares ou compor um *mirror* distribuído armazenando a única cópia existente de filmes não populares. A Figura 2.3 ilustra esta sugestão [8].

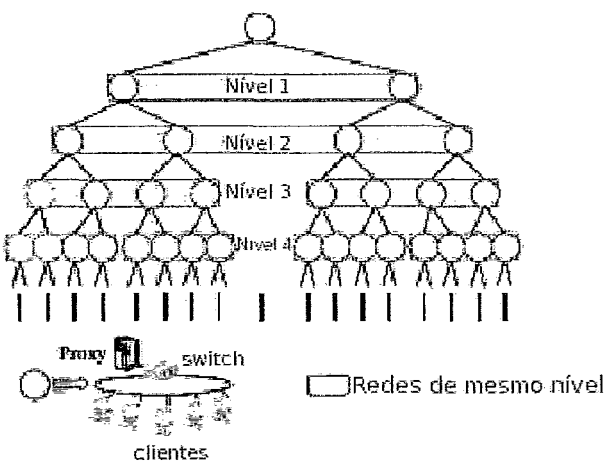

Figura 2.3: Exemplo de uma rede com proxies hierárquicos.

Seja qual for a forma de armazenamento escolhida, é possível obter-se um aumento da capacidade total do sistema em número de usuários servidos dado um mesmo *goodput* [7]. Entretanto, o problema de termos um ponto único de falha ainda permanece, uma vez que é preciso recorrer ao servidor VoD quando um proxy não possui um bloco requisitado por um cliente sendo servido.

CDNs: marketing ou ganho real?

As redes de distribuição de conteúdo ou CDNs (*Content Distribution Network*) foram anunciadas nos meios de comunicação como a solução definitiva para distribuição de conteúdo multimídia na Internet. Empresas como a Akamai [13] e a Digital Island [14] se empenham em convencer o público em geral que a sua tecnologia proprietária para CDNs é o estado da arte nesta área. Entretanto, estudos como [15] mostram que o balanceamento de carga destas CDNs está longe do ótimo. Dado um pedido de um objeto por um cliente, na maior parte dos casos, o roteamento destas CDNs apenas evita que ele seja direcionado para o pior servidor, ou seja, o servidor com maior carga que possui tal objeto, ao invés de direcioná-lo para o melhor servidor possível.

Apesar das CDNs possuírem tecnologia fechada, é possível inferir seu funcionamento através de testes baseados em engenharia reversa, como feitos em [15]. Tipicamente, uma CDN é composta de milhares de servidores, conforme a Figura 2.4 mostra. Por exemplo, a Akamai possui aproximadamente 10,000 servidores, segundo [13]. Eles são instalados muito próximos dos clientes em potencial, em geral dentro dos próprios grandes provedores de acesso à Internet ou ISPs (*Internet Service Provider*). Os objetos possuem muitas réplicas, a fim de viabilizar o balanceamento de carga. O principal problema é saber como rotear um dado pedido para um "bom"servidor, onde "bom"significa que a réplica irá chegar com a menor latência possível para o cliente.

Uma CDN pode ser encarada como um servidor VoD com proxies que possui características específicas. Por exemplo, as CDNs costumam direcionar os pedidos de clientes não atendidos para seu interior. Ou seja, o seu roteamento tenderá a balancear a carga através do direcionamento dos pedidos das folhas para nós superi-

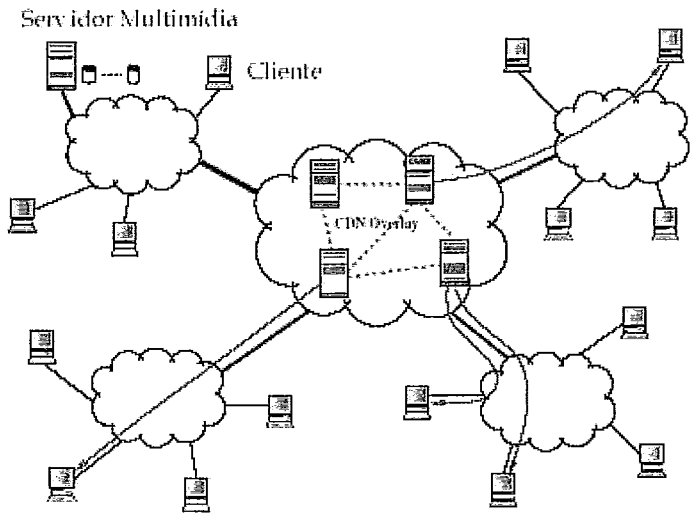

Figura 2.4: Exemplo de rede com a distribuição de conteúdo multimídia feito por uma CDN.

ores. Menos significativamente, podemos também ressaltar que as CDNs procuram fazer a população de seus caches proativamente, a fim de antecipar uma demanda futura [16].

Em geral, somente informações geográficas ou a topologia da rede até o cliente não são suficientes para determinar um bom servidor. É necessário incorporar a esta decisão informações dinâmicas sobre o estado da rede e a carga instantânea de cada servidor que possui uma réplica. Aparentemente, isto não é feito nas CDNs através da utilização de balanceamento via resolução de DNS [15]. Como se pode esperar, este balanceamento está longe de ser ótimo, apesar de apresentar ganhos em termos de cache cooperativo [16]. Como possuímos maior controle sobre os agentes do sistema, a abordagem não é interessante para um sistema de ensino a distância. Devemos procurar um balanceamento de carga mais eficiente para nossa aplicação.

2.1.3 Arquiteturas Peer-to-Peer

Apesar da arquitetura DVoD possibilitar a escalabilidade que falta à arquitetura VoD centralizada, um sistema ainda terá capacidade limitada pelo número de servidores adicionados ao sistema. Para resolver esta limitação um sistema deveria ter sua capacidade naturalmente crescente, de acordo com o aumento do número de

usuários. E isto só será possível se cada cliente também contribuir para a capacidade total do sistema. Esta é a motivação desta subseção, que agrupa sugestões na literatura para a utilização de arquiteturas *peer-to-peer* (P2P).

Por ser uma arquitetura promissora, muito se pesquisou no sentido de se utilizar P2P para VoD [17] [18] e transmissões ao vivo [1] [19]. Entretanto, é notório a elasticidade do emprego do termo P2P. Por isso, é comum a confusão deste paradigma com outros, como a arquitetura para computação *Grid* [20], por exemplo. Portanto, é vital tentarmos definir estes termos, pelo menos no escopo deste trabalho, para melhor distinção entre os mesmos.

Definição de uma arquitetura *peer-to-peer*

Um dos motivos pelos quais o termo P2P é amplamente utilizado se deve ao fato de que aplicações são rotuladas como tal não devido ao seu funcionamento interno, mas sim pela maneira que elas são percebidas externamente, ou seja, se elas dão uma percepção de prover uma interação direta entre computadores. Por isso que diversas aplicações são rotuladas como P2P, apesar de não funcionarem desta maneira, como é o caso do pioneiro Napster, por exemplo. Tendo isto em vista, uma definição para P2P bastante apropriada para nossos fins é a encontrada em [21]:

"Sistemas peer-to-peer são sistemas distribuídos que consistem em nós interconectados, capazes de se auto-organizarem em uma topologia de rede com o propósito de compartilhar recursos como conteúdo, ciclos de CPU, capacidade de armazenamento e banda, que podem se adaptar a falhas e acomodar populações transientes de nós, mantendo uma conectividade e performance aceitáveis, sem necessitar a intermediação ou o suporte de uma autoridade ou servidor global centralizado."

Definição de uma arquitetura com o uso de computação *Grid*

Desde a criação do termo *Grid computing* por Ian Foster, Carl Kesselman e Steven Tuecke [20], muito foi pesquisado com o objetivo de se construir as fundações deste novo paradigma [22] [23] [24]. Entretanto, a definição do que seria exatamente um *Grid* se tornou obnubilado devido a utilização maciça do termo, da mesma forma que aconteceu com "*peer-to-peer*".

Tabela 2.1: Diferenças básicas entre arquiteturas *peer-to-peer* (P2P) e *Grid*.

	P2P	*Grid*
Transiência dos nós	Alta	Baixa ou Nenhuma
Número médio de nós	Grande	Pequeno
Capacidade média dos nós	Pequena	Grande
Serviços ofertados	Populares e Simples	Especializados e Complexos

Assim sendo, em [20], encontramos uma definição de qual seria o objetivo original de *Grid computing*:

"O problema real e específico que encabeça o conceito de Grid computing é o compartilhamento de recursos coordenados e a resolução de problemas em organizações virtuais multi-institucionais."

Grid x *Peer-to-Peer*

Apesar da tentativa de definição de ambas arquiteturas anteriormente, ainda podemos ter alguma dificuldade em sua clara diferenciação. Em [25], esta distinção é feita. Basicamente é argumentado que, apesar de aparentemente similares, arquiteturas P2P e *Grid* são diferentes em diversos aspectos cruciais. Arquiteturas P2P costumam prover serviços muito mais populares e simples, como compartilhamento de músicas, por exemplo, do que *Grids* tradicionais, como o Grid da NASA [23], que prove serviços computacionalmente complexos. Assim, P2P costuma ser composto de uma massa muito maior de usuários com poucos recursos e pouca demanda, enquanto que *Grids* são o exato oposto, com poucos usuários, muitos recursos e demanda por serviços complexos e especializados. Por fim, ainda é notório que nós P2P são muito mais transientes do que nós de *Grids*, o que obriga a arquiteturas P2P sempre ter algum tipo de recuperação a falhas. Em contrapartida, arquiteturas de *Grids* podem até assumir que seus nós nunca se desconectarão, se necessário. Esta discussão está sumarizada na Tabela 2.1.

Arquiteturas P2P quanto à organização e função dos *peers*

Arquiteturas *peer-to-peer* propostas na literatura costumam organizar os *peers* de duas grandes formas genéricas: ou por *clusters* que transmitem em *unicast* e particionam a taxa necessária por cada nó que possui o filme [6] [26] [27] ou em

árvores de transmissão *multicast*, construídas de diversas maneiras [17] [28] [18].

No trabalho de [6], é proposto o agrupamento dos nós por *clusters* construídos por proximidade. Dentro de um cluster, os nós assumem papéis diferentes, de acordo com a sua capacidade de transmissão e baixa transiência. Estes nós são eleitos líderes de seus *clusters* e tem responsabilidade extra em indexar o conteúdo dos nós em seus grupos, identificar e organizar os nós que estejam disponíveis para transmissão ao chegar um novo pedido de *streaming* de algum vídeo e organizar a dispersão pelos nós de um novo vídeo. A taxa necessária para cada transmissão é particionada de acordo com a capacidade de cada nó, tal qual visto na Figura 2.5.

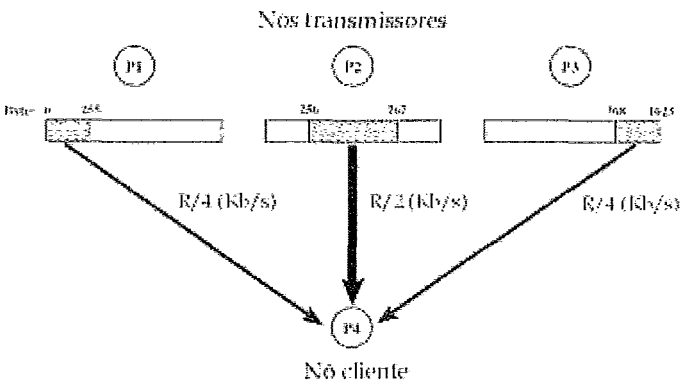

Figura 2.5: Exemplo da partição do fluxo de acordo com a capacidade dos nós. Da taxa total R necessária, o nó P2 tem mais banda disponível, logo ele é responsável por transmitir R/2 enquanto que os dois outros transmitem R/4.

Já em [26], o agrupamento é feito por uma lista encadeada que é chamada de *forward chain*. Esta lista funciona da seguinte forma: um nó nesta lista transmite para seu filho que, por sua vez, transmite para seu neto e assim por diante. Um nó com maior capacidade e que possui o vídeo inteiro é eleito servidor e ele tem as seguintes atribuições: é responsável por ser a raiz da lista encadeada, por transmitir um *stream* extra, com *Forward Error Correction* (FEC) toda vez que o buffer de algum nó em sua lista encadeada estiver abaixo de um *threshold H* pré-definido e reorganizar a lista de acordo com um mecanismo chamado de *Parent-Child Exchange* (PCX). O PCX funciona de acordo com o seguinte algoritmo: cada vez que um nó percebe que o seu pai está transmitindo a uma taxa menor do que ele, seu filho, está

consumindo, ele aguarda até o seu *buffer* esvaziar abaixo de um *threshold H* dado. Quando o *buffer* esvazia além deste *threshold H*, este nó conclui que está em uma condição de rede melhor do que seu pai. Assim, passa a receber a transmissão de seu avô e passa a transmitir para seu pai. O mesmo acontece no caso de seu pai sair da lista por qualquer motivo. Assim, a lista encadeada é organizada de acordo com a capacidade de transmissão dos nós. Inicialmente, um novo nó recebe a transmissão do último nó na lista e do servidor, pois seu buffer está vazio e abaixo do *threshold H*, o que diminui a latência inicial.

Em [27], é utilizado o algoritmo de *Reed-Solomon Erasure Correction* (RSE) [29] na transmissão dos fluxos para recuperação da falha de algum nó em um cluster, que foi agrupado por proximidade. E é feita a transmissão de um dado fluxo multimídia para um nó em um cluster particionando a banda necessária para tal entre seus membros. Apesar dos métodos acima descritos atigirem uma suficiente eficiência, alguns problemas permanecem a serem resolvidos, como a determinação e eleição de *peers* com maior capacidade e a dependência de todo o sistema nestes nós especiais.

Outra forma de agrupamento de *peers*, que também é popular, é o agrupamento por árvore de transmissão *multicast*. Em [17] [28] [18] é proposta a formação da árvore de *multicast* através do estabelecimento de gerações, onde cada geração é um agrupamento de nós, determinada pelo vídeo escolhido e o ponto no tempo no qual um nó fez a requisição para juntar-se a árvore. Assim, com base nestas duas informações, seria determinado qual geração o nó deve ser agrupado. Cada geração tem uma tolerância para a entrada de um nó, de acordo com uma janela de tempo deslizante. Assim, a parte faltante pelo nó ter chegado atrasado a uma dada geração, mas dentro da janela deslizante, seria recebida através de múltiplas transmissões *unicast* dos outros nós desta árvore. Portanto, este novo nó desta geração teria um cache do tamanho de tempo entre a janela deslizante inicial e o ponto em que se juntou a esta geração.

Árvores de *multicast* são eficientes para transmissões ao vivo e VoD com baixa interatividade, mas ao usuário saltar de um ponto de um vídeo para outro, há um inevitável atraso, que é maior do que em sistemas que se utilizam de *unicast* como forma de transmissão. Isto porque um salto no vídeo requer que toda a árvore de

multicast construída para aquela transmissão seja refeita, pois os nós que estavam transmitindo não necessariamente possuem o novo trecho pedido. Isto acaba por inserir um maior atraso na interatividade do usuário, o que não é desejado em sistemas de ensino a distância.

Arquiteturas P2P quanto à distribuição dos dados

Para arrumação dos dados dos vídeos em arquiteturas P2P, temos duas formas recorrentes dentre as várias propostas de arquiteturas P2P na literatura: a primeira consiste em se fazer *striping* entre os nós de um cluster [27] e a segunda armazena os vídeos indivisivelmente, só variando a forma em que é feita o cache dos mesmos. Como *striping* será explicado na subseção 2.3.1, nos deteremos aqui na segunda forma citada.

A primeira e mais simples forma de se fazer *cache* dos vídeos durante uma transmissão é armazená-los por completo a medida que são recebidos, conforme feito em [26]. Outra forma é fazer um cache parcial temporário, tal qual uma janela deslizante, de acordo com [28] e [17]. Destas duas formas, temos como recuperar de falhas localmente, sem necessitar recorrer a um líder, ou seja, um servidor de um *cluster* ou uma raiz de uma árvore.

Uma terceira forma de se arrumar os dados consiste em fazer o cache do início ou prefixo do vídeo e isto é encontrado em [18] e [6]. A idéia principal de Hefeeda em [6] é criar um buffer grande o suficiente para suportar oscilações nas taxas de transmissão e transiência dos *peers*. Em seu trabalho ele conclui que um buffer inicial de 10 a 20 segundos de vídeo é o suficiente para assegurar qualidade total à transmissão, mas ele não considera operações de VCR em seu estudo, o que degradaria muito a performance, pois a cada operação teríamos de esperar o buffer ser preenchido novamente. Já em [18], o objetivo do cache inicial é agrupar clientes nas mesmas sessões *multicast* e fazer a transmissão por *unicast* da porção inicial perdida, tal qual a técnica conhecida como *Patch*. Entretanto, a diferença entre a técnica proposta e o *Patch* tradicional é que a transmissão do *patch* é também feita via *multicast*. O problema com esta abordagem é a sua inaptitude para sistemas de ensino a distância, que possuem maior interatividade e, portanto, maior quantidade de operações de

VCR, o que degradaria sua performance. Uma solução para esta limitação do *Patch*, chamada de *Patching Interativo*, foi inicialmente estudada em [30], mas em um contexto de um servidor DVoD. Em [31], é encontrado um estudo mais detalhado da técnica *Patching*, que inclui um modelo analítico para o cálculo da distribuição da banda do servidor VoD. Além disto, em [31], também são propostas três novas técnicas de compartilhamento de banda: duas delas baseadas em *Patching* e a outra no paradigma de HSM (*Hierarchical Stream Merging*). Os resultados de simulação comprovam a eficiência satisfatória das técnicas propostas.

Problemas específicos às arquiteturas P2P

A abordagem *peer-to-peer* aparenta ser extremamente promissora em face de suas concorrentes, por trazer escalabilidade sem limites a um sistema. Contudo, ela também possui problemas peculiares a sua natureza. Trataremos de alguns deles a seguir.

Um problema ainda sem solução definitiva é como iniciar uma rede P2P que não possui nenhum tipo de líder ou servidor central, como no Gnutella [32], por exemplo. Uma solução é possuir uma lista de *peers* que estão sempre conectados, acessando-a através de algum *site* ou que seja distribuída com a aplicação que acessa determinada rede P2P. Na Figura 2.6 podemos ver o exemplo de uma rede que utiliza esta técnica e que é organizada em dois níveis, de acordo com a capacidade dos *peers*.

Outro problema é como escolher os nós que serão vizinhos uns dos outros, pois isto influencia a topologia da rede *overlay* que um sistema P2P intrinsecamente cria. Em [33], é mostrado como usar *Random Walks* para estabelecer o grafo bem conectado da rede *overlay*, ou seja, um grafo que seja k-conexo, com $k > 1$. Quando um nó que vai se juntar a rede acha outro nó já conectado, ele executa um algoritmo de conexão com a rede, de forma que os nós que são retornados como possíveis vizinhos são quase uniformemente escolhidos, a fim de rearrumar a topologia da mesma. Assim, é obtida uma alta probabilidade de que a rede permaneça bem conectada. Neste mesmo artigo também é discutido como fazer com que os nós se desconectem sem que o grafo da rede perca esta propriedade.

Também vem sido debatido o comportamento dos nós em redes P2P. É claro

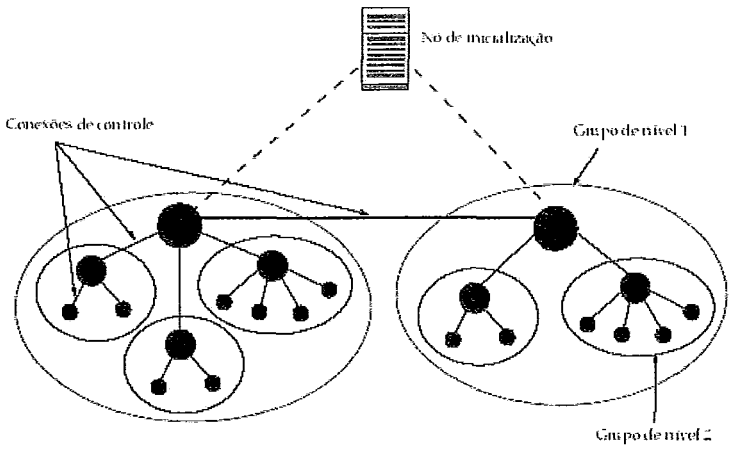

Figura 2.6: Exemplo de uma rede P2P que inicializa com um nó de inicialização (*bootstrap*) e que é organizada em dois níveis, de acordo com a capacidade dos nós. Os círculos preto maiores indicam nós com maior capacidade.

que, para que uma rede P2P possa funcionar propriamente, os nós precisam cooperar para o compartilhamento de seus recursos. Por isso que a maioria dos sistemas P2P assumem que esta cooperação existe, confiando aos nós tarefas que nem sempre possuem um benefício individual direto. Entretanto, quando sistemas grandes e abertos são implantados, esta suposição de cooperação não permanece verdadeira porque podem surgir nós com um comportamento aparentemente egoísta, por só utilizar-se de recursos da rede sem contribuir. Um exemplo disto são redes de distribuição de arquivos como o Kazaa ou o Gnutella onde foi mostrado em [34] que existem muito usuários que não compartilham seus próprios arquivos, se comportando, assim, de forma egoísta para com os outros nós da rede.

O tempo de permanência dos nós em uma rede P2P é muito específico ao tipo de serviço que ela provê e como o nó interage com a mesma. Por exemplo, em uma rede de compartilhamento de arquivos como Gnutella, o tempo médio de permanência de um nó é de 60 minutos [35]. Já em uma rede como o Bittorrent, o tempo de permanência pode variar de 3 minutos a 3 meses [36].

Finalmente, um problema muito debatido é como indexar o conteúdo da rede P2P e as principais abordagens são sumarizadas em [37]. Existem três grandes métodos:

- termos o índice inteiro em um nó ou inteiramente replicado em diversos nós;
- construir o índice somente com base do conteúdo de um grupo, que é definido como todos os nós distantes um número x de saltos do *peer* que funciona como servidor de índice local. Estes servidores de índice locais se intercomunicam para formar toda a rede;
- não termos índice;

O primeiro método é o mais simples, mas também o com menor confiabilidade, por ter um ou poucos pontos únicos de falha, que são os *peers* que funcionam como servidores de índices. O segundo método consiste em distribuir o índice em grupos cujo tamanho é de x saltos do nó que funciona como este índice local. Estes nós de índice local funcionam de maneira conjunta para formar o índice de toda a rede. Além do problema de eleição de líder deste grupo, o problema desta abordagem é quando o servidor de índice local se desconecta da rede, o que retira todos os nós que estavam sendo indexados da rede P2P até que um novo líder seja eleito e seu índice reconstruído. A terceira abordagem não possui estes problemas anteriormente citados, mas por outro lado é preciso realizar um *flood* em toda a rede para cada busca, conforme exemplificado na Figura 2.7.

De um modo geral, arquiteturas P2P são pouco adequadas a sistemas de ensino a distância por dois motivos: transiência dos nós e de conteúdo. Um sistema de ensino a distância necessita que todas suas aulas estejam disponíveis todo o tempo e com a mesma qualidade de serviço: sistemas P2P não dão nenhuma garantia de disponibilidade em tempo integral de algum material específico. Também não temos garantia de disponibilidade eqüânime de todas as aulas. Assim, pelo o que foi exposto até agora, sistemas baseados em DVoD são mais indicados para o objetivo deste trabalho, a aplicação no ensino a distância.

2.2 Quanto à forma de *streaming*

Existem duas filosofias para servir clientes em um servidor VoD ou DVoD: ou eles são servidos de acordo com o que requisitam (*client-pull*) ou apenas consomem

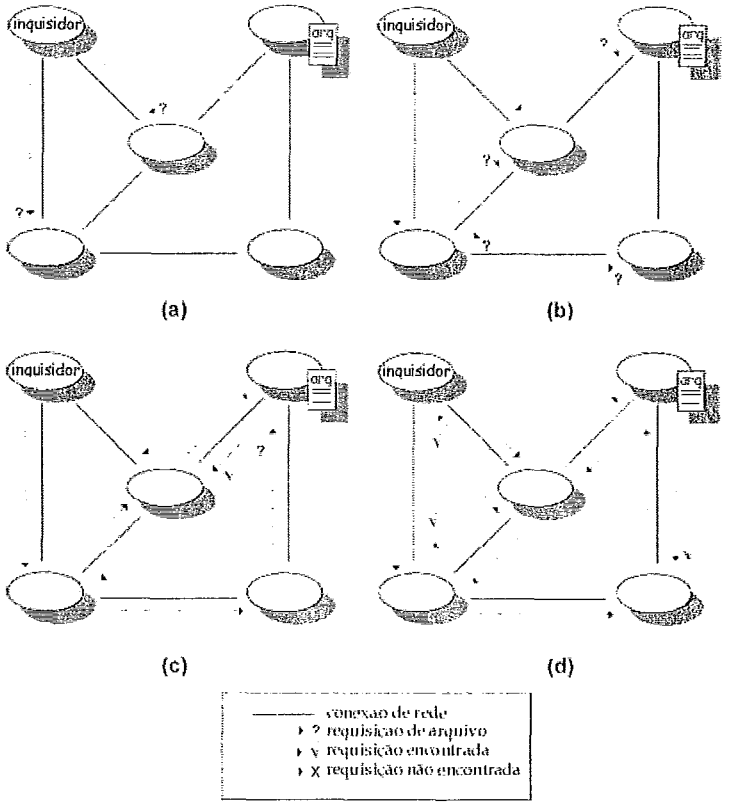

Figura 2.7: Exemplo de uma execução de um algoritmo de flood como o encontrado no Gnutella. Em (a) o nó inquisidor manda para todos seus vizinhos a sua requisição. Em (b) os nós vizinhos repassam a requisição para seus vizinhos, uma vez que eles não possuem o arquivo procurado. Em (c) é encontrado o arquivo e a resposta é enviada. E em (d) a resposta é repassada ao nó inquisidor original.

o que o servidor transmite (*server-push*). Inicialmente, a primeira filosofia condiz mais com técnicas de *unicast*, enquanto que a segunda se adequa mais facilmente com *multicast*. A *priori*, é comum pensar que a primeira filosofia é mais apropriada para a transmissão de vídeos interativos como ensino a distância e a segunda filosofia seria mais útil na transmissão de programas ao vivo ou videoconferências. O ideal seria que fosse possível utilizar *multicast* para prover serviços interativos. Existem sugestões na literatura com este fim e um apanhado delas pode ser encontrado em [30] e em [31].

Uma última sugestão encontrada na literatura quanto à forma de *streaming* é a utilização de redes ativas, conforme visto em [38]. Naquele trabalho é sugerida

a utilização de um novo codec baseado em *wavelets*, em detrimento ao MPEG, em conjunto com um mecanismo de controle de qualidade de cada fluxo implantado nos roteadores. Este mecanismo diminui a qualidade do vídeo de acordo com a carga nos roteadores e o novo codec possibilita uma maior granularidade deste controle do que utilizando MPEG. Com o uso de todas estas sugestões, em [38] há um ganho de qualidade no vídeo do cliente de 15 dB PSNR ao auferir-se a luminosidade do mesmo. Entretanto, a utilização deste controle de qualidade por fluxo implicaria na homogeneização dos roteadores existentes na Internet, a fim de que eles fossem capazes de ser expansíveis com este algoritmo de controle. Infelizmente, tal padronização é muito difícil de se obter em um mundo cada vez mais heterogêneo.

2.3 Quanto à forma de alocação dos vídeos

O método de alocação dos vídeos nos discos de um dado sistema é crucial para seu funcionamento eficiente. Esta alocação tem impacto direto no seu balanceamento de carga e, portanto, em sua eficiência na utilização de seus recursos. Nas próximas subseções, iremos tratar dos dois principais aspectos de alocação dos vídeos, que é quanto à forma em que os blocos são dispersos nos discos e quanto à forma de replicação dos vídeos no sistema.

2.3.1 Formas de dispersão dos blocos nos discos

As formas de alocação dos blocos dos vídeos se concentram em torno de dois grandes grupos, além da óbvia alocação indivisível de um vídeo, como em [8] [39] [40] [41]. O primeiro e mais popular grupo usa *striping* como base de suas propostas. Esta técnica consiste em se espalhar blocos do vídeo em alguma dada seqüência. Por exemplo, as sugestões de uso de *striping* mais comuns falam em se criar uma seqüência dos discos e, então, distribuir os blocos do vídeo seguindo esta seqüência, voltando ao primeiro disco ao se chegar ao último [42] [43] [3] [4]. Assim, deseja-se aproveitar a seqüencialidade de acesso de um vídeo de um filme para a distribuição de carga uniformemente entre os discos.

O *striping* pode ser feito de duas formas: local e total. *Striping* local consiste

em se distribuir os blocos de um dado vídeo apenas nos discos de um nó de armazenamento conforme descrito anteriormente. O *striping* total é fazer o mesmo, só que considerando todos os discos do sistema, não importando em quais nós eles se encontram. [42] faz uma comparação entre ambos e sugere um sistema híbrido, em que réplicas de vídeos populares utilizam *striping* local e os demais vídeos utilizam *striping* total. Por fim, é válido ressaltar que, em [43], compara-se a utilização do *striping* tradicional com o *staggered striping*. Este último consiste em agrupar os discos em conjuntos e distribuir os blocos dos vídeos também em uma seqüência, mas espaçada por um dado k. Entretanto é visto que os ganhos de se fazer isto são reduzidos e específicos a dados cenários, não compensando sua maior complexidade de implementação se utilizado de forma mais genérica.

Apesar da utilização de *striping* ser bastante popular nas propostas de arquiteturas de sistemas distribuídos por ser extremamente convincente intuitivamente, esta técnica possui diversas limitações. Por exemplo, a grande vantagem que o *striping* trás é de se aproveitar da seqüencialidade do acesso de um cliente aos blocos de um filme, uma vez que um filme é visto do começo até o fim. Entretanto, se quisermos utilizá-lo para outras aplicações multimídia em voga, como, por exemplo, ensino a distância, cuja taxa de operações VCR como avanço ou retrocesso rápido é significativamente mais alta que em outras aplicações, a performance de *striping* é sensivelmente afetada [5].

Para tratar estes problemas, em [44] [45] [46] [47] [48] é utilizado a alocação randômica dos blocos do vídeos pelos discos, uniformemente distribuídos. Em [48] é mostrado que a alocação randômica dos blocos é, no pior dos casos, tão boa quanto a utilização de *striping* e ainda temos uma implementação mais simples para a alocação dos blocos, bem como a ausência de fragmentação nos discos, pequena probabilidade de fragmentação prolongada de banda, etc. Apesar de todos estas vantagens em relação a *striping*, a alocação randômica ainda é muito pouco explorada nas propostas encontradas na literatura.

2.3.2 Formas de replicação dos vídeos

A replicação dos vídeos se tornou quase que uma obrigatoriedade em qualquer trabalho que trate de servidores multimídia tolerantes a falhas. Além disso, a replicação permite o balanceamento de carga. Em [49] encontramos uma detalhada análise de diversas variações de modelos de muliservidores com filas nos quais os clientes podem investigar randomicamente (*random probing*) d servidores e, então, entrar na menor fila auferida. Os resultados lá encontrados mostram que uma melhora exponencial no tempo médio de espera dos clientes é obtida em sistemas que possuem várias escolhas de quais filas um cliente quer entrar do que em sistemas com escolha única. Em particular, a maior parte dos ganhos são obtidos quando os clientes só possuem duas escolhas. Assim, como o problema de rotear um pedido para o nó menos carregado pode ser mapeado neste problema de investigação randômica, a utilização de replicação tem grande potencial para o balanceamento de carga.

Podemos citar os trabalhos [50] [39] [51] em que são sugeridas diversas formas de replicação, tanto para aumentar a confiabilidade de seus sistemas bem como para aumentar sua eficiência. Na literatura, temos três tipos principais de propostas referentes a este tópico:

- replicação fixa e pré-determinada por popularidade [39];
- replicação randômica e parcial dos blocos dos vídeos [50];
- replicação dinâmica por um *threshold* de popularidade [51];

A replicação fixa e pré-determinada por popularidade é, por exemplo, utilizada em [39]. O usuário preenche um questionário em que indica que categoria de filmes está interessado e o proxy mais perto irá armazenar os filmes mais populares desta categoria. Entretanto, a popularidade de filmes e categorias é um conceito fluido, ou seja, um filme hoje popular pode não o ser mais em um curto espaço de tempo.

Para lidar com esse problema, é sugerida a replicação dinâmica por um *threshold* de popularidade. Em geral, este esquema funciona da seguinte maneira: os usuários se conectam e requisitam filmes através de um proxy local. Este anota a freqüência

dos pedidos dos clientes e, caso não possua o filme em seu cache, ele o requisita ao servidor principal. Quando o número de pedidos a este mesmo filme ultrapassa um dado *threshold*, o proxy armazena um cópia local do mesmo. Desta forma, as réplicas dos filmes são localizadas perto dos clientes que as desejam. Contudo, esta é um técnica reativa a demanda, onde assume-se que um filme que comece a se tornar popular de fato se tornará no futuro, o que nem sempre será necessariamente verdade. Caso isto não se torne realidade, todos os recursos do sistema gastos com a replicação do filme terão sido desperdiçados e a capacidade geral em se atender novos clientes, reduzida. Além disso, para este método, é necessário estimar a carga nos discos. Esta carga flutua e, na melhor das hipóteses, é possível estimar-se a carga média. Assim sendo, este método não consegue lidar com variações freqüentes na carga dos discos. Esta técnica pode ser encontrada em [51].

Finalmente, outra técnica de replicação utilizada na literatura é a replicação randômica e parcial dos blocos dos vídeos, como visto em [50]. Neste trabalho, os autores encaram a popularidade como um conceito não quantitativo e muito difícil de ser estimado. Por isso eles utilizam a replicação aleatória dos blocos dos vídeos. Entretanto, eles não replicam o vídeo inteiro, somente uma fração α dos seus blocos, onde temos $0 < \alpha < 1$. Assim, neste mesmo trabalho, é mostrado que esta forma de replicação é eficiente na obtenção de um maior número de pedidos dado um limite de atraso (*delay bound*).

2.4 Quanto ao caminho

A idéia desta seção é sumarizar o que foi proposto na literatura no tocante ao problema dos atrasos randômicos (*jitter*) e às variações das taxas de perda inerentes às redes de pacotes e, por conseqüinte, a Internet. Segundo a tabela 2.2, temos duas principais formas de escolher um caminho entre o servidor e o cliente: caminho único ou múltiplos caminhos. A primeira abordagem é utilizada pela maioria dos trabalhos já citados. Ela é a mais intuitiva de todas, uma vez que utiliza-se da infraestrutura já existente da Internet. Ela é útil em cenários em que temos variações de atraso pequenas, como em redes locais onde um proxy serve clientes locais, conforme descrito

na subseção 2.1.2, por exemplo. Entretanto, para as demais arquiteturas, podemos encontrar atrasos com variações significativas, além de taxas de perda elevadas, o que diminui a eficácia de uma política de caminho único de melhor esforço para a transmissão dos dados multimídia.

Entretanto, é possível elaborar soluções para os problemas supracitados sem a necessidade de se alterar a infraestrutura da Internet: em [52], é constatado que, caso existam múltiplos caminhos entre dois pontos, nunca existe um caminho que possua uma taxa de perda inferior todo o tempo dos demais. Ou seja, as taxas de perda são variáveis com o tempo. Tendo isto em vista, se soubermos como prever o caminho que possui a menor taxa de perda a cada instante, poderemos ter uma maior vazão para o cliente e, consequentemente, uma melhor qualidade de serviço para o usuário ou mais usuários dado uma qualidade de serviço.

Esta é a motivação para as propostas na literatura em se utilizar diversidade de caminhos para a transmissão dos dados, conforme visto em [53], por exemplo. No trabalho citado, o autor trabalha um cenário em que se tem servidores distribuídos capazes de enviar os mesmos blocos para um dado cliente. Entretanto, estes servidores possuem caminhos diferentes, ou seja, gargalos diferentes, para o mesmo cliente. Então, o cliente observa a taxa de perda de cada servidor e requisita a transmissão de quantidades de dados de diferentes servidores de acordo com esta taxa de perda calculada. Entretanto, como achar a taxa ótima e como predizer a taxa de perda nos caminhos disponíveis ainda são problemas a serem resolvidos por abordagens deste tipo, apesar de [53] e [52] começarem a tratá-los.

Em [54], temos a proposição de um algoritmo de predição da taxa de perda de curto prazo com a utilização de uma Cadeia de Markov Oculta (*Hidden Markov Model*), que poderia ser utilizado para se obter diversidade de caminhos na transmissão do vídeo para o cliente. Por exemplo, supondo que, na figura 2.2, os proxies são, na verdade, nós de armazenamento por onde irei distribuir aleatoriamente todo meu conteúdo, um cliente situado em qualquer local da rede receberá a transmissão do vídeo por mais de um caminho. Em [55] temos que, quando utilizamos mais de um caminho, diminuímos o tamanho médio das rajadas de perda. Logo, como o servidor RIO, utilizado por este trabalho, tem uma arquitetura como a descrita na

Tabela 2.2: Classificação das propostas de arquiteturas para streaming multimídia.

Quanto a função dos agentes	Quanto a forma de *streaming*	Quanto ao caminho	Quanto a alocação de blocos
VoD DVoD P2PVoD	Unicast Multicast Redes Ativas (*active routing*)	Caminho único (*best effort single path*) Diversidade de Caminhos (*patch switching*)	Indivisíveis *Striping* Randômica

Figura 2.2, podemos fazer a sua relação com o problema de diversidade de caminhos porque teremos caminhos diferentes sendo usados por um mesmo cliente para receber o *stream* de vídeo requisitado.

Capítulo 3

Arquitetura do RIO

O objetivo deste capítulo é apresentar a arquitetura do servidor multimídia DVoD utilizado neste trabalho: o *Randomized I/O Multimedia Storage Server* ou RIO [46]. A técnica chamada de *Randomized I/O* (RIO) foi concebida na *University of California at Los Angeles* (UCLA) por José Renato Santos e Richard Muntz, conforme descrito em [46]. Nas próximas seções, iremos discorrer sobre sua arquitetura de modo geral e detalhar alguns objetos mais importantes da mesma.

3.1 Origem e evolução do RIO

Uma primeira versão do RIO foi desenvolvida na UCLA como um servidor de mundos virtuais (*Virtual World Data Server*). Através de um projeto de cooperação internacional CNPq/NSF entre a UCLA, UFRJ e UFMG, uma nova versão do RIO foi desenvolvida. O RIO continuou a ser aprimorado visando aplicações de ensino a distância na UFRJ. Desta forma, um novo cliente foi projetado e praticamente todo o código do servidor foi reescrito através dos novos requisitos implementados, voltados para a aplicação alvo. Estes novos requisitos incluem, por exemplo, métodos de compartilhamento de banda. Este novo servidor encontra-se hoje em operação em sete pólos do consórcio CEDERJ do Estado do Rio de Janeiro (*http://www.cederj.rj.gov.br*). Um protótipo em constante evolução existe no laboratório LAND/COPPE/UFRJ.

A proposta inicial do servidor RIO consistia em criar um servidor multimídia

universal, onde não importaria que mídia ou que tipo de codificação seria utilizada nos objetos por ele armazenados. A aplicação cliente é que seria responsável por saber como decodificar e exibir estes objetos. Por isso que, no RIO, é possível definir dois tipos de tráfego para uma dada transmissão: com ou sem restrição de tempo. Conforme o esperado, o tráfego com restrição de tempo possui uma prioridade maior sobre o outro tipo de tráfego. Visualização de imagens ou qualquer outro conteúdo estático são exemplos de aplicações sem restrição de tempo, enquanto que a visualização de vídeos representa o tráfego do outro grupo.

Outra característica que se desejava inicialmente era construir um servidor propício para a visualização de ambientes 3D. Assim, o RIO possuía mecanismos de antecipação de demanda dos clientes, como buffers de *lookahead*, por exemplo. Por último, desejava-se um servidor que pudesse testar outra proposta de alocação de vídeos que não o tradicional *striping*. Assim, o RIO utiliza a alocação randômica, particionando os objetos em blocos de tamanho pré-definidos em sua inicialização, conforme teoria descrita na seção 2.3.2. Sua implementação será descrita na seção 3.5.1.

Após ter sido comprovada a viabilidade de sua arquitetura via simulações, como as de [45], sua implementação inicial foi para máquinas SUN E4000, sendo portado para um *cluster* de PCs com sistema operacional Linux, utilizando-se C++ como linguagem. Posteriormente, o grupo de pesquisas LAND|COPPE|UFRJ (*http://www.land.ufrj.br*) passou a manter a arquitetura, com uma total reconfecção do cliente, chamado de *RioMMClient* e a adição de novas funcionalidades, conforme o reportado em [30] e [56], por exemplo. Com isso, os objetivos do servidor foram redirecionados. Ele passou a ser usado principalmente para ensino a distância e seu primeiro emprego em um ambiente real com esta aplicação foi nos cursos administrados pelo CEDERJ. Para ambientes virtuais 3D, o RIO continua a ser utilizado na UCLA.

Entre algumas das novas funcionalidade do servidor desenvolvidas pelo LAND, encontram-se

- o *RioMMClient*, interface gráfica para visualização de vídeos e operações de VCR, com a utilização do *MPlayer* [57] para exibição dos vídeos;
- o *riosh*, interface textual e gráfica para administração dos objetos armazena-

dos;

- um módulo de sincronização de transparências com o vídeo sendo exibido;
- um módulo para compartilhamento de banda voltada para o ensino a distância (*Patching Interativo* [30]);
- um buffer de leitura no servidor para mitigar os efeitos de atrasos randômicos decorrentes da leitura nos discos e da transmissão pela rede [56];
- um novo algoritmo de controle de admissão, conforme descrito em [56];
- um módulo de coleta de medidas de desempenho, como o tempo médio de leitura de um bloco de dados nos discos [56];
- várias outras funcionalidades para melhorar o desempenho para o ambiente de ensino a distância;

A Figura 3.1 apresenta a arquitetura do RIO. Temos um único módulo servidor, que é o núcleo do sistema e aonde os clientes devem se conectar para acessar o conteúdo multimídia. Podemos ter um ou mais nós de armazenamento do conteúdo multimídia administrados pelo nó servidor. A comunicação de controle entre cada entidade é feita através de uma conexão TCP. Por exemplo, a comunicação de controle consiste na visualização do conteúdo do servidor, sua atualização ou a transmissão de pedidos de blocos de dados ao se iniciar a exibição de algum vídeo. A transferência do conteúdo multimídia é feita via UDP, diretamente entre os nós de armazenamento e os clientes.

3.2 *Framework* das entidades do RIO

Iremos descrever sucintamente algumas das classes e objetos do *framework* do RIO. Estas classes e objetos podem ter seus inter-relacionamentos esquematizados na Figura 3.2.

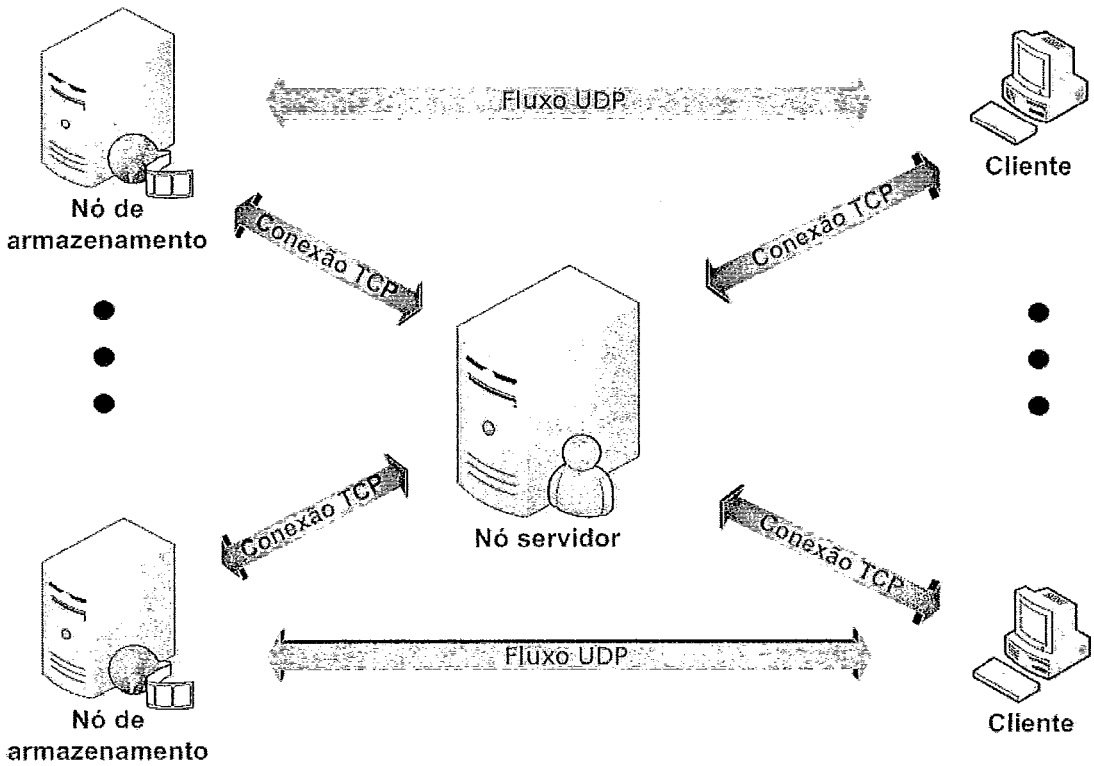

Figura 3.1: Arquitetura do RIO.

3.2.1 O Objeto Servidor

Em um objeto servidor, suas principais classes componentes são: *SessionManager*, *StreamManager*, *ObjectManager*, *Router*, *DiskManager* e *EventManager*. Iremos detalhar estas classes a seguir.

Gerenciador de Sessão (*SessionManager*)

O *SessionManager* é a porta de entrada do cliente no servidor. Ele é responsável em tratar e encaminhar para a classe responsável todos os pedidos do cliente. A cada nova sessão é iniciada uma nova *thread* para o tratamento dos pedidos daquele cliente. O *SessionManager*, então, repassa o pedido para o *StreamManager*. O *SessionManager* também verifica a consistência de todos os parâmetros de entrada do servidor, como, por exemplo, garantir que o número de réplicas de um objeto não seja maior do que o número de nós de armazenamento, uma vez que só podemos ter uma cópia por nó.

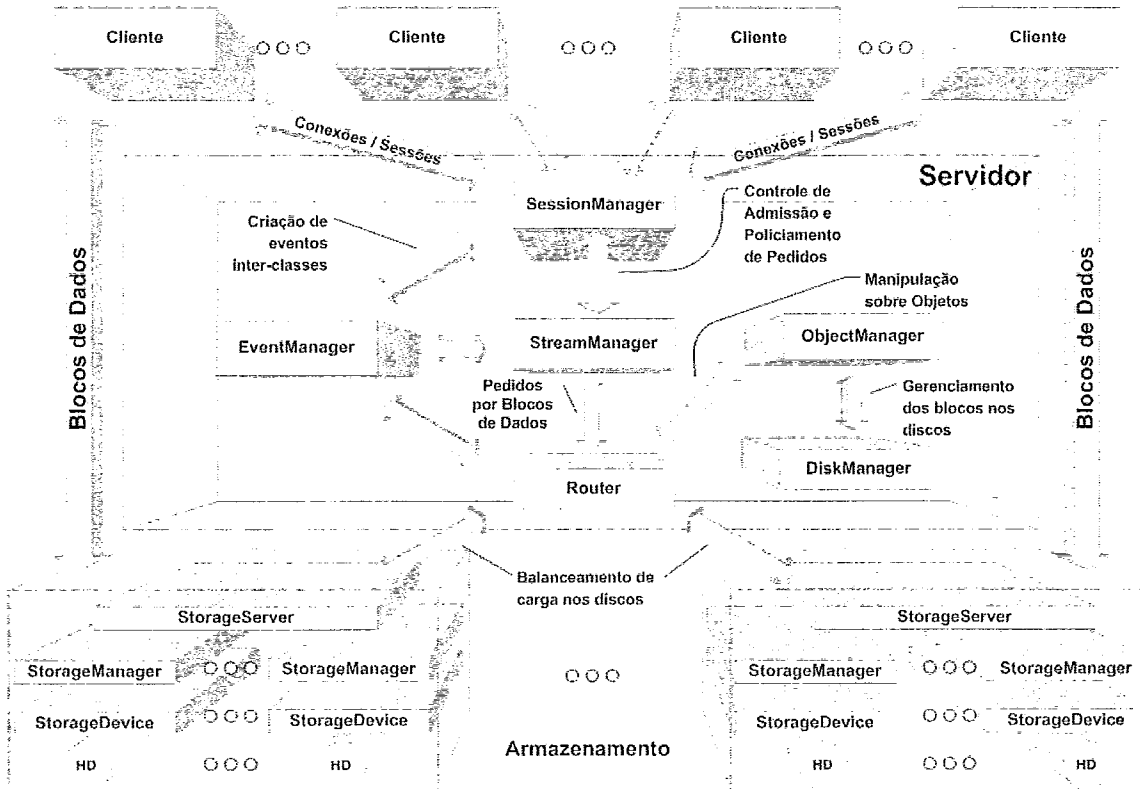

Figura 3.2: Diagrama de classes dos componentes do RIO.

Gerenciador de Eventos (*EventManager*)

Esta classe tem como responsabilidade a geração dos eventos que serão utilizados como comunicação entre as diversas entidades do objeto servidor. A comunicação é feita da seguinte forma, em geral: a cada evento gerado, ele é posto na fila de requisições não atendidas que as classes do objeto Servidor possuem, específica ao tipo de requisição realizada. A cada chamada de geração de evento, um novo objeto de evento é instanciado e inicializado com as informações passadas, de acordo com o tipo de evento a ser criado. Tipos de evento incluem a requisição do envio de um bloco do nó de armazenamento para um dado cliente, a criação ou deleção no disco de um dado objeto, dentre outros.

Gerenciador de Fluxos (*StreamManager*)

O *StreamManager* é o responsável pelo gerenciamento dos fluxos entre o cliente e o servidor. Ele é quem executa o controle de admissão, que será descrito na seção 3.3.

O *StreamManager* também realiza um policiamento de pedidos, descrito na seção 3.4. Caso o pedido do cliente passe por todos estes controles, o *StreamManager* cria um evento no *EventManager* com o pedido recém-chegado e o encaminha para o *Router*.

Gerenciador de Objetos (*ObjectManager*)

O *ObjectManager* é responsável pelos metadados dos objetos armazenados no servidor. O gerenciamento das operações sobre os objetos, tais como: criação, exclusão, abertura e fechamento é outra função do *ObjectManager*. Quando algum cliente se conecta ao servidor para a criação de um novo objeto, o *ObjectManager* deve alocar os blocos de todas suas cópias e atualizar os metadados dos objetos nos discos, através do *DiskManager*. O mesmo ocorre para a exclusão de objetos.

Os metadados dos objetos são escritos em um arquivo da seguinte forma: no diretório do servidor, é criado uma pasta chamada de *FileRoot*. Para cada usuário cadastrado no sistema, uma subpasta é criada, para funcionar como diretório raiz deste usuário. Então, de acordo com a estrutura de diretórios que o usuário crie, ela é reproduzida a partir desta raíz. A cada objeto criado, um arquivo homônimo é criado. Ele contém um cabeçalho com as seguintes informações:

- a assinatura de um objeto do RIO, que o servidor usa para constatar que o arquivo se trata mesmo de um arquivo de metadados do RIO;
- a data e a hora que o objeto foi criado;
- o tamanho do bloco utilizado, em bytes;
- a quantidade total de blocos do objeto;

Depois do cabeçalho, encontramos a localização dos blocos propriamente ditos. Em cada linha, temos em qual disco e qual posição no disco cada bloco se encontra. Cada disco possui um número único, dado pelo servidor ao iniciá-lo. Os blocos são dispostos no arquivo de forma ordenada, ou seja, na primeira linha temos o primeiro bloco, na segunda linha temos o segundo bloco e assim por diante. No caso de termos replicação de blocos, em cada linha imediatamente seguinte a linha de um

dado bloco encontramos as informações de suas réplicas. Assim, para achar a linha em que se encontra uma dada réplica de um

bloco, calculamos:

(número do bloco - 1) * número de replicações no sistema + número da réplica desejada.

Na Figura 3.3, temos um exemplo comentado de um arquivo de metadados do RIO.

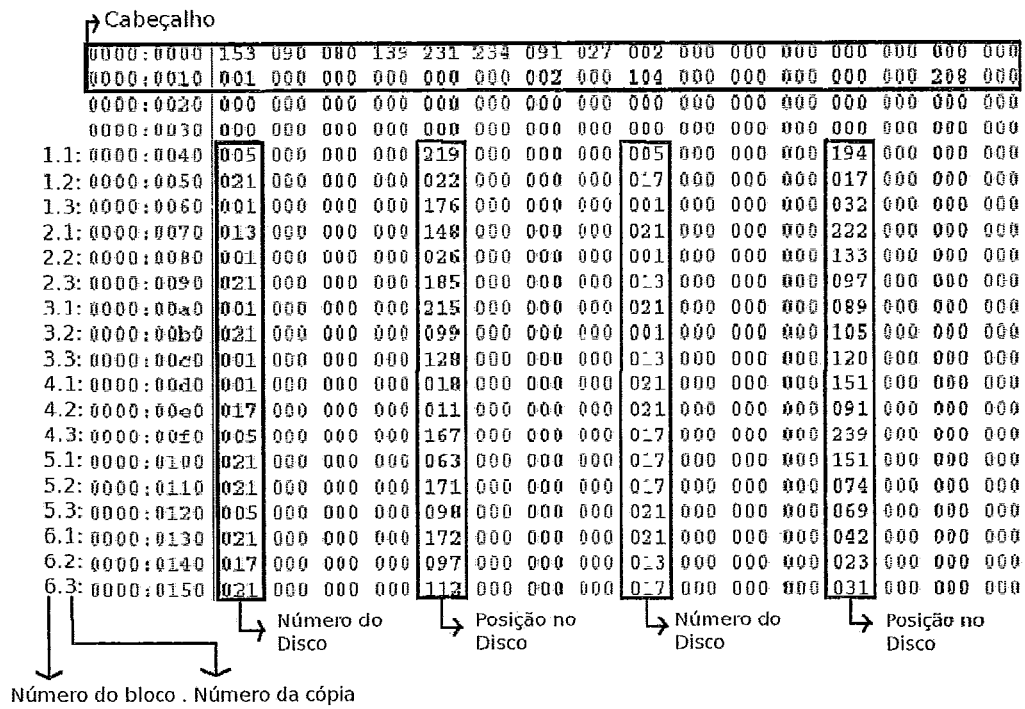

Figura 3.3: Arquivo de metadados de um objeto do RIO com três cópias.

Gerenciador de Disco (*DiskManager*)

A alocação física dos blocos nos discos é a responsabilidade do *DiskManager*. Quando alguma classe solicita ao *ObjectManager* a criação de um novo bloco, esta encaminha o pedido ao *DiskManager* para que este devolva uma posição livre em algum dos discos. Tanto um disco quanto uma posição livre são escolhidas randomicamente, de maneira uniforme. No caso de replicação, o *DiskManager* retorna uma posição livre para cada cópia, assegurando que nenhuma das cópias esteja alocada

no mesmo nó de armazenamento. Maiores detalhes da estratégia de replicação são encontrados na subseção 3.5.1.

Roteador (*Router*)

No *Router* encontramos, para cada disco do sistema, duas filas: uma fila para os pedidos com restrição de tempo e outra para os pedidos sem restrição de tempo, conforme é esquematizado na Figura 3.4. A fila com restrição de tempo tem prioridade sobre a fila sem restrição de tempo. Cada fila tem o FIFO como política de atendimento. Os pedidos de leitura e escrita recebidos pelo *Router* são encaminhados para o dispositivo que pode atender este pedido, conforme determinado pelo *ObjectManager*. No caso de termos mais de um dispositivo que possa atender um dado pedido, o *Router* executa um algoritmo de balanceamento de carga, a saber: para cada réplica, o *Router* examina a fila do disco correspondente e escolhe a réplica com menor fila. Caso haja empate, é escolhido o bloco na ordem em que a replicação foi feita. É válido ressaltar que, ao auferir o tamanho da fila, o *Router* leva em conta qual a natureza do pedido. Ou seja, para pedidos com restrição de tempo, somente a fila com restrição de tempo é considerada. Para pedidos sem restrição de tempo, ambas filas, com e sem restrição de tempo, são consideradas.

3.2.2 Objeto de armazenamento

Em cada nó de armazenamento, encontramos as seguintes classes: *RouterInterface*, *StorageDevice*, *StorageManager* e *ClientInterface*. Este diagrama de classes é ilustrado na Figura 3.5 [30].

Interface com o Roteador (*RouterInterface*)

A responsabilidade do RouterInterface consiste no recebimento e pelo envio de informações de controle entre o nó de armazenamento e o nó servidor, via uma conexão TCP. Exemplos de informações de controle são: pedidos de envio de bloco de dados para um cliente, confirmação de envio de um pedido, etc. Por exemplo, no caso do envio de um pedido, ele é imediatamente encaminhado para o *StorageManager* e no caso do envio de uma confirmação, ela é enviada ao *ClientInterface*.

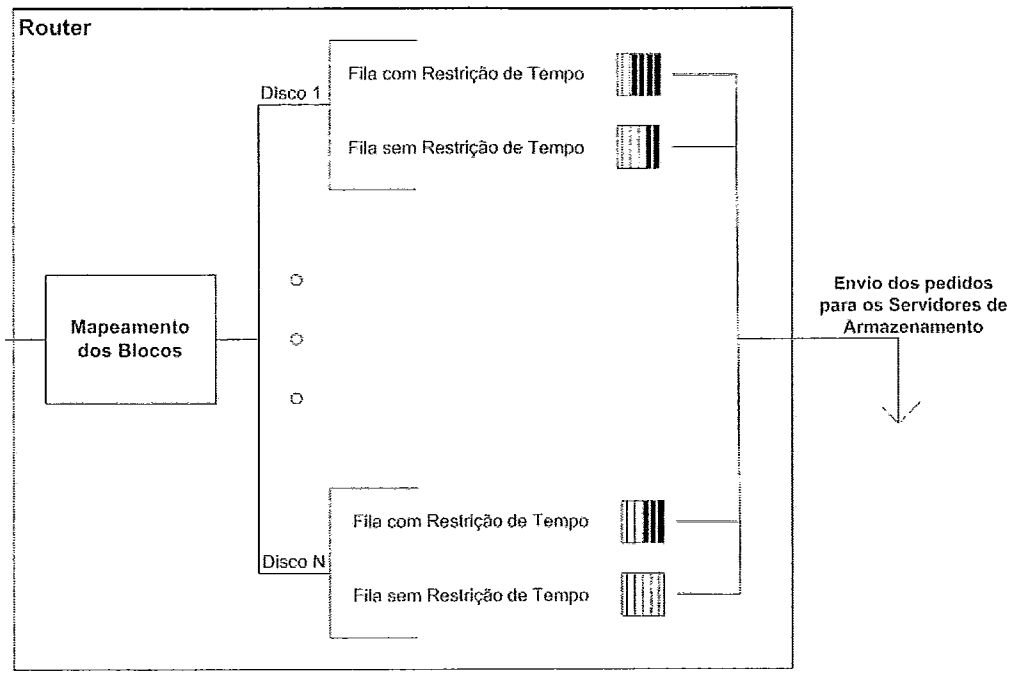

Figura 3.4: Fluxo lógico do *Router*.

Dispositivo de Armazenamento (*StorageDevice*)

Para a realização de operações de entrada e saída de cada disco do sistema, temos uma instância do *StorageDevice*. Os discos que ele é capaz de controlar podem ser ou um arquivo ou uma partição do disco rígido do computador.

Gerenciador de Armazenamento (*StorageManager*)

Para cada *StorageDevice* existe um *StorageManager* para gerenciá-lo. O *StorageManager* é responsável pelo escalonamento dos pedidos feitos ao disco que gerencia. Assim, é construída uma fila com os pedidos a serem atendidos. Para cada pedido de leitura, é associado um buffer para o armazenamento do bloco solicitado e ele é enviado ao cliente pelo *ClientInterface*.

Interface com o Cliente (*ClientInterface*)

Temos a classe *ClientInterface* como responsável pelo envio e recebimento dos blocos de dados. O nó de armazenamento transmite, durante uma sessão de um cliente, e recebe, durante uma cópia de um vídeo, blocos de vídeos através desta

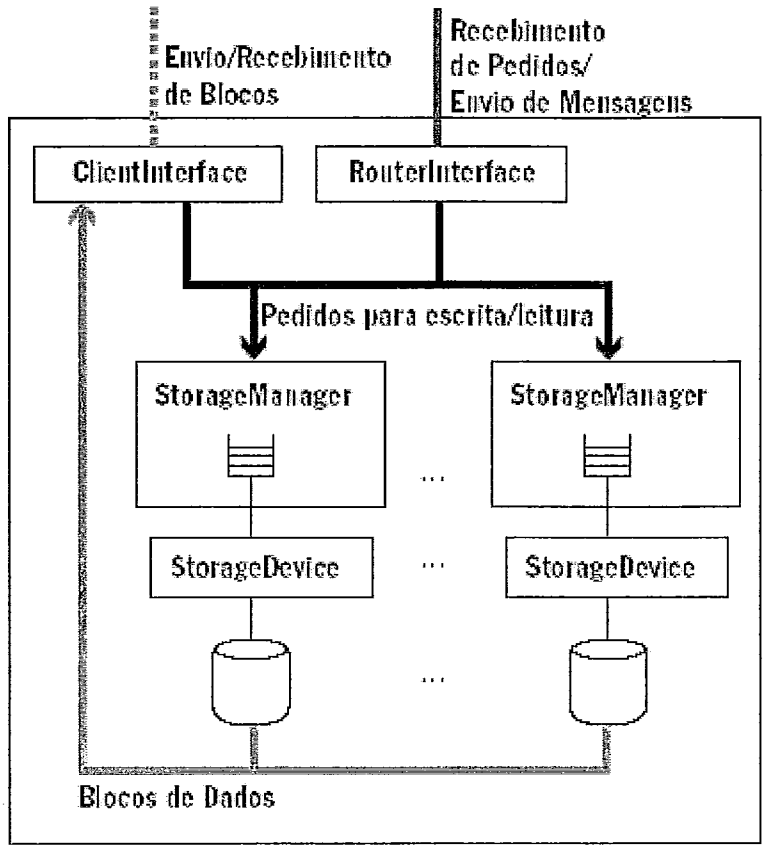

Figura 3.5: Diagrama de Classes do nó de armazenamento.

classe. Estas operações são feitas via UDP sendo que, ao receber um vídeo, temos um controle da transmissão pelo cliente, com retransmissões no caso de perda. Cada bloco a ser enviado para o cliente é fragmentado, de acordo com algumas variáveis pré-configuradas, a saber: quantidade de fragmentos que compõe um bloco e o endereço do receptor, formado pelo IP, porta e identificação da requisição. A troca de mensagens entre o nó de armazenamento e o nó servidor é feita via TCP, através desta classe.

3.2.3 O Objeto Cliente

Utilizamos dois clientes para o servidor RIO: o *riosh* e o *RioMMClient*. O primeiro tem como fim a administração dos objetos armazenados dentro do servidor. Possui funções de cópia, exclusão, listagem do conteúdo dos diretórios entre outras.

Já o segundo é o cliente para visualização de vídeos do servidor RIO. Possui funcionalidades como avançar, retroceder, pausar, além de sincronização com transparências. Para exibir a mídia recebida, é utilizado o MPlayer [57].

Simplificadamente, o cliente funciona da seguinte maneira: é definido um tamanho de *buffer* em número de blocos em sua inicialização (*playout buffer*). Ao se iniciar a exibição de um vídeo, o cliente envia para o servidor os pedidos dos blocos a fim de encher o *playout buffer* e inicia um temporizador. O primeiro evento que ocorrer entre o *buffer* encher ou o temporizador expirar dispara a exibição do vídeo. Depois disso, ele passa a fazer uma requisição de um novo bloco a cada vez que um bloco é consumido, até o fim do vídeo. Caso o bloco não chegue a tempo ou não chegue por completo, o cliente recupera os fragmentos que chegaram e os repassa para o MPlayer. Em todos os experimentos que realizamos foi utilizado a reprodução seqüencial do vídeo.

Mais detalhes sobre as classes que compõem os clientes *RioMMClient* e *riosh* podem ser encontrados em [30].

3.3 Controle de admissão

O RIO implementa, no *StreamManager*, um controle de admissão de usuários, no momento em que é solicitado pelo cliente a abertura de um novo fluxo de dados. Posteriormente, em [56] foi implementado um outro controle de admissão, mais apurado, porém com um custo computacional mais elevado. Descreveremos abaixo o controle original. Ele é baseado nas taxas solicitadas por cada cliente. Um arquivo de configuração é lido durante a inicialização do servidor, no qual é informada a taxa total aceita pelo servidor e a taxa reservada para fluxos que não possuem restrições de tempo. Esta reserva de banda garante o atendimento destas aplicações pelos discos, mesmo com a existência de fluxos com restrição de tempo. Outra variável do sistema é a taxa alocada, que é inicializada com 0 e, de acordo com o que cada cliente requisita em cada fluxo, essa variável é incrementada de acordo. Assim, para a admissão de um cliente, este envia ao servidor se deseja um tráfego com restrição de tempo ou não, se é uma operação de escrita ou leitura e a qual a taxa solicitada.

O controle considera somente os fluxos com restrição de tempo e um novo usuário é aceito se a seguinte condição for verdadeira:

(Taxa Alocada + Taxa Solicitada) <= (Taxa Total - Taxa Reservada Sem Restrição de Tempo)

Caso esta condição seja verdadeira, o fluxo novo será aceito. Por fim, a Taxa Alocada é acrescida da Taxa Solicitada. Obviamente este esquema não leva em consideração a variabilidade do tráfego. O esquema de [56] é mais eficiente pois não assume taxa constante. Por outro lado, conforme explicaremos no Capítulo 4 estamos interessados no limite máximo do servidor e, portanto, o controle de admissão é irrelevante para nossos estudos.

3.4 Policiamento de pedidos

Devido a rajada de pedidos pelos clientes, possíveis desbalanceamentos de carga e conseqüente atraso dos demais pedidos podem ocorrer. Com o objetivo de evitar este cenário, o servidor implementa um algoritmo de policiamento de pedidos. Este policiamento é feito pelo *StreamManager*, através da implementação de uma fila [56] tipo *leaky bucket*. Esta fila possui dois parâmetros: a capacidade de fichas F que cada cliente pode armazenar e a taxa de geração de fichas r. Assim, o custo de enviar um pedido para o Router é uma ficha. Desta forma, pedidos só serão armazenados no caso em que existem fichas disponíveis. A quantidade de fichas vai sendo incrementada até o valor F e de acordo com a taxa de geração r. Este valor F é preenchido na inicialização do servidor. Já a taxa de geração de fichas é obtida a partir da taxa solicitada pelo cliente. Assim, podemos concluir que a quantidade máxima de pedidos que serão enviados em qualquer intervalo de tempo t é rt+F. Este mecanismo foi implementado em [56] e é utilizado hoje. A vantagem do *leaky bucket* é a suavização de pedidos transmitidos e consequente suavização da taxa de transmissão.

3.5 Contribuições deste trabalho ao Servidor RIO

Por fim, esta seção destina-se a sumarizar as contribuições feitas ao servidor RIO pela realização deste trabalho. As principais contribuições são:

1. Remodelagem da estratégia de replicação;
2. Correções no algoritmo de comunicação com a interface de rede;
3. Otimização do tamanho de pilha de algumas *threads* do objeto de armazenamento;
4. Modificação do cliente para a impressão de informações de envio e recebimento;
5. Criação de scripts para análise de resultados e cálculo de métricas;
6. Alterações gerais no código, como inicialização de variáveis, melhora de mensagens de erro, reestruturação de alguns algoritmos, embelezamento do código, etc;

Para a compreensão deste trabalho, é necessário aqui elucidar os itens 1 e 4. Os demais itens serão detalhados no Apêndice A.

3.5.1 Estratégia de replicação

A estratégia de replicação implementada no RIO funciona da seguinte forma: o número de réplicas desejado é informado através de um arquivo de configuração externo, armazenado pelo *SystemManager*. O seu limite é o número de nós de armazenamento existentes: mais a frente veremos que o servidor garante que só teremos uma cópia por nó de armazenamento. Então, quando um cliente inicia uma sessão para cópia de dados, o servidor deve informar ao cliente qual posição e em qual nó de armazenamento cada bloco do arquivo enviado será gravado. Note que um mesmo bloco pode ser enviado diversas vezes, caso tenha sido configurado que os arquivos no servidor terão replicação. O servidor também deve informar ao nó de armazenamento que o cliente enviará um bloco de dados para uma dada posição em seu disco.

Para o servidor fornecer estas informações, é utilizado o *StreamManager*, o qual requisita ao *DiskManager* que informe posições disponíveis para o armazenamento de um dado bloco. Para tal, é passado para o *DiskManager* qual o bloco em questão e quantas réplicas são desejadas. O *DiskManager*, por sua vez, verifica se há espaço disponível para um novo bloco. Caso exista, ele sorteia um nó de armazenamento, com probabilidade uniforme. Caso este nó já possua uma cópia do bloco a ser gravado, o *DiskManager* sorteia um novo nó, até escolher um nó sem nenhuma cópia daquele bloco. Uma vez escolhido o nó, ele passa a sortear com probabilidade uniforme uma posição em seu disco. Ele repete este sorteio até achar uma posição livre. Quando obtém esta informação, ele a retorna ao *StreamManager*, que a repassa ao cliente e ao devido nó de armazenamento. O fluxo dos dados pode ser melhor visualizado na Figura 3.6.

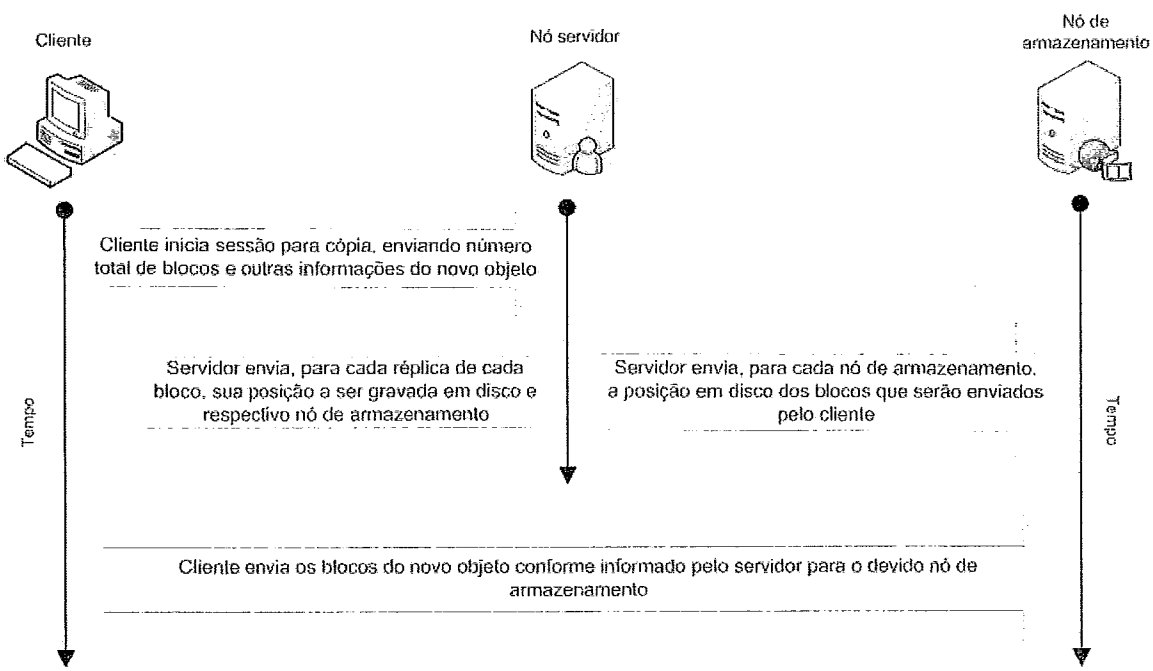

Figura 3.6: O fluxo da troca de informações entre os agentes do sistema durante o processo de cópia de um novo objeto.

3.5.2 Impressão de informações pelo cliente

A impressão de dados pelo cliente, para ser usada nos experimentos que serão descritos no Capítulo 4, consiste nos seguintes dados:

- Em qual computador está sendo executado;
- O exato momento em milissegundos que enviou um pedido de um bloco;
- O instante exato em milissegundos que recebeu um bloco inteiro;
- O instante em milissegundos em que iniciou a execução de um bloco;
- Por qual bloco está esperando quando o seu buffer esvazia;
- Se está na hora de reproduzir um dado bloco e o mesmo ainda não chegou por completo, é impresso qual o número do bloco que isso ocorreu e quantos bytes do mesmo foram reproduzidos;
- Quando um fragmento de um bloco chega depois do momento de sua reprodução, é impresso qual o número do fragmento e do bloco que chegou atrasado;

No capítulo 4 serão explicadas as métricas calculadas em cima destes dados coletados, que são impressas ao final do log.

Capítulo 4

Arquitetura da Rede Giga/RNP, Experimentos Realizados e Resultados

DETALHES da arquitetura da rede Giga e dos experimentos realizados serão o tema deste capítulo. Cada experimento sempre será motivado com o objetivo e os resultados esperados de cada teste realizado. Após o detalhamento de cada experimento, serão mostrados os resultados obtidos, acompanhados de uma breve justificativa para os mesmos.

4.1 Objetivo dos Experimentos

O objetivo dos experimentos é o de estudar o desempenho do servidor RIO em um ambiente distribuído e heterogêneo. Por exemplo, verificar a qualidade do vídeo com um número crescente de usuários. Este corresponde ao primeiro cenário do projeto Giga/RNP [58]. Deve-se ressaltar que, neste cenário, um dos nós de armazenamento não fará parte desta rede Giga. O impacto desta configuração será avaliado. Ao se realizar este estudo, será procurado determinar o gargalo de desempenho do cenário de cada experimento realizado. Além disso, serão motivados os experimentos subseqüentes com possíveis soluções para os possíveis gargalos existentes. Por fim será mostrado quais destas soluções são realmente efetivas e quais não o são.

4.2 Arquitetura da Rede Giga

A implantação da rede Giga foi uma iniciativa da Rede Nacional de Ensino e Pesquisa (RNP - *http://www.rnp.br*), em conjunto com diversos centros de pesquisa e universidades. Para a realização dos experimentos deste trabalho, as instituições envolvidas foram: a Universidade Federal do Rio de Janeiro (UFRJ), a Universidade Federal Fluminense (UFF) e a Fiocruz, onde o *switch* giga foi alocado dentro do Canal Saúde. Além destas, a Universidade Federal de Minas Gerais (UFMG) também participou dos testes, mas sem que a rede Giga chegasse até seu campus. Ou seja, o acesso à UFMG foi através da conexão à Internet já existente nesta instituição.

A arquitetura da rede Giga propriamente dita pode ser visualizada na Figura 4.1. Ela consiste em uma rede virtual privada (VPN) (*Virtual Private Network*) que interconecta os *switches* giga da UFRJ, UFF e Fiocruz. Ou seja, nesta VPN, temos todos os *switches* a um salto de distância.

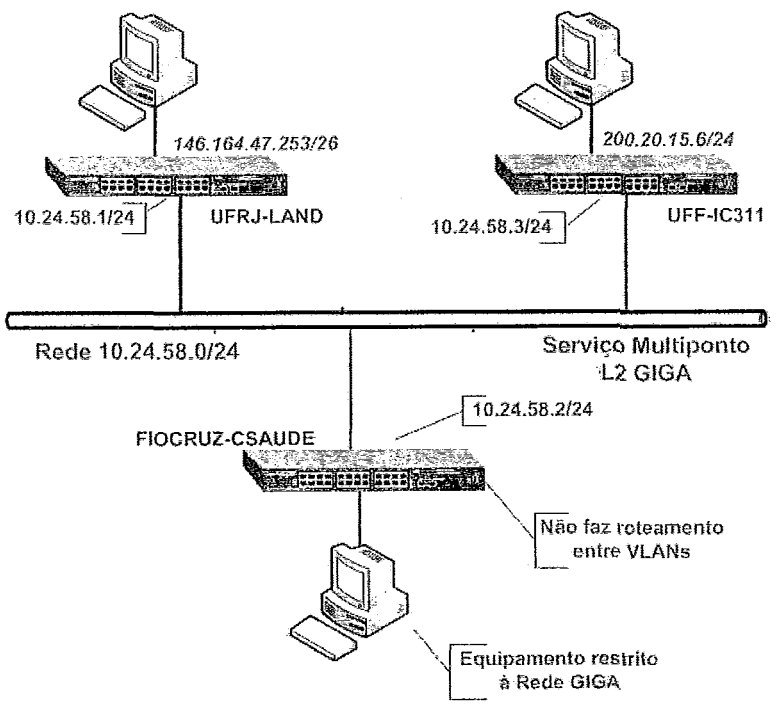

Figura 4.1: Representação simplificada da VPN da rede Giga.

Em cada instituição foi montada uma estrutura de acordo com a disponibilidade de equipamentos e utilização de cada máquina. Na Fiocruz temos a estrutura mais

simples, com um computador ligado no *switch* giga da instituição, conforme mostra a Figura 4.1. Na UFF e na UFRJ, temos os computadores do projeto conectados no *switch* giga. Este, por sua vez, está ligado no *switch* principal do laboratório da instituição. Assim, as máquinas também podem acessar a Internet normalmente. Temos apenas um computador disponível na UFF. Na UFRJ possuímos dez computadores para testes. A arquitetura da UFRJ pode ser vista na Figura 4.2 e a da UFF na Figura 4.1. Já na UFMG possuímos apenas uma máquina para testes e esta encontra-se ligada diretamente ao *switch* principal do laboratório em que está alocada. Assim, esta máquina possui apenas acesso via a conexão Internet da instituição. A arquitetura completa da rede pode ser vista na Figura 4.3.

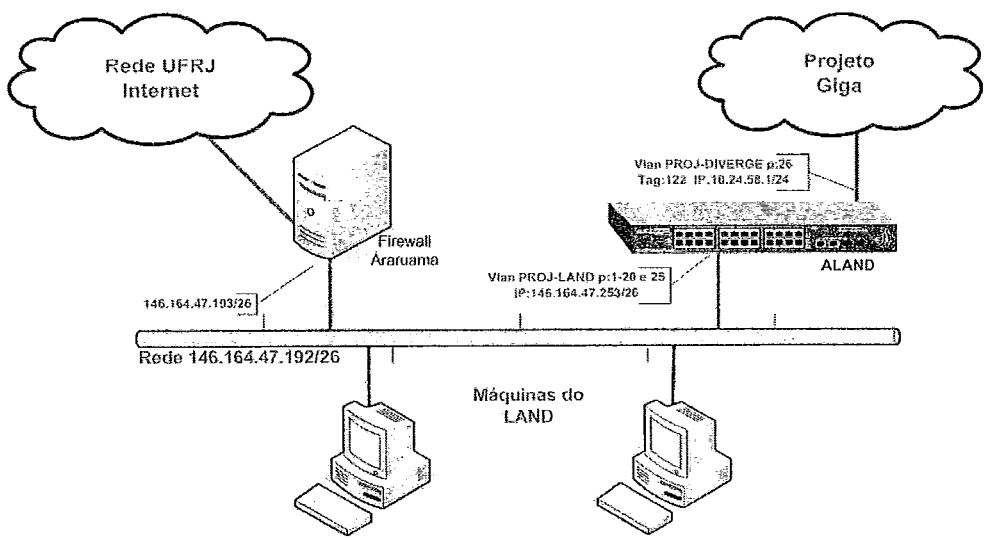

Figura 4.2: Representação simplificada da arquitetura da rede no laboratório da UFRJ.

A partir dos equipamentos disponíveis para os experimentos descritos acima, determinamos como seriam distribuídos os papeis para cada computador na rede. Em cada instituição teremos, no máximo, 1 nó de armazenamento, exceto a UFRJ que terá, no máximo, 2 nós de armazenamento. O servidor sempre ficará na UFRJ, bem como todos os clientes. Na Figura 4.3 temos ilustrada a configuração da rede com 5 nós de armazenamento.

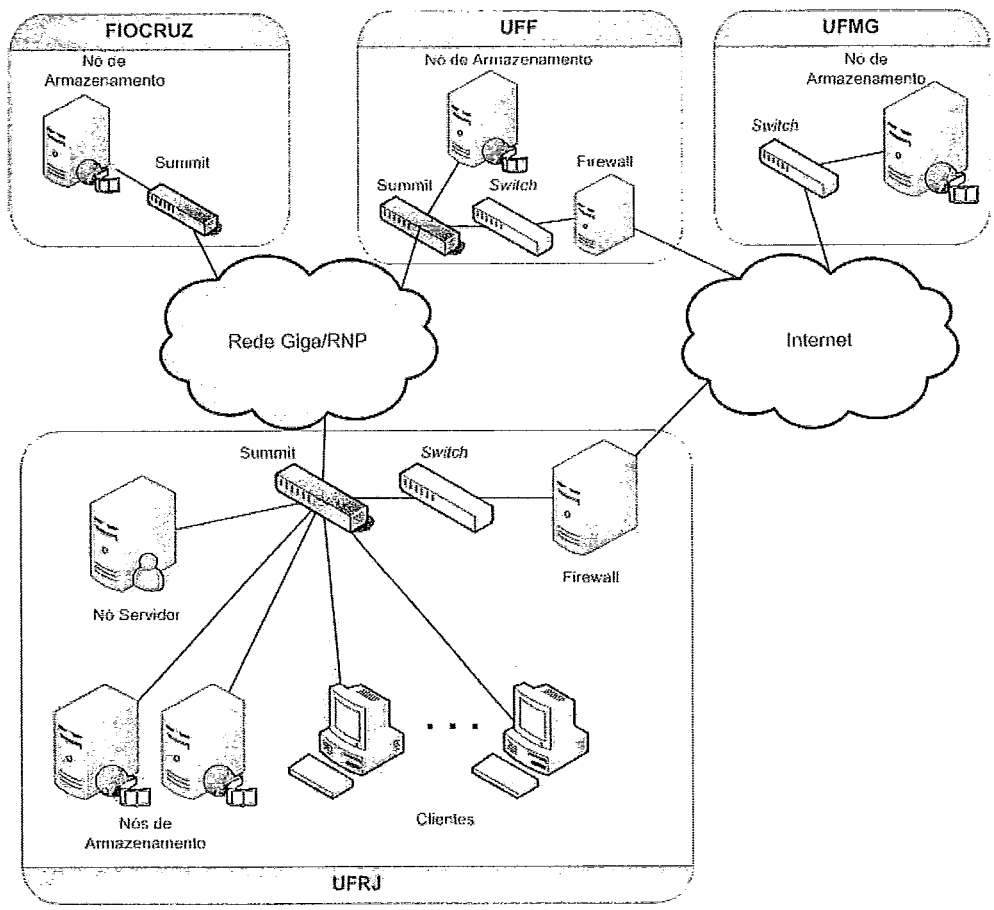

Figura 4.3: Representação simplificada da arquitetura de todo o ambiente de testes entre as instituições participantes dos experimentos.

4.2.1 Configuração dos Equipamentos Utilizados

É de suma importância ressaltar alguns detalhes gerais de todos os equipamentos utilizados para os experimentos, uma vez que estas características serão os determinantes do gargalo de desempenho de nosso cenário.

Os *switches* utilizados na rede Giga testada são todos da *Extreme Networks* (*http://www.extremenetworks.com*), modelo Summit 200-24. Este equipamento será chamado de summit. Eles possuem 24 portas 10/100 Mbits por segundo, doravante portas 100. Também possuem 2 portas 1000 Mbits por segundo, outrossim portas 1000. Todos os computadores utilizados têm uma configuração mínima melhor ou equivalente a um Pentium IV 3 GHz, 1 GB de RAM, HD SATA 133, rodando Mandrake Linux 10.2.

Todos os clientes e servidores foram conectados nas portas 100 do summit. O summit foi cascateado no *switch* do laboratório da UFRJ em sua porta 1000. Sua outra porta 1000 é utilizada para a conexão com a rede Giga. Todas as outras instituições seguiram o mesmo padrão. Somente a Fiocruz, por não utilizar o summit em sua rede interna, conectou o único computador lá existente em uma porta 1000 e a outra foi utilizada para a conexão com a rede Giga. A saída para a UFMG também é feita através de uma máquina conectada a um *switch* da COPPE em uma porta 100. Todos os gargalos no sistema são de 100 Mbits.

Por fim, foram realizados experimentos de 1 a 5 nós de armazenamento. Assim sendo, foram utilizados os nós das instituições participantes, incrementalmente, na seguinte ordem: dois na UFRJ, um na Fiocruz e um na UFF. Um nó de armazenamento na UFMG foi utilizado em paralelo com cada configuração, conforme será explicado em cada experimento.

4.3 Metodologia dos Experimentos

Os testes do ambiente neste primeiro cenário foram realizados com a configuração do servidor RIO de forma distribuída entre as instituições que participam do projeto. Com o servidor RIO distribuído conforme o descrito na seção 4.2, foram realizados testes de estresse do servidor através da simulação de clientes que solicitam blocos

de vídeos que estão guardados nos servidores de armazenamento distribuídos. Em todos os testes, todos os clientes foram configurados para utilizar um *buffer* de 5 blocos. Com o vídeo escolhido, este *buffer* armazena, em média, 4 segundos de vídeo. Ao analisar o caminho para a UFMG na seção 4.7, a escolha deste valor será explicada com maiores detalhes.

Para todos os testes de estresse, três métricas foram definidas:

- a porcentagem do vídeo requisitado que efetivamente chegou a tempo de ser exibido para um cliente, a qual chamamos de *Goodput*;
- a porcentagem de fragmentos atrasados, que são os fragmentos que chegam após ter passado o momento de serem exibidos, mas ainda durante a sessão do cliente;
- a porcentagem de fragmentos perdidos, que são os fragmentos que não chegam até o final da sessão do cliente;

Os passos seguidos para a realização dos testes foram os seguintes:

1. Configuração física das máquinas envolvidas no experimento;
2. Configuração das interfaces de rede destas mesmas máquinas;
   (a) Desligar autonegociação;
   (b) Forçar modo full-duplex e velocidade de 100 Mbps;
3. Configuração do RIO na rede Giga com N nós de Armazenamento;
   (a) Conectar, via ssh, a cada máquina;
   (b) Iniciar e parametrizar os nós de armazenamento;
   (c) Configurar nó servidor com localização dos nós de armazenamento, número de réplicas a serem usadas, etc;
   (d) Iniciar servidor e copiar vídeo usado nos experimentos;
4. Início da simulação de X clientes para um número N de servidores de armazenamento;

(a) Conectar a cada máquina, via ssh, para iniciar os clientes;

(b) Indicar qual o comportamento de cada cliente;

(c) Configurar a duração do vídeo e da sessão;

5. Coleta e análise dos logs de cada cliente ao fim da exibição do vídeo;

   (a) Reunir todos os logs em um ponto único para análise;

   (b) Analisar todo o conteúdo para cálculo das métricas;

6. Reconfiguração do RIO com a adição do nó extra na UFMG;

7. Reinício da simulação com X clientes;

8. Coleta e análise dos logs de cada cliente ao fim da exibição do vídeo;

9. Voltar ao passo 3 por V vezes;

10. Aumentar o número de usuários e, se não chegar no número máximo do experimento, voltar ao passo 4;

A configuração física do passo 1 envolve, basicamente, a conexão dos computadores no summit. Já no passo 2, é preciso configurar a interface de rede das máquinas, desligando a autonegociação e forçando a velocidade de 100 MBits por segundo e *full duplex*. Isto só é feito por questão de compatibilidade com o summit. A parametrização do servidor RIO, no passo 3 significa a definição de diversas variáveis de ambiente, como, por exemplo, se será utilizada replicação dos blocos, quantos e aonde estão localizados os nós de armazenamento, etc. Neste passo somente são configurados nós de armazenamento na rede Giga. Neste passo também é copiado o vídeo utilizado em todos os testes: um fragmento de 64 segundos de uma das aulas do CEDERJ, onde o vídeo é codificado em MPEG-2 com uma taxa de 1,25 MBits por segundo, aproximadamente. O tamanho do trecho será explicado na próxima subseção.

No passo 4 temos o início da simulação, que é feita da forma descrita a seguir: um mesmo número n de clientes são iniciados em c computadores, de forma que tenhamos $n * c = N$ clientes na simulação. Caso a divisão não seja exata, o resto da

divisão é distribuído sequencialmente nas máquinas. O intervalo da quantidade N de clientes é definido em cada experimento, em relação a quantos nós de armazenamento teremos. Cada experimento explicará melhor a razão da escolha da faixa de N.

Após o cálculo de n, é iniciado um *script* em cada máquina que executa cada um destes clientes. Há um intervalo de 10 ms entre o início deste *script* em cada computador. Para iniciar os clientes, é calculado um atraso randômico a fim de evitar a sincronização de pedidos de blocos. Este atraso é um número uniformemente gerado em um intervalo de 0 a 10 ms.

O cliente passa a fazer os pedidos de blocos ao servidor. O cliente faz um log de toda a atividade, conforme descrito na seção 3.5.2. Ao terminar a simulação, todos estes logs são copiados para o computador que iniciou os *scripts* nos outros computadores. Só então é iniciado um terceiro *script* que faz todos os cálculos das métricas definidas nesta seção.

Após o cálculo das métricas o resultado é armazenado em um arquivo. Este arquivo é depois utilizado para o cálculo de todos os gráficos exibidos neste trabalho. Para calcular estes gráficos, calculamos a média dos pontos a serem traçados e seu intervalo de confiança de 95%. O cálculo do intervalo de confiança foi realizado conforme explicado em [59]. Por fim, o experimento é repetido por V vezes conforme o passo 3. Usamos V = 5, para o cálculo do intervalo de confiança.

4.3.1 Duração do experimento

A duração do experimento pode ser contabilizada da seguinte forma:

1. 1 minuto para estabelecer as conexões e iniciar os nós de armazenamento remotos, durante o passo 3;

2. 1 minuto de vídeo + 1 minuto de espera + 1 minuto aguardando a finalização do último cliente: 3 minutos para o passo 4;

3. 7 minutos, no pior caso, devido a quantidade de informações processadas durante o passo 5;

Assim, o total de tempo gasto durante um laço de experimento são de 11 minutos. Portanto, a geração de cada ponto do gráfico, que indica uma quantidade de clientes

simulados naquela passada, pode durar até 55 minutos, pois são 5 repetições de 11 minutos; Em geral, os gráficos possuem de 8 a 10 pontos, o que resulta de 7 a 9,2 horas de experimentação. Como não desejamos que o trafego de uso normal do laboratório interferisse com os experimentos, o vídeo de 64 segundos foi a maior duração que poderíamos usar em todos experimentos.

4.4 Primeiro Experimento: RIO e a rede Giga

4.4.1 Descrição dos objetivos

Inicialmente testamos o desempenho do servidor RIO em uma rede Giga cuja arquitetura foi descrita na seção 4.2. Para tal, este experimento será feito conforme o descrito na seção 4.3. O objetivo é auferir se o desempenho do servidor RIO é afetado pelo número de nós de armazenamento utilizados. Serão utilizados de 1 a 4 nós de armazenamento, incrementalmente. Foi escolhido 4 como número máximo de nós de armazenamento por limitações de disponibilidade de equipamentos.

4.4.2 Descrição dos resultados esperados

O primeiro experimento consistirá em comparar o desempenho do servidor RIO inicialmente sem qualquer replicação dos seus vídeos armazenados. Sistemas multimídia costumam ter como gargalo esperado ou o acesso aos discos ou a interface de rede. Entretanto, neste ambiente, o gargalo do sistema deverá ser a interface de rede, cuja capacidade máxima teórica é de 100 Mbits/s, muito menor do que a capacidade máxima teórica dos discos Serial ATA 133 utilizados, que é de 133 MBytes/s.

Portanto, cada nó de armazenamento teoricamente conseguiria servir no máximo 100 Mbps / 1,25 Mbps = 80 clientes. Ganhos de desempenho são esperados, em termos de menor porcentagem de fragmentos perdidos e atrasados, a cada novo nó de armazenamento adicionado.

4.4.3 Resultados e Análise

O gráfico da Figura 4.4 é composto pelo número de clientes simulados no eixo X e a porcentagem de fragmentos recebidos a tempo de serem exibidos, doravante *Goodput*, no eixo Y. Cada curva representa uma configuração distinta do sistema: o número de nós de armazenamento foi variado incrementalmente de 1 até 4, localizados conforme descrito na subseção 4.2.1. Também é possível ver os gráficos complementares: o gráfico da Figura 4.6 representa a porcentagem de fragmentos perdidos pelo número de clientes, no mesmo cenário. O gráfico da Figura 4.5 mostra a porcentagem de fragmentos atrasados pelo número de clientes simulados.

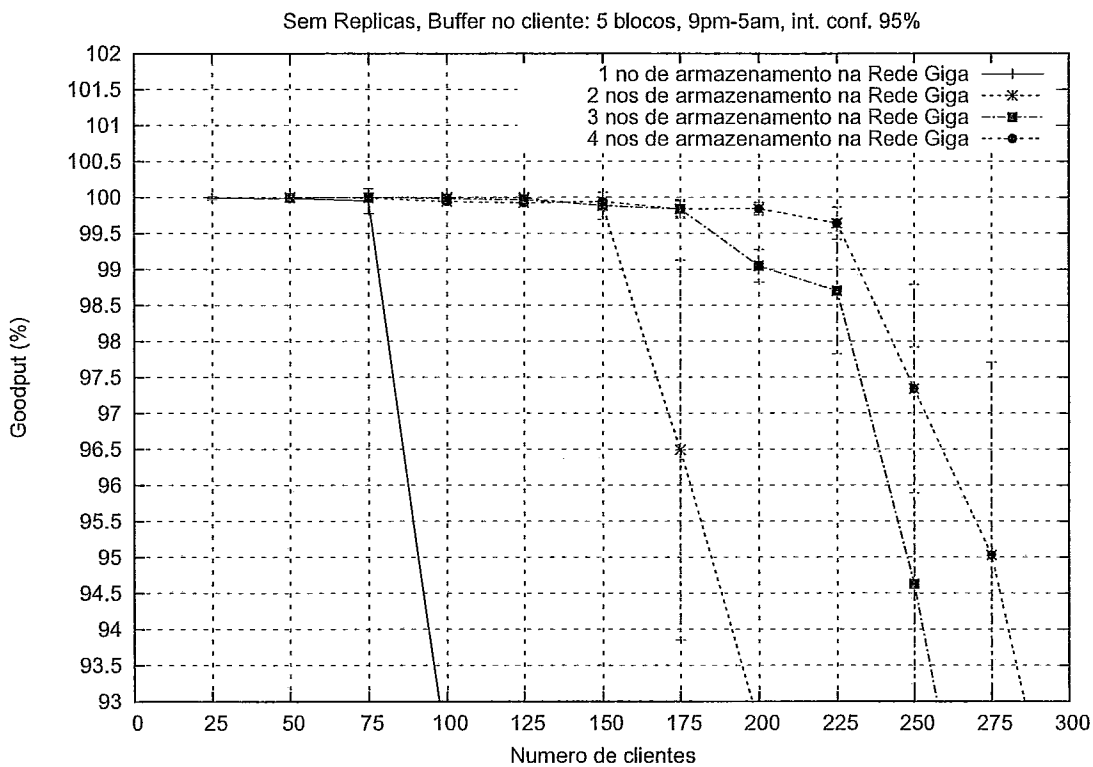

Figura 4.4: Gráfico do número de clientes x *Goodput* com o RIO somente na rede Giga, de 1 a 4 nós de armazenamento, sem réplicas.

Conforme o esperado, a cada nó de armazenamento adicionado é possível observar um ganho de desempenho. Não foram obtidos os 80 clientes por nó de armazenamento sem perdas porque existem diversos fatores que influenciam no atraso para o envio dos fragmentos dos blocos dos vídeos: o tempo de transmissão das mensagens entre o nó servidor e o nó de armazenamento, a flutuação da taxa do vídeo, o desba-

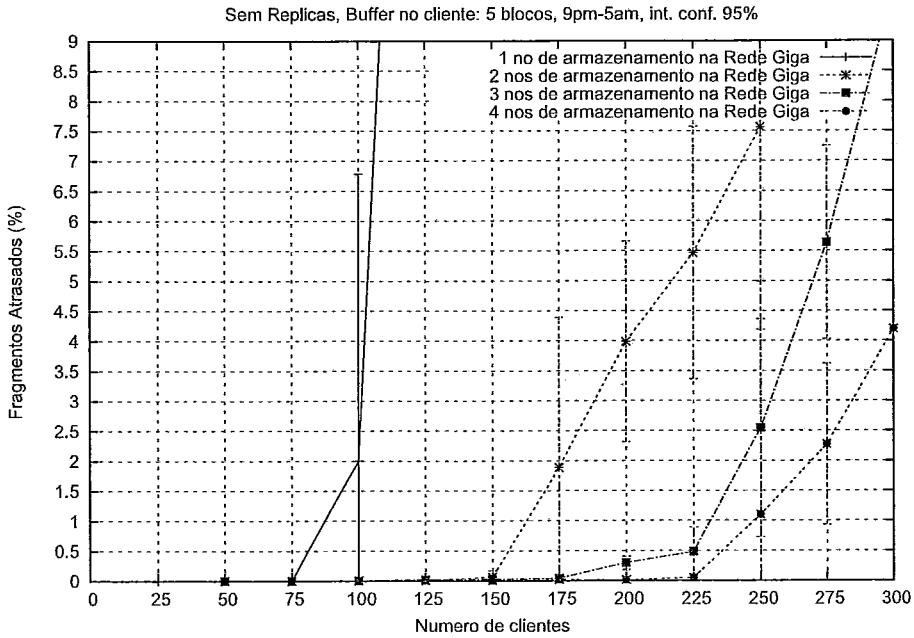

Figura 4.5: Gráfico do número de clientes x porcentagem de fragmentos atrasados com o RIO somente na rede Giga, de 1 a 4 nós de armazenamento, sem réplicas.

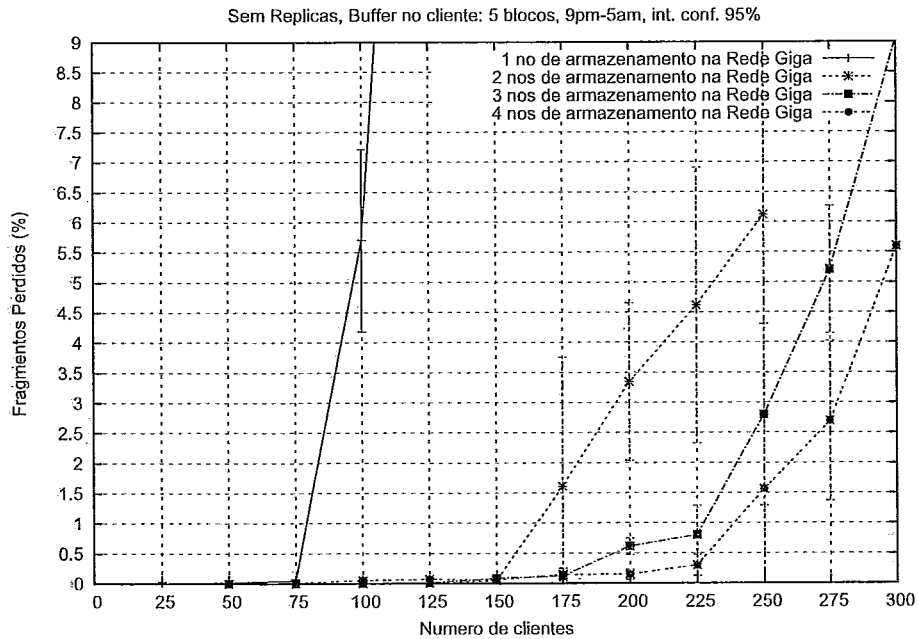

Figura 4.6: Gráfico do número de clientes x porcentagem de fragmentos perdidos com o RIO somente na rede Giga, de 1 a 4 nós de armazenamento, sem réplicas.

lanceamento da carga dos nós de armazenamento, limitações na implementação da comunicação com a interface de rede pelo kernel do Linux ao se chegar perto de seu limite, etc. É necessário relembrar que estes 80 clientes calculados seriam um limite téorico, um teto máximo que seria possível chegar. Mesmo com todos estes fatores supracitados, temos aproximadamente 60 clientes por nó de armazenamento, o que dá aproximadamente 75% do limite máximo, o que é razoável.

De 1 para 2 nós de armazenamento observamos um ganho significativo no número de médio clientes com o mesmo goodput que não se repete nas outras configurações. Acreditamos que isto comece a ocorrer devido ao fato das máquinas que executam os clientes começarem a chegar a sua capacidade de rodar clientes simultaneamente devido aos seus recursos. Podemos observar que não há predominância entre fragmentos atrasados e perdidos neste experimento porque chegamos a um média de, aproximadamente, 90 Mbps por placa de rede 100, o que é muito próximo de sua capacidade máxima e, portanto, a explicação do resultado encontrado.

4.5 Segundo Experimento: Redundância no Servidor

4.5.1 Descrição dos objetivos

O segundo experimento consistirá em averiguar qual o impacto de se utilizar replicação de blocos no desempenho geral do servidor. É desejável certificar que o gargalo não é o disco, através de um possível desbalanceamento ocorrido pela alocação aleatória dos blocos. Assim, para minorar este efeito, será configurado o sistema para utilizar sempre a replicação máxima possível. Ou seja, será repetido o teste anterior, só que sempre com uma réplica do vídeo em cada nó de armazenamento. Assim, conforme o descrito no capítulo 3, o servidor irá balancear a carga através da escolha do nó de armazenamento que possui a menor fila no momento de análise de um dado pedido de bloco por um cliente. Como não podemos replicar um mesmo bloco em um mesmo nó de armazenamento, iremos variar os nós de armazenamento, incrementalmente, de 2 até 4.

4.5.2 Descrição dos resultados esperados

A replicação não deve afetar significativamente os resultados anteriores, pois acreditamos que os discos não devem ser o gargalo do sistema. Este será a interface de rede em conjunto com o *overhead* de processamento das mensagens entre os nós. Entretanto, outros experimentos serão realizados a fim de confirmar esta hipótese.

4.5.3 Resultados e Análise

O gráfico da Figura 4.7 é composto pelo número de clientes simulados no eixo X e a porcentagem de *Goodput*, no eixo Y. Cada curva representa uma configuração distinta do sistema: o número de nós de armazenamento foi variado incrementalmente, de 2 até 4. É possível ver nos gráficos complementares: a porcentagem de fragmentos perdidos, no gráfico da Figura 4.9 e a porcentagem de fragmentos atrasados no gráfico da Figura 4.8.

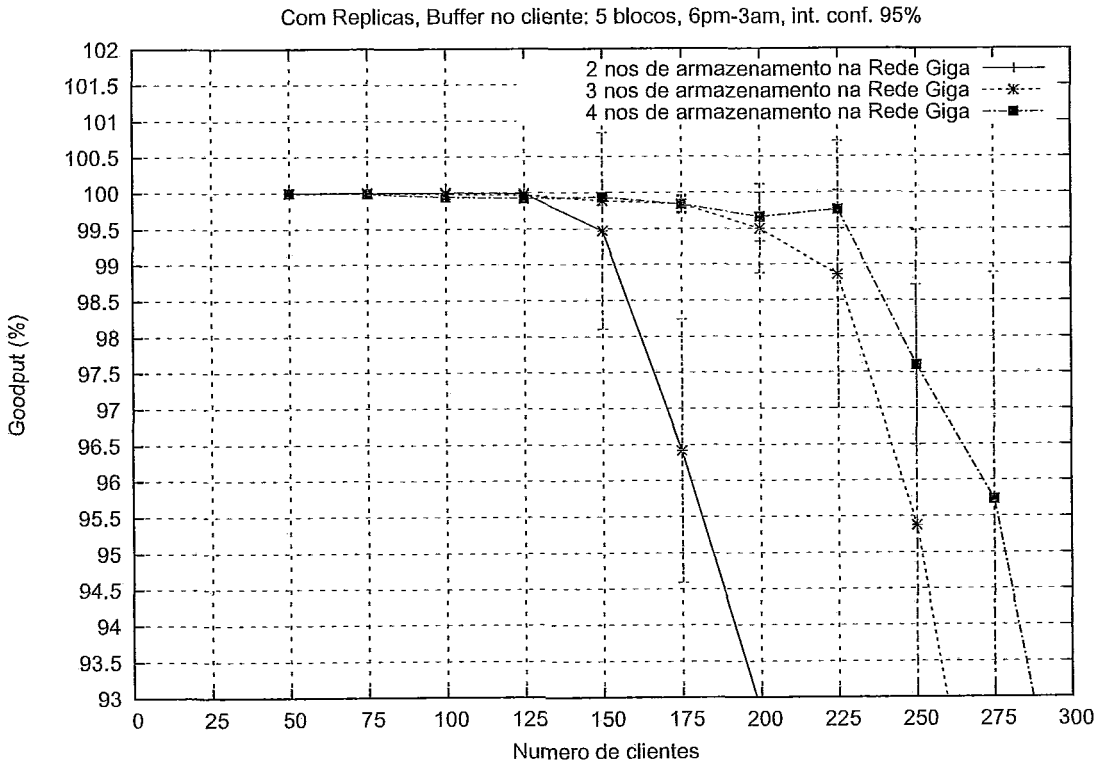

Figura 4.7: Gráfico do número de clientes x *Goodput* com o RIO somente na rede Giga, de 2 a 4 nós de armazenamento, com réplicas.

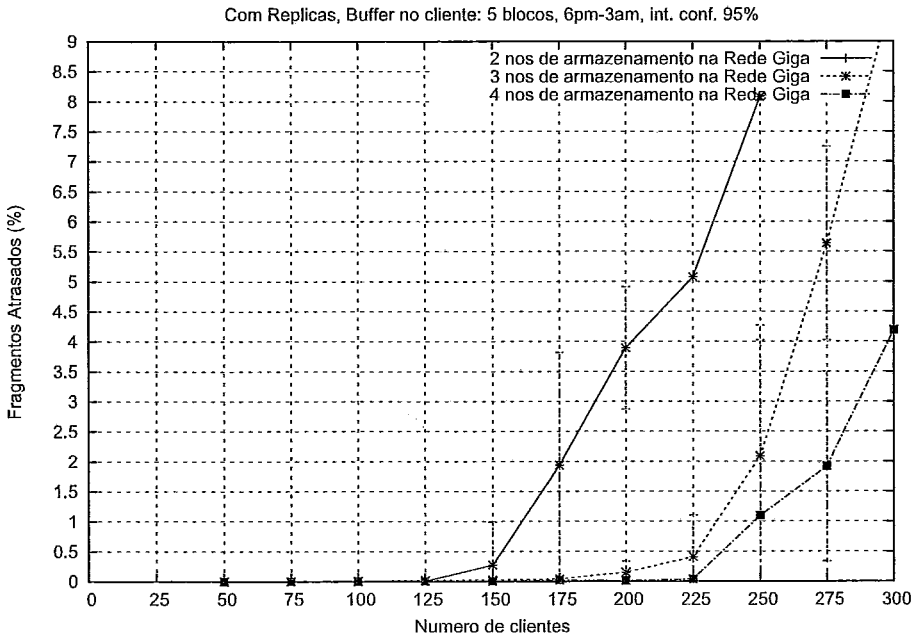

Figura 4.8: Gráfico do número de clientes x porcentagem de fragmentos atrasados com o RIO somente na rede Giga, de 2 a 4 nós de armazenamento, com réplicas.

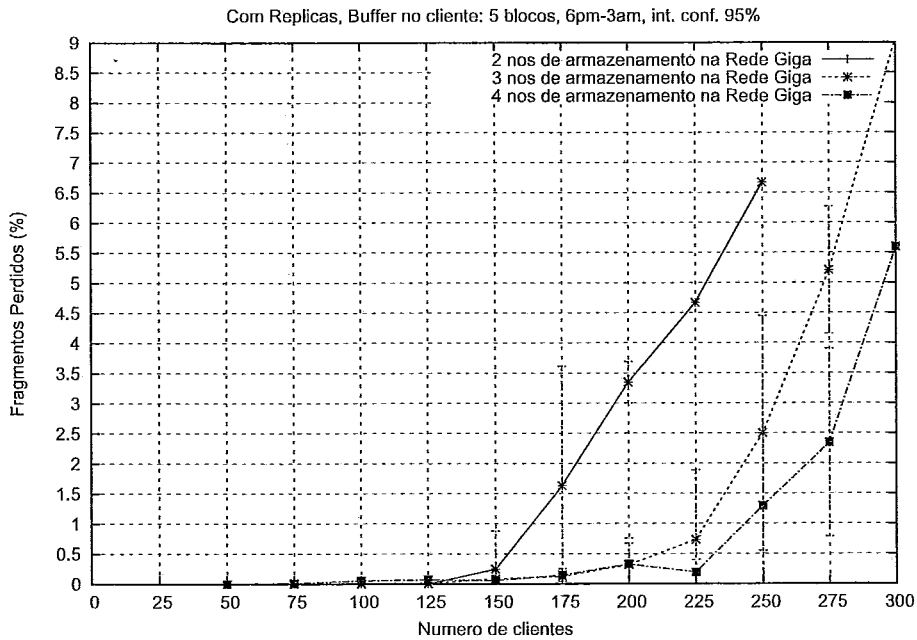

Figura 4.9: Gráfico do número de clientes x porcentagem de fragmentos perdidos com o RIO somente na rede Giga, de 2 a 4 nós de armazenamento, com réplicas.

4.5 Segundo Experimento: Redundância no Servidor

Conforme o esperado, não houve ganhos significativos com a a replicação. Poderemos melhor visualizar este fato no gráfico da Figura 4.10. Lá encontramos uma comparação entre o sistema com 4 nós de armazenamento sem replicação e com replicação. Podemos perceber que o ganho é mínimo. Além disso, é preciso relembrar que a replicação tem dois custos: o primeiro e mais óbvio é o espaço extra utilizado. O segundo e menos óbvio é o tráfego extra gerado ao se copiar o novo arquivo. Neste experimento não foi avaliado este segundo impacto, pois ele é irrelevante uma vez que as cópias dos vídeos normalmente não competem com o sistema em operação.

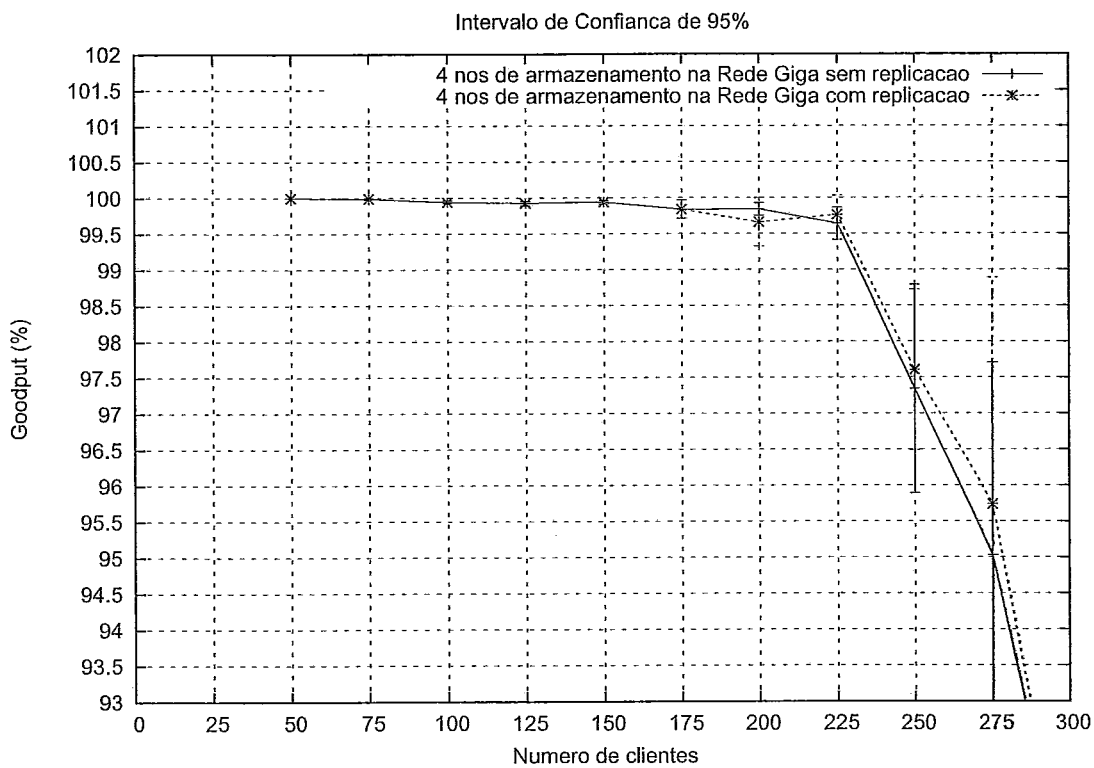

Figura 4.10: Gráfico do número de clientes x *Goodput* com o RIO somente na rede Giga, comparando 4 nós de armazenamento com e sem réplicas.

O balanceamento da carga dos discos através da replicação teoricamente diminui o atraso médio causado pelas filas em disco. Entretanto, ganhos não foram obtidos porque acreditamos que o atraso médio devido a carga total dos discos no sistema sem replicação já era absorvido pelo *playout buffer* do cliente, ou seja, este atraso não era o principal atraso no sistema. Logo, a sua diminuição não resultou em um melhor desempenho. Entretanto, a utilização de replicação confere um ganho de

confiabilidade ao sistema, uma vez que um ou mais nós de armazenamento podem falhar sem que o sistema seja afetado, se tivermos uma ou mais replicas dos vídeos e da carga do sistema no momento da falha. Nos próximos experimentos iremos tentar diminuir outro atraso que compõe o atraso total para o cliente: o gargalo da interface de rede.

4.6 Terceiro Experimento: Carga de 1 nó de armazenamento com interface gigabit

4.6.1 Descrição dos objetivos

Este experimento tem por finalidade confirmar a hipótese de que a interface de rede é o provável gargalo do sistema até agora. No experimento anterior, foi comprovado que o atraso inerente aos discos não são o gargalo. Entretanto, isto não faz automaticamente com que o atraso inerente a alta carga das interfaces de rede seja o gargalo do sistema. Para confirmar tal hipótese, seria ideal a realização dos mesmos experimentos acima com um *switch* que possuísse suficientes interfaces 1000. Infelizmente não dispúnhamos de tal equipamento. Uma solução natural foi conectar um dos nós de armazenamento na interface 1000 do summit e o servidor e os clientes nas demais interfaces 100. Contudo, isto gerou um outro conhecido problema que será tratado na próxima subseção.

Cálculo do *buffer* de um *switch*

A solução de se conectar o nó de armazenamento na porta 1000 e todos os outros nós nas portas 100 a fim de ser gerado um tráfego agregado maior é a ideal, intuitivamente. Entretanto, ela implica em um potencial problema: Como os fragmentos a serem transmitidos para cada cliente, ou seja, para cada porta 100, sempre chegam em rajadas transmitidas a 1000 Mbps, caso o summit não possua um buffer grande o suficiente para armazenar tais rajadas, a perda será inevitável. Para explicar melhor este problema, vamos simplificar o cenário e os números. Suponha 10 portas 100 para receber pacotes de uma porta 1000. Suponha que cada pacote tenha tamanho

de 100 unidades. Logo, a cada segundo chegam 10 pacotes. Se cada pacote transmitido pela porta 1000 sempre for repassado para uma porta 100 distinta, nunca teremos filas nem precisaremos de espaço em *buffer* caso o summit seja rápido o suficiente para comutar todos os dados. Ora, 1 Gbps é uma capacidade muito baixa para um summit cujo fabricante afirma comutar a 8,8 Gbps, logo isso não seria problema. O problema é o armazenamento de rajadas para uma porta. Note que, mesmo que fosse possível a utilização de algum tipo de controle de fluxo na porta 1000, para frear o envio de pacotes pela aplicação sem que haja um *overflow* de um *buffer*, isto poderia resultar na redução da vazão total. Isso, portanto, anularia a finalidade do experimento e, logo, não seria uma solução viável.

Voltando ao cenário original, tal problema foi observado porque rodamos uma simulação com poucos clientes, em que nos outros cenários sempre nos deu um Goodput muito próximo de 100% e foi obtido menos da metade deste valor. Uma vez que o fabricante não divulga a informação a respeito do tamanho do *buffer* do summit, foi necessário calculá-lo.

Para tal, foi utilizado o procedimento publicado em [60]. Resumidamente, ele consiste no seguinte: Ao gerar um tráfego CBR durante um tempo T, se a taxa com a qual este tráfego foi gerado é maior do que a taxa de transmissão do canal, então os fragmentos vão se enfileirar no *buffer* de saída do *switch*. Suponha que este tráfego é gerado a uma taxa λ e que a capacidade alocada é C. Então os fragmentos são acumulados no *buffer* a uma taxa $(\lambda - C)$ e nenhum fragmento será perdido até que o *buffer* esteja cheio. Seja τ ser o tempo necessário para encher o *buffer*. Então $(\lambda - C) * \tau$ será o tamanho do *buffer*. Além disso, depois que o *buffer* encher, a taxa de perda de fragmentos é de $(\lambda - C)$, uma vez que o tráfego está sendo gerado a uma taxa constante.

Assumindo que o tráfego está sendo gerado durante o tempo $(0, t)$, se $t < \tau$ então nenhuma perda será observada em $(0, t)$. Seja t_1 e t_2 tais que $t_2 > t_1 > \tau$. Seja l_1 e l_2 o número de bytes perdidos durante $(0, t_1)$ e $(0, t_2)$, respectivamente. Então devemos ter $\frac{(l_2 - l_1)}{(t_2 - t_1)} = \lambda - C$, e, assim, obter facilmente $\lambda - C$ e τ. Portanto, teremos: tamanho do *buffer* $= \frac{(l_2 - l_1)}{(t_2 - t_1)} * t_2 - l_2$

Dado o exposto acima, utilizando os valores $t_1 = 2{,}5$ seg, $t_2 = 3{,}9$ seg, $l_1 =$

1789 kilobytes e $l_2 = 2913$ kilobytes. Assim, calculamos que o *buffer* do summit é de aproximadamente 218,14 kilobytes, o que armazenaria 153,38 fragmentos que correspondem a aproximadamente 1,7 blocos. Um cliente que assiste a um vídeo sequencialmente certamente irá pedir mais do que 1,7 blocos consecutivamente, logo assim temos explicada a alta taxa de perda encontrada na primeira tentativa do terceiro experimento.

4.6.2 Descrição dos objetivos: Continuação

Observada esta limitação do ambiente de testes, foi-se obrigado a passar a utilizar um novo *switch* que tivesse portas 1000 suficientes para testar o nó de armazenamento com diversos clientes. Assim, escolheu-se um *switch* da DELL modelo PowerConnect 2724 diponível que atendia a tais requisitos. Basicamente, ele é idêntico ao summit: sua única diferença fundamental é somente possuir portas 1000. Eliminamos, desta forma, o problema descrito anteriormente.

O teste consistiu em verificar a carga máxima gerada para um nó de armazenamento de forma que as filas em disco começassem a gerar atraso e perda. Para tal, utilizamos 6 máquinas para executar os clientes e uma para executar o nó servidor e o nó de armazenamento, para também assim eliminar o custo de transmissão das mensagens entre ambos.

4.6.3 Descrição dos resultados esperados

Como a hipótese a ser analisada com este experimento é se o gargalo é a interface de rede, é esperado um aumento da quantidade de clientes sem que haja perdas ou atrasos em volume significativo, uma vez que eliminamos todos os componentes do atraso dos outros cenários, que são o retardo na rede, o atraso das mensagens entre os nós e a capacidade das interfaces de rede. Os dois primeiros atrasos são necessáriamente pequenos em uma rede Giga. Na próxima seção veremos estes atrasos com mais detalhes. Entretanto, a saturação devido à utilização da interface de rede perto de sua capacidade nominal gera erros e perdas cada vez maiores. Portanto, com a ampliação deste gargalo, é plausível esperar uma substancial melhora no desempenho do sistema.

4.6.4 Resultados e Análise

O gráfico da Figura 4.11 é composto pelo número de clientes simulados no eixo X e a porcentagem de *Goodput*, no eixo Y. Os gráficos complementares são: a porcentagem de fragmentos perdidos, no gráfico da Figura 4.13 e a porcentagem de fragmentos atrasados no gráfico da Figura 4.12.

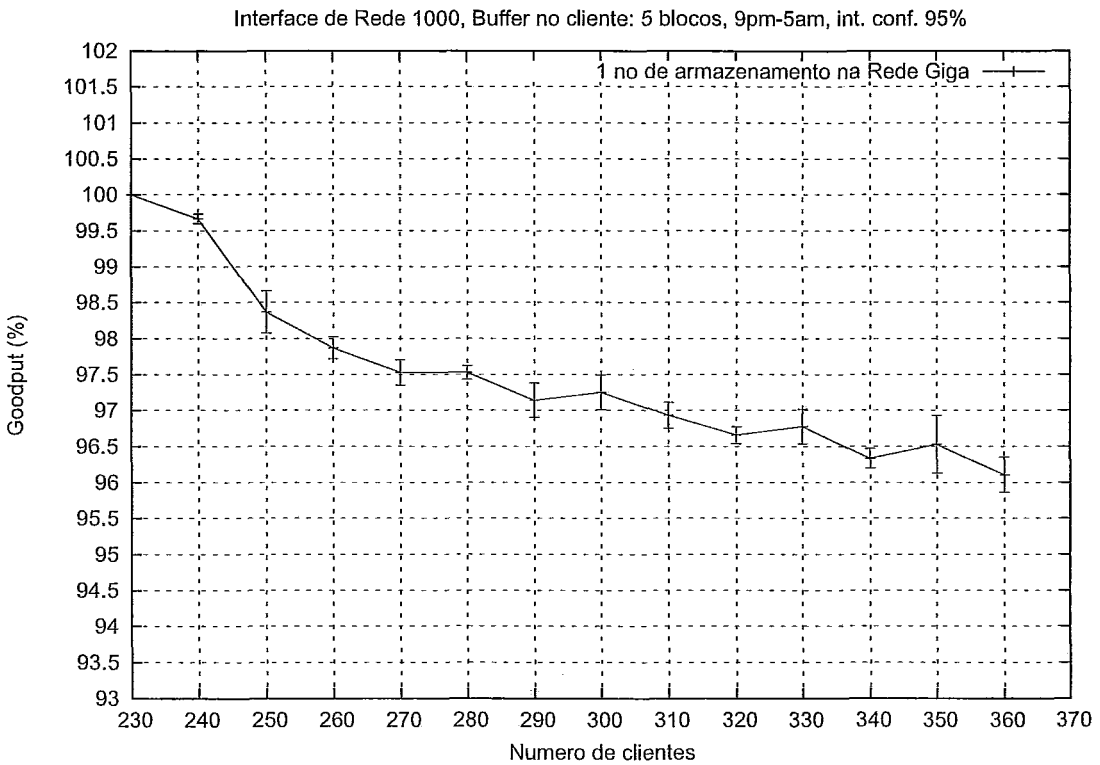

Figura 4.11: Gráfico do número de clientes x *Goodput* com o RIO em uma interface de rede 1000, 1 nó de armazenamento.

É possível notar uma drástica melhora na quantidade de clientes servidos sem perda na qualidade neste cenário em relação aos experimentos anteriores. A perda gradual pode ser explicada com o fato de que o servidor e nó de armazenamento estavam sendo executados na mesma máquina, o que onerou demasiadamente a CPU do servidor, que se tornou o gargalo no sistema neste cenário. Mesmo assim, somente no gráfico da Figura 4.14, apenas com 4 nós de armazenamento e utilizando replicação consegue-se um desempenho similar ao obtido com interfaces de rede de maior capacidade. Portanto, podemos concluir que o gargalo do sistema são as interfaces de rede e sim a CPU, conforme anteriormente descrito. Logo, para

4.6 Terceiro Experimento: Carga de 1 nó de armazenamento com interface gigabit

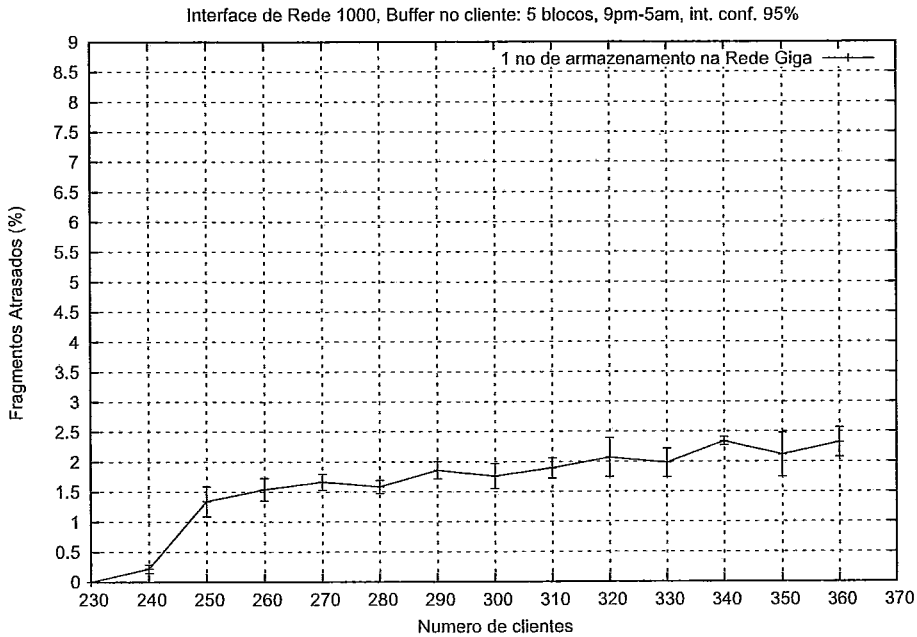

Figura 4.12: Gráfico do número de clientes x porcentagem de fragmentos atrasados com o RIO em uma interface de rede 1000, 1 nó de armazenamento.

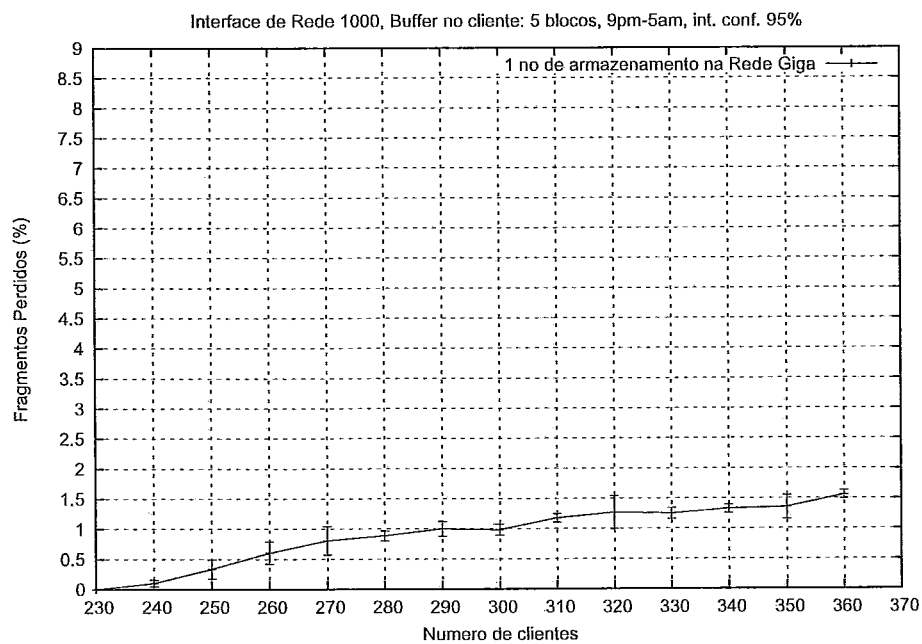

Figura 4.13: Gráfico do número de clientes x porcentagem de fragmentos perdidos com o RIO em uma interface de rede 1000, 1 nó de armazenamento.

aumentar sua capacidade, ou utilizamos um equipamento com interfaces 1000 ou passamos a colocar mais nós de armazenamento, a fim de aumentar a capacidade agregada destas interfaces. Nas próximas seções, avaliaremos o desempenho do RIO com nós de armazenamento em redes com características bem distintas.

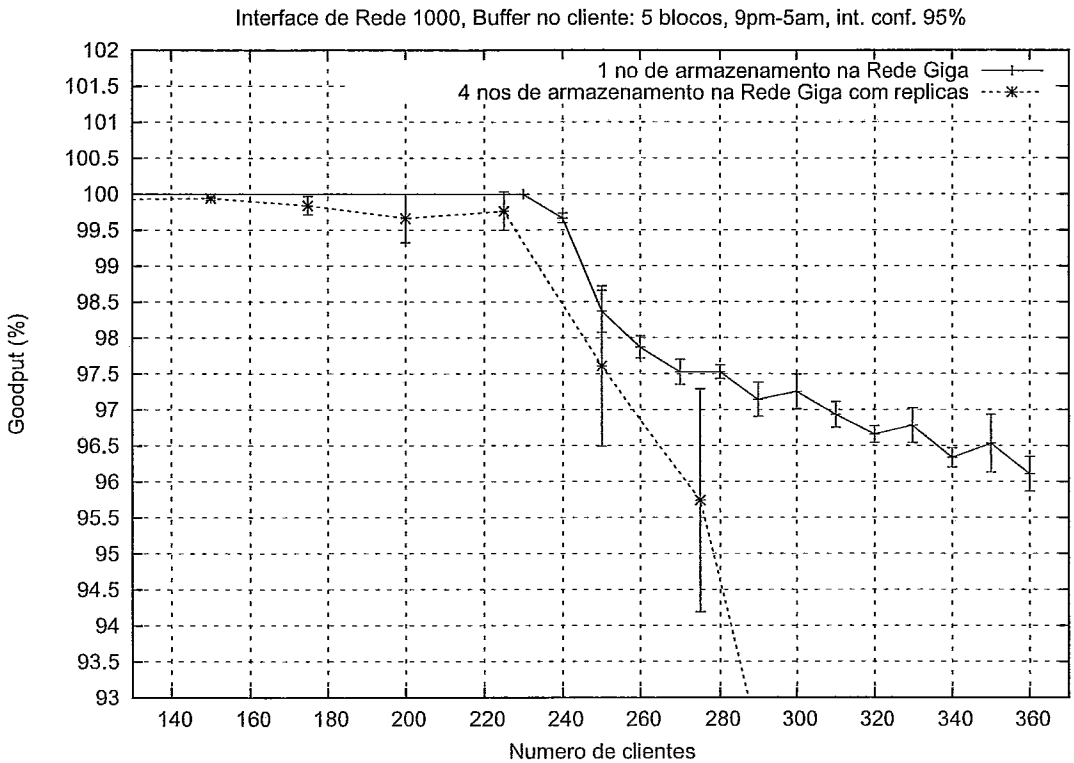

Figura 4.14: Gráfico do número de clientes x goodput com o RIO em uma interface de rede 1000, 1 nó de armazenamento, e o RIO na rede Giga, com 4 nós de armazenamento e replicação.

4.7 Quarto Experimento: Análise dos caminhos entre os nós de armazenamento e os clientes

4.7.1 Descrição dos objetivos

Conforme a motivação feita no final da análise do experimento anterior, o objetivo dos experimentos a partir de agora é analisar o comportamento do sistema quando um dos nós de armazenamento só pode ser acessado através de um cam-

inho com características notoriamente diferentes dos demais. Mais especificamente, deseja-se saber se um nó de armazenamento com uma conexão de rede significativamente inferior as demais irá ou não comprometer o desempenho do sistema. Por inferior entende-se um caminho com atraso e perdas médias significativamente maiores do que as dos demais caminhos.

Este experimento consistirá em três cenários:

1. Um cliente na UFRJ acessará um servidor na UFRJ e um nó de armazenamento na UFF, através da rede Giga;

2. Um cliente na UFRJ acessará um servidor na UFRJ e um nó de armazenamento na UFMG, através da conexão Internet já existente nestas duas instituições;

3. Utilizando as ferramentas de geração de tráfego do TANGRAM II [61], será calculado o gargalo da conexão Internet a ser utilizada nos testes;

Para os dois primeiros cenários, realizamos os experimentos em duas etapas: a primeira etapa é exatamente igual a descrita na seção 4.3, com a emulação de apenas um cliente. Queremos, com isso, avaliar o RTT: o tempo entre o pedido de um bloco e a sua chegada no sistema. A segunda etapa consiste em analisar o arquivo de log gerado para o cálculo da média, variância, coeficiente de variação e o intervalo de confiança das medidas.

No terceiro cenário, foi utilizado o módulo de geração de tráfego do TANGRAM II, que utiliza um método de pares de pacotes (*packet pair*) para cálculo de gargalo, baseado na proposição inicial de Keshav em [62]. O gargalo foi calculado durante o quinto experimento e durante outros testes: os resultados obtidos sempre foram muito próximos, com uma variação muito pequena, quase desprezível.

4.7.2 Descrição dos resultados esperados

Neste experimento sabemos que o caminho via Internet é muito inferior do que o caminho via rede Giga. Por inferior entende-se um caminho com atraso e perdas médias significativamente maiores do que as dos demais caminhos. O que queremos auferir é a diferença entre eles. É esperado uma diferença de uma ordem de

Tabela 4.1: Comparação entre os RTTs através da aplicação que foram medidos na rede Giga e na Internet entre a UFRJ e UFMG, em milissegundos.

	Média	Variância	Maior	Coeficiente de Variação	Intervalo de Confiança de 95%
RTT de 1 cliente na UFRJ, o servidor na UFRJ e o nó de armazenamento na UFF, via rede Giga	50.5286	52.8745	2205.85	4550.78	5.58663
RTT de 1 cliente na UFRJ, o servidor na UFRJ e o nó de armazenamento na UFMG, via Internet	376.395	995.441	4296.08	2650.72	24.2401

grandeza entre as médias dos RTTs. Para o gargalo, espera-se algo em torno de 50 Mbps devido a capacidade de alguns canais que compõe o caminho entre as duas intituições.

4.7.3 Resultados e Análise

A capacidade do gargalo entre a UFRJ e a UFMG, em ambos os sentidos, é idêntica: 71,32 Mbps. Este valor foi um pouco superior ao esperado. Na tabela 4.1 temos o RTT do primeiro cenário na primeira linha e o RTT do segundo cenário na segunda linha. As grandezas auferidas encontram-se listadas nas columas homônimas.

Os dados da tabela 4.1 confirmam o esperado. Existe uma diferença de aproximadamente uma ordem de grandeza entre os RTTs das duas redes. A qualidade do serviço do cliente será afetada caso o *playout buffer* não absorva a variabilidade destes RTTs. Como, na UFMG, o maior RTT coletado foi de 4,3 segundos, foi decidido utilizar um *buffer* de 5 blocos, portanto, com capacidade de armazenar, em média, 4,2 segundos do vídeo utilizado nos testes. Este valor é mais do que suficiente em face a variabilidade deste RTT sem aumentar muito o atraso de inicialização (*startup delay*) do vídeo para o cliente.

4.8 Quinto Experimento: RIO na UFMG

4.8.1 Descrição dos objetivos

Uma vez tendo estudado o caminho para a UFMG, o quinto experimento consistirá em averiguar se existe algum benefício em se ter um nó de armazenamento conectado a uma rede de qualidade muito inferior aos demais nós. Também deseja-se saber se teremos ganhos em todas as configurações anteriores, quando o número de nós de armazenamento variar.

Para tal, será repetido o primeiro experimento, só que sempre acrescentando um nó de armazenamento na UFMG para cada configuração prévia do servidor. Ou seja, enquanto que no primeiro experimento foi variado incrementalmente de 1 a 4 nós de armazenamento, neste experimento faremos o mesmo, só que com 2 a 5 nós de armazenamento, sempre mantendo um nó na UFMG. Também não utilizaremos replicação neste experimento.

4.8.2 Descrição dos resultados esperados

Mesmo com um caminho inferior, acreditamos que o nó de armazenamento da UFMG irá ajudar no desempenho geral do servidor. Não temos idéia de quanto, mas sabemos que não teremos os mesmos ganhos do que um nó na rede Giga.

4.8.3 Resultados e Análise

O gráfico da Figura 4.15 é composto pelo número de clientes simulados no eixo X e a porcentagem de *Goodput*, no eixo Y. Cada curva representa uma configuração distinta de nós de armazenamento. Os gráficos complementares são: a porcentagem de fragmentos perdidos, no gráfico da Figura 4.17 e a porcentagem de fragmentos atrasados no gráfico da Figura 4.16.

Pode-se observar um ganho na utilização do nó extra, tal qual no primeiro experimento. Entretanto, neste cenário, mesmo com poucos usuários, existem perdas. Isto é devido às perdas inerentes do caminho para a UFMG: independentemente da carga sendo gerada pelo sistema, a perda no canal existe e é significativa. Considerando que a taxa média de perda para a UFMG é de 4%, a Figura 4.18 traça o

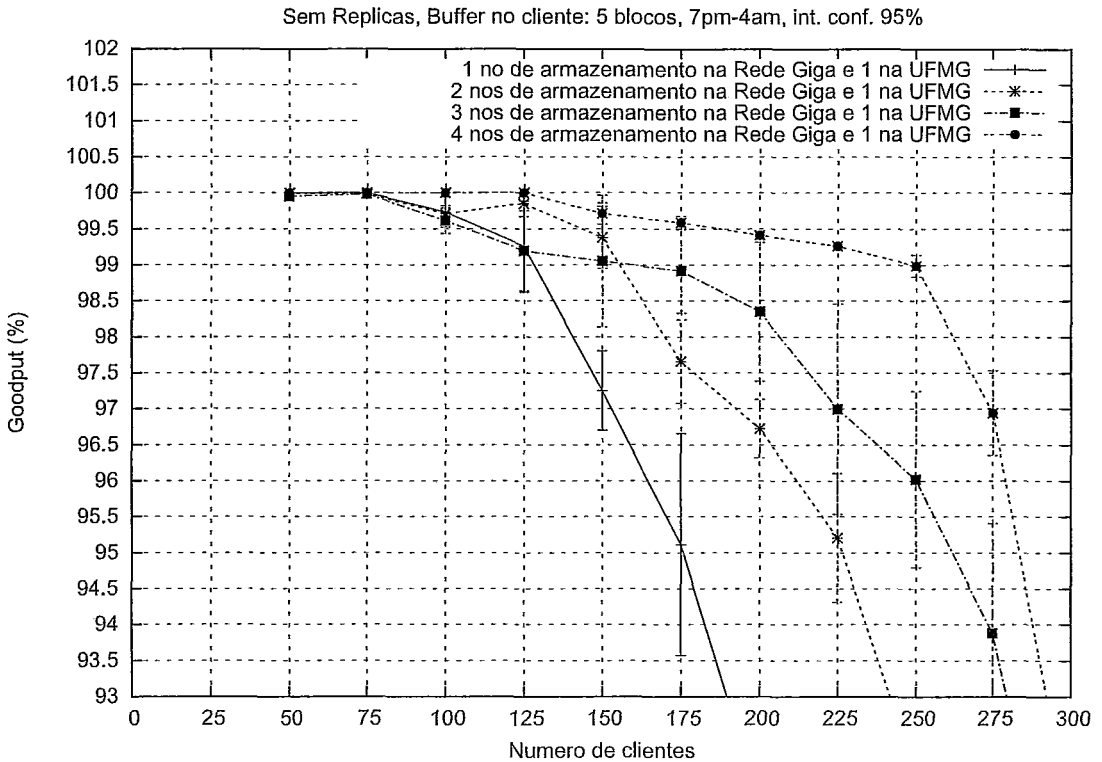

Figura 4.15: Gráfico do número de clientes x *Goodput* com o RIO na rede Giga e sempre com 1 nó na UFMG, de 2 a 5 nós de armazenamento, sem réplicas.

gráfico comparando o número de clientes máximo com um *goodput* de até 96% entre as configurações de nós de armazenamento somente na rede Giga e com o nó extra na UFMG. É claro também que o ganho com um nó de armazenamento extra diminuirá de acordo com o aumento do número total de nós de armazenamento do sistema e podemos também ver isto na Figura 4.18. Isto é verdade porque a adição do nó de armazenamento na UFMG representa, cada vez menos, uma contribuição significativa a capacidade total dos nós de armazenamento. Por exemplo, quando temos apenas 1 nó na rede Giga, ao adicionarmos outro nó de armazenamento na UFMG estaremos dobrando a capacidade total dos nós de armazenamento. Quando temos 2 nós de armazenamento na rede Giga, a adição de outro nó na UFMG só aumenta pela metade a capacidade total dos nós de armazenamento do sistema. Portanto, é natural esperar uma contribuição menor do nó, neste último caso, para o número de usuários servidos com o mesmo *goodput* do que no primeiro caso, uma vez que a proporção dos blocos armazenados pelo nó extra na UFMG também diminui.

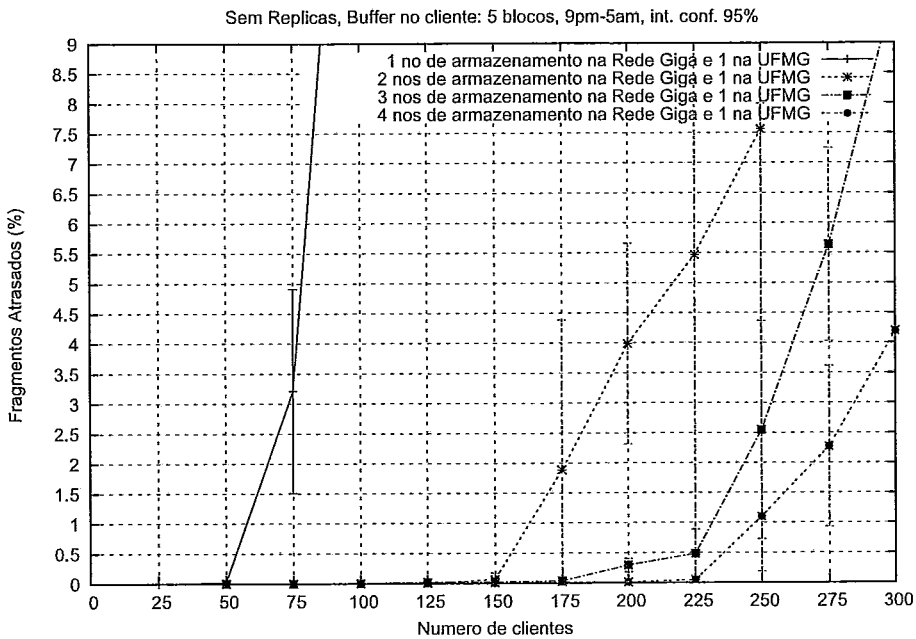

Figura 4.16: Gráfico do número de clientes x porcentagem de fragmentos atrasados com o RIO na rede Giga e sempre com 1 nó na UFMG, de 2 a 5 nós de armazenamento, sem réplicas.

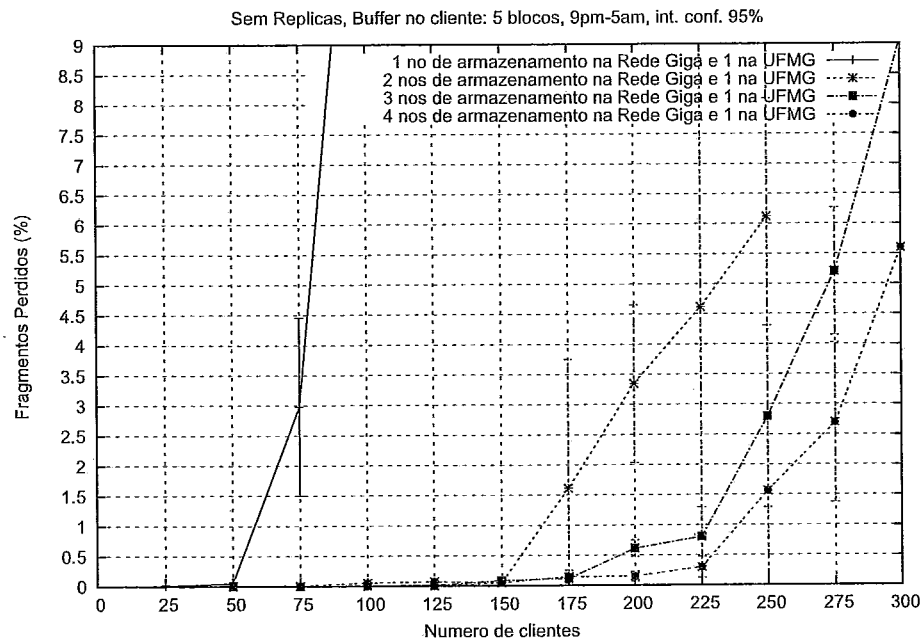

Figura 4.17: Gráfico do número de clientes x porcentagem de fragmentos perdidos com o RIO na rede Giga e sempre com 1 nó na UFMG, de 2 a 5 nós de armazenamento, sem réplicas.

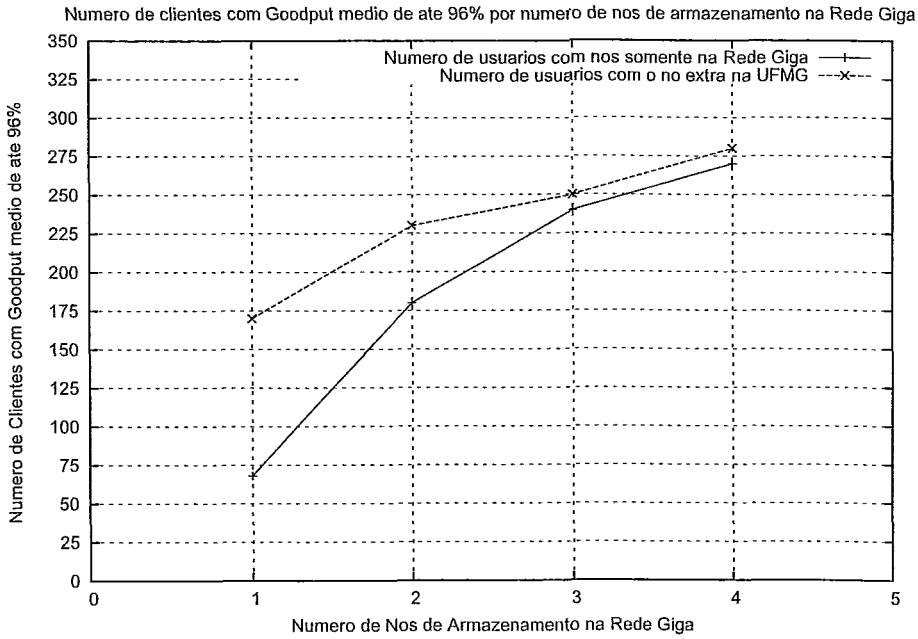

Figura 4.18: Gráfico do número de clientes com *goodput* de até 96% x número de nós de armazenamento somente na rede Giga em comparação com a mesma configuração adicionando-se o nó extra na UFMG, sem réplicas.

4.9 Sexto Experimento: RIO na UFMG com redundância

4.9.1 Descrição dos objetivos

Com a mesma intenção do experimento anterior, deseja-se agora repetir o segundo experimento mas sempre tendo um nó de armazenamento na UFMG. E, tal qual o segundo experimento, deseja-se utilizar réplicas para os blocos de vídeos. Será variado o número de nós de armazenamento, de 2 a 5, sempre mantendo um nó na UFMG. Replicação será utilizada, logo, cada nó de armazenamento terá uma cópia do vídeo.

4.9.2 Descrição dos resultados esperados

Com o que foi observado no segundo experimento, não acreditamos que a replicação irá influir no desempenho final do cenário e teremos resultados muito semelhantes aos obtidos no experimento anterior.

4.9.3 Resultados e Análise

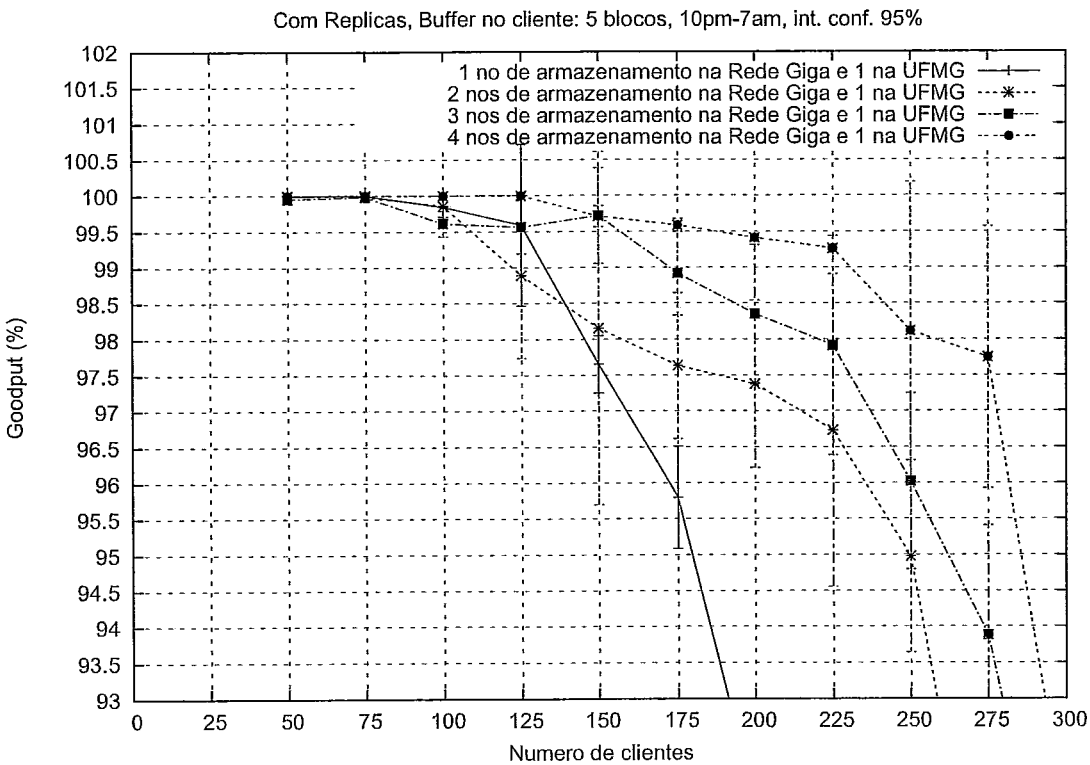

Figura 4.19: Gráfico do número de clientes x *Goodput* com o RIO na rede Giga e sempre com 1 nó na UFMG, de 2 a 5 nós de armazenamento, com replicações.

O gráfico da Figura 4.19 é composto pelo número de clientes simulados no eixo X e a porcentagem de *Goodput* no eixo Y. Os gráficos complementares são: a porcentagem de fragmentos perdidos, no gráfico da Figura 4.21 e a porcentagem de fragmentos atrasados no gráfico da Figura 4.20.

Novamente obteve-se pequenos ganhos com a replicação. Seria possível conseguir ganhos maiores se o algoritmo de balanceamento de carga do RIO levasse em conta não apenas a fila em disco, mas também propriedades do canal como a sua taxa de perda. Desta forma, somente é possível implementar uma configuração com caminhos determinísticos, já que o algoritmo de balanceamento de carga não leva em conta variáveis aleatórias como a taxa de perda do canal. Este cenário é semelhante ao obtido com *diversidade de caminhos* [55].

4.9 Sexto Experimento: RIO na UFMG com redundância

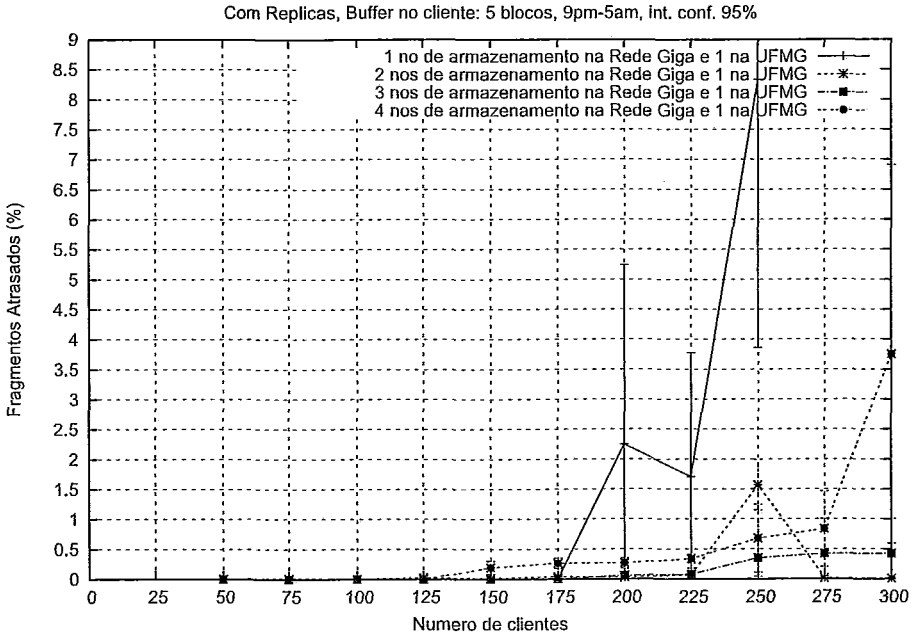

Figura 4.20: Gráfico do número de clientes x porcentagem de fragmentos atrasados com o RIO na rede Giga e sempre com 1 nó na UFMG, de 2 a 5 nós de armazenamento, com replicações.

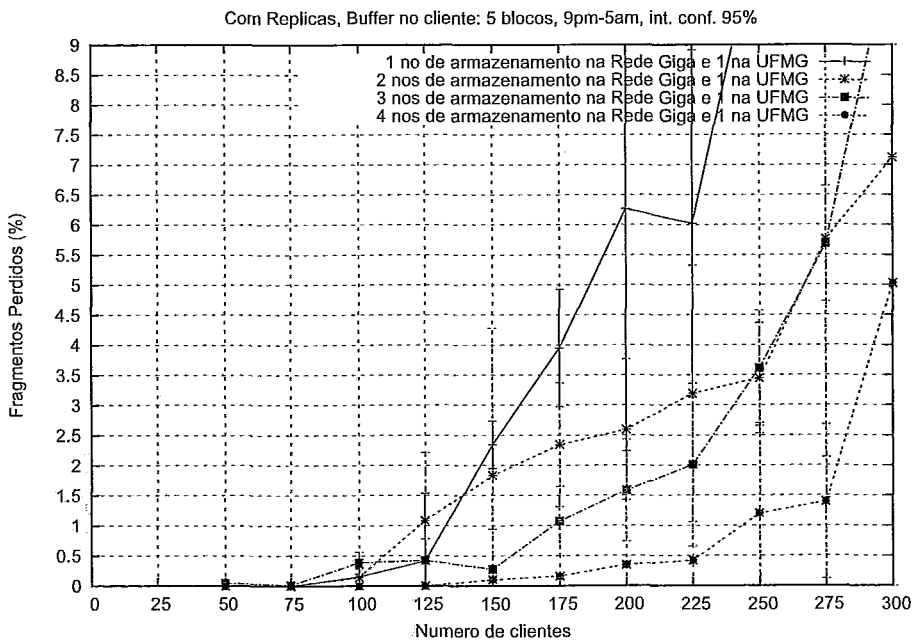

Figura 4.21: Gráfico do número de clientes x porcentagem de fragmentos perdidos com o RIO na rede Giga e sempre com 1 nó na UFMG, de 2 a 5 nós de armazenamento, com replicações.

4.10 Comparação entre o RIO na rede Giga e com o nó extra na UFMG

4.10.1 Descrição dos objetivos

Esta seção tem como objetivo comparar o desempenho do servidor RIO somente na rede Giga com o nó extra na UFMG, com base nos resultados anteriores. Deseja-se tentar auferir qual o ganho de se adicionar um nó de armazenamento mesmo sendo o caminho entre o nó de armazenamento e o cliente com características de perda e retardo piores em relação a cada configuração existente, sem réplicas.

Para cada configuração serão mostrados dois gráficos: o primeiro comparando uma configuração com outra com um nó extra; o segundo comparando configurações com o mesmo número de nós de armazenamento, a qual chamaremos de "configuração ideal". Por exemplo, será comparado qual foi o ganho de se utilizar 3 nós da rede Giga com a utilização de 3 nós na rede Giga e 1 nó na UFMG. Quer-se saber o ganho com o nó extra. Depois será feita a comparação da configuração de 3 nós na rede Giga e 1 na UFMG com a configuração de 4 nós na rede Giga, a fim de saber quão próximo chegou-se da configuração ideal.

4.10.2 Descrição dos resultados esperados

Pelo o que foi visto até agora, é natural esperar que o nó na UFMG traga ganhos para o desempenho do sistema, de um modo geral. O esperado é que este ganho seja um pouco inferior de se ter um nó na rede Giga, mas sem grandes discrepâncias.

4.10.3 Resultados e Análise

O gráfico da Figura 4.22 mostra um ganho de 88,3% em número de usuários servidos, de 85 para 160 com o nó extra na UFMG, considerando um *Goodput* mínimo de aproximadamente 96%. A configuração ideal seria ter, neste caso, os dois nós na rede Giga. É possível observar no gráfico da Figura 4.23 que chegamos a 89% desta configuração, pois 2 nós na rede Giga nas mesmas condições servem a 180 clientes, enquanto que, com 1 nó na rede Giga e com o nó extra na UFMG,

servimos 160 clientes.

No gráfico da Figura 4.24 temos um ganho de 19,5% em número de usuários servidos, de 180 para 215, ao adicionar o nó extra na UFMG aos 2 nós de armazenamento na rede Giga e considerando um *Goodput* mínimo de até 96%. O gráfico da Figura 4.25 demonstra que esta configuração chega a 91,5% da configuração ideal, que consistiria em comparar 3 nós na rede Giga, que servem a 235 clientes, com 2 nós na rede Giga e o nó extra na UFMG, configuração que atende a 215 clientes, com o mesmo *Goodput* mínimo de 96%.

No gráfico da Figura 4.26, observamos um ganho de 8,5% em número de usuários servidos, de 235 para 255 clientes, com a adição do nó extra na UFMG aos 3 nós de armazenamento existentes na rede Giga e com um *Goodput* de até 96%. Com o gráfico da Figura 4.27, observa-se que chegamos a 94,45% da configuração ideal, que seria termos 4 nós de armazenamento na rede Giga, que servem 270 clientes, em comparação com os 3 nós na rede Giga e o nó extra na UFMG, que atende a 255 clientes, respeitando o mesmo *Goodput* mínimo de 96%.

Por fim, no gráfico da Figura 4.28 temos um ganho de 3,7% no número de clientes do sistema quando comparamos 4 nós na rede Giga, que servem a 270 clientes, com os mesmos 4 nós em conjunto com o nó extra na UFMG, que atendem a 280 clientes, com um *Goodput* mínimo de até 96%. Neste caso, devido ao limite de número de equipamentos, este é o teste com o maior número de nós de armazenamento e não foi possível compará-lo com a sua configuração ideal, que seria termos os 5 nós de armazenamento somente na rede Giga.

4.10 Comparação entre o RIO na rede Giga e com o nó extra na UFMG

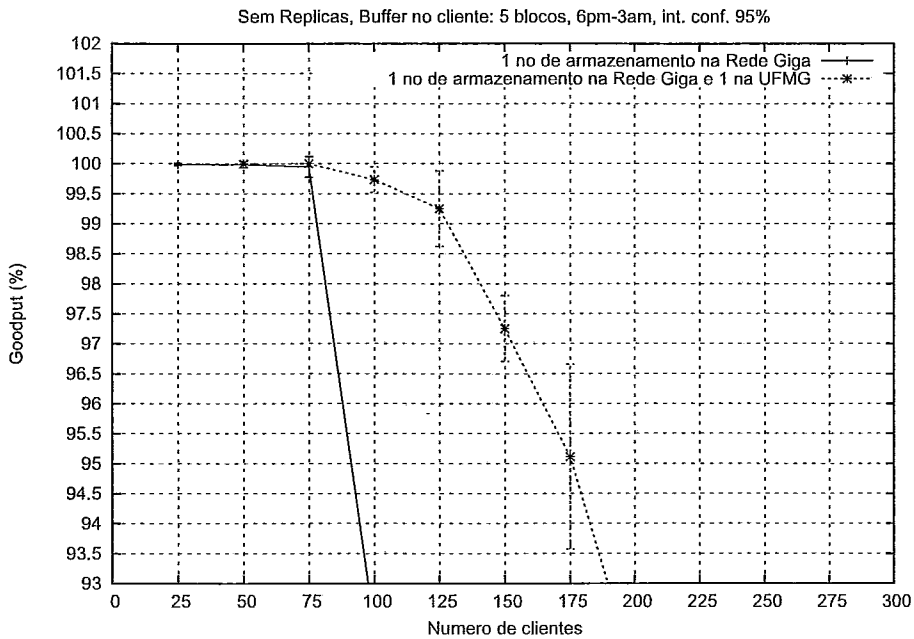

Figura 4.22: Gráfico do número de clientes x *Goodput* com o RIO com 1 nó na rede Giga em comparação com 1 nó na rede Giga e 1 nó na UFMG, sem replicações.

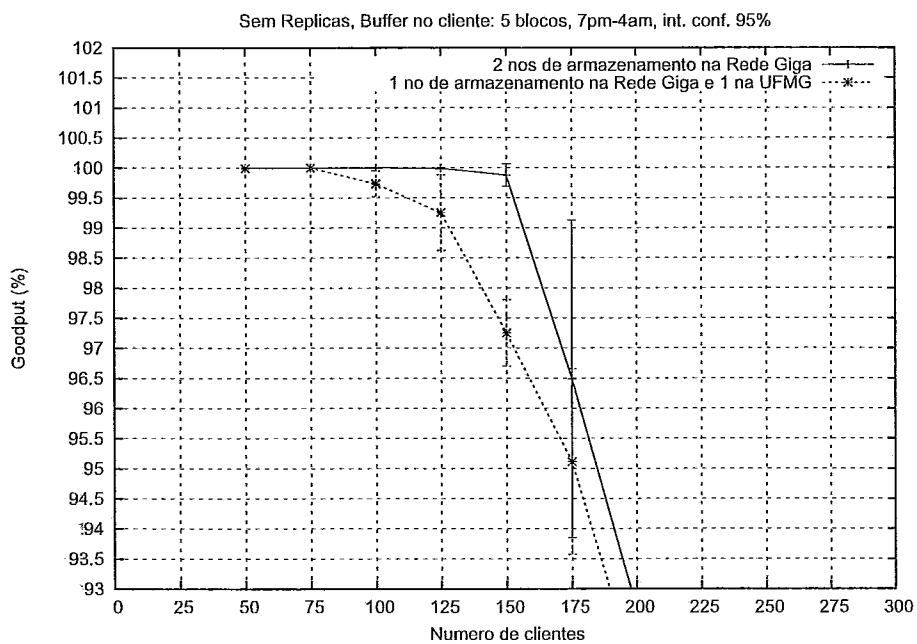

Figura 4.23: Gráfico do número de clientes x *Goodput* com o RIO com 2 nós na rede Giga em comparação com 1 nó na rede Giga e 1 nó na UFMG, sem replicações.

4.10 Comparação entre o RIO na rede Giga e com o nó extra na UFMG

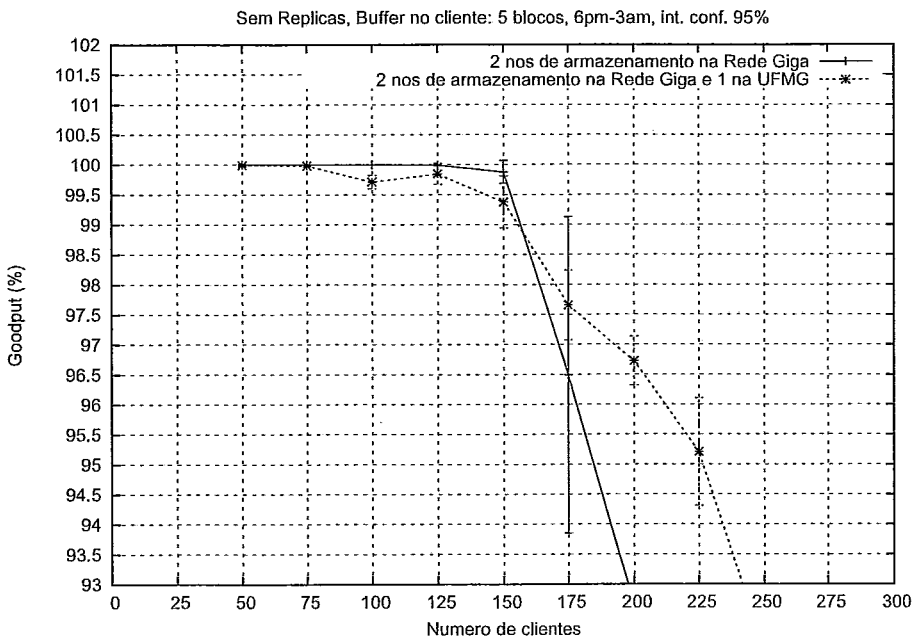

Figura 4.24: Gráfico do número de clientes x *Goodput* com o RIO com 2 nós na rede Giga em comparação com 2 nós na rede Giga e 1 nó na UFMG, sem replicações.

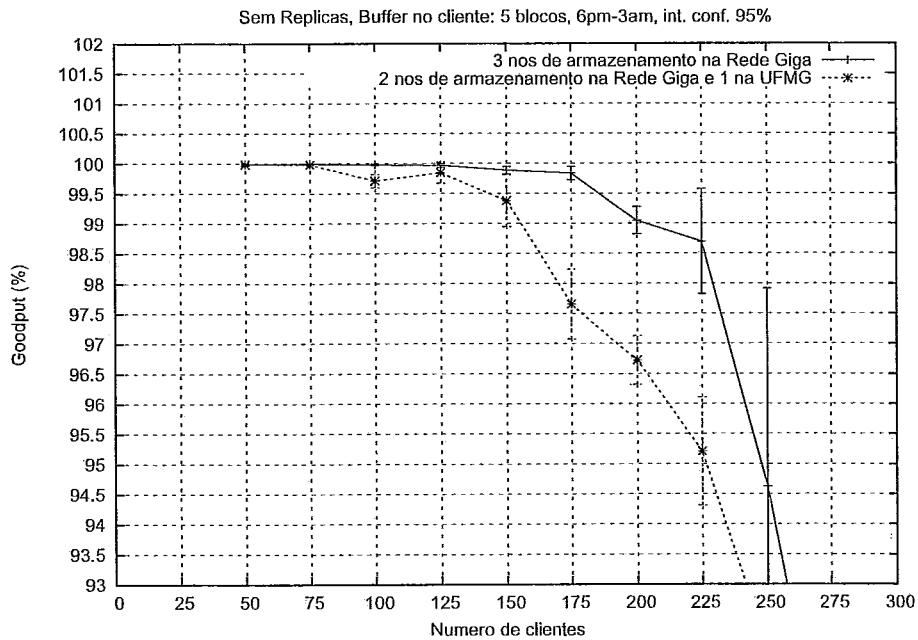

Figura 4.25: Gráfico do número de clientes x *Goodput* com o RIO com 3 nós na rede Giga em comparação com 2 nós na rede Giga e 1 nó na UFMG, sem replicações.

4.10 Comparação entre o RIO na rede Giga e com o nó extra na UFMG

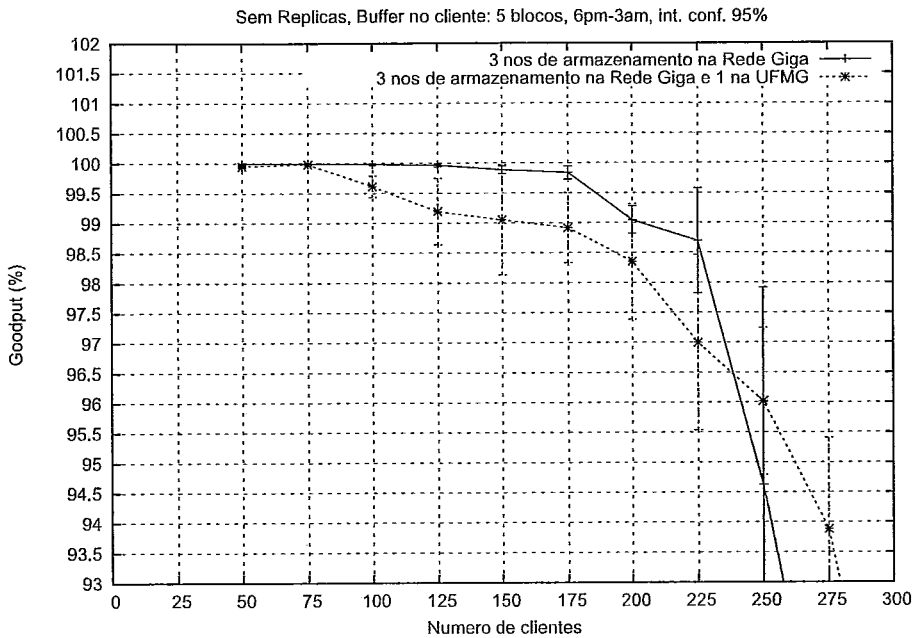

Figura 4.26: Gráfico do número de clientes x *Goodput* com o RIO com 3 nós na rede Giga em comparação com 3 nós na rede Giga e 1 nó na UFMG, sem replicações.

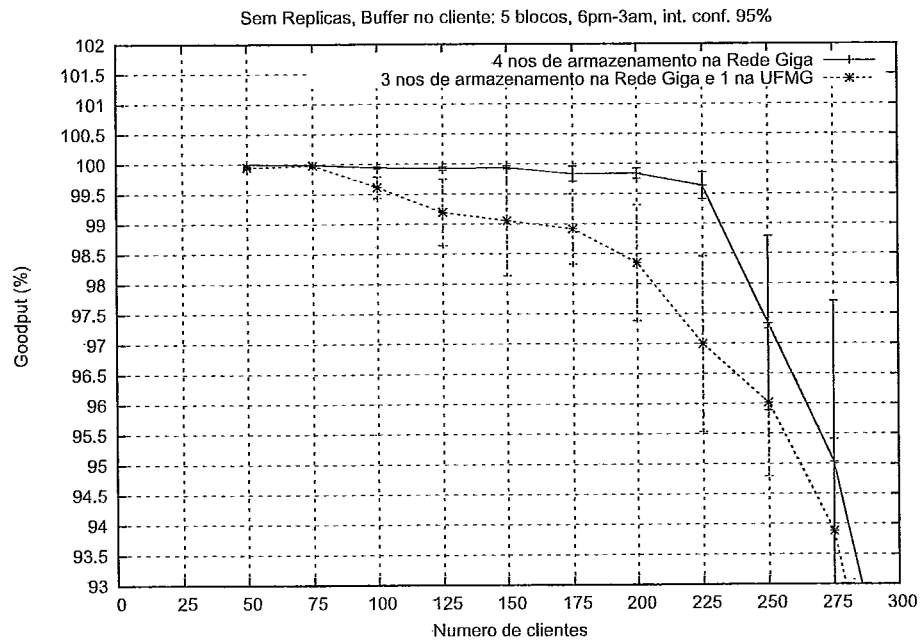

Figura 4.27: Gráfico do número de clientes x *Goodput* com o RIO com 4 nós na rede Giga em comparação com 3 nós na rede Giga e 1 nó na UFMG, sem replicações.

4.10 Comparação entre o RIO na rede Giga e com o nó extra na UFMG

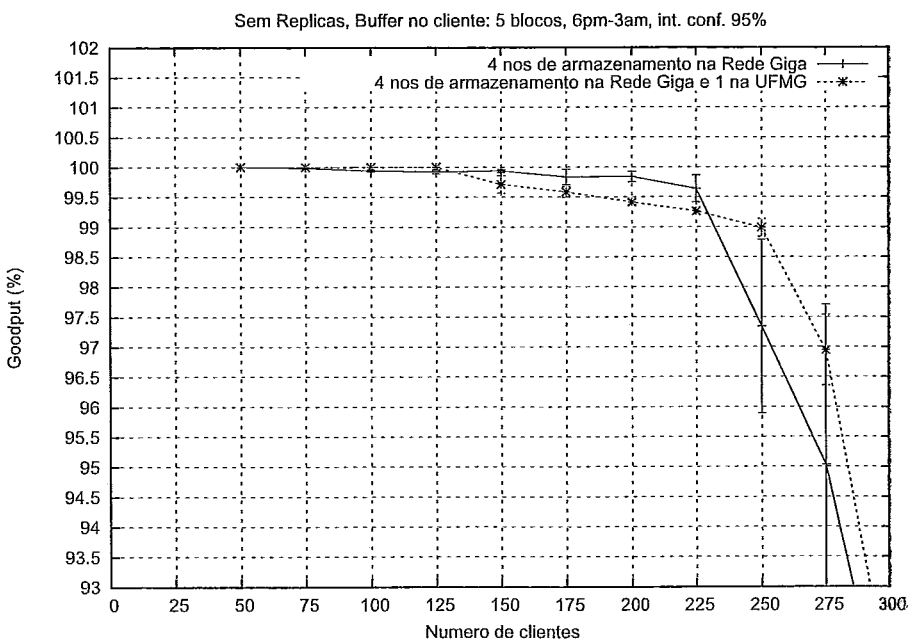

Figura 4.28: Gráfico do número de clientes x *Goodput* com o RIO com 4 nós na rede Giga em comparação com 4 nós na rede Giga e 1 nó na UFMG, sem replicações.

4.11 Considerações sobre a escalabilidade do servidor RIO

Para que o servidor RIO seja escalável, podendo servir a um número crescente de clientes pelo aumento do número de nós de armazenamento, o nó servidor não pode se tornar o gargalo do sistema. Portanto, para investigarmos se o RIO é ou não escalável, devemos auferir a taxa de ocupação do processador e o consumo de memória RAM no nó servidor quando aumentamos o número de clientes no sistema. Com este fim, utilizamos uma configuração do servidor com 4 nós de armazenamento na rede Giga e comparamos o consumo destes recursos com 200 e 300 clientes que assistem a um vídeo de 150 segundos de duração. No gráfico 4.29, temos os resultados da taxa de ocupação do processador quando simulamos 200 clientes. Os resultados da taxa de ocupação do processador por 300 clientes estão no gráfico 4.30. Podemos observar que o aumento do número de clientes não onerou a utilização da CPU e o pico de ocupação é de apenas 8%. O consumo de memória RAM pelo servidor quando temos 200 e 300 clientes estão nos gráficos 4.31 e 4.32, respectivamente. O consumo de RAM, com 200 ou 300 clientes, praticamente não se altera e é de apenas 5%, aproximadamente. O aumento de 50% do número de clientes no servidor causou um aumento desprezível na utilização do processador e de memória RAM pelo servidor e é muito baixo para estas cargas. Assim, podemos concluir que há muito espaço para escalar o servidor de forma a atender um número de clientes considerável.

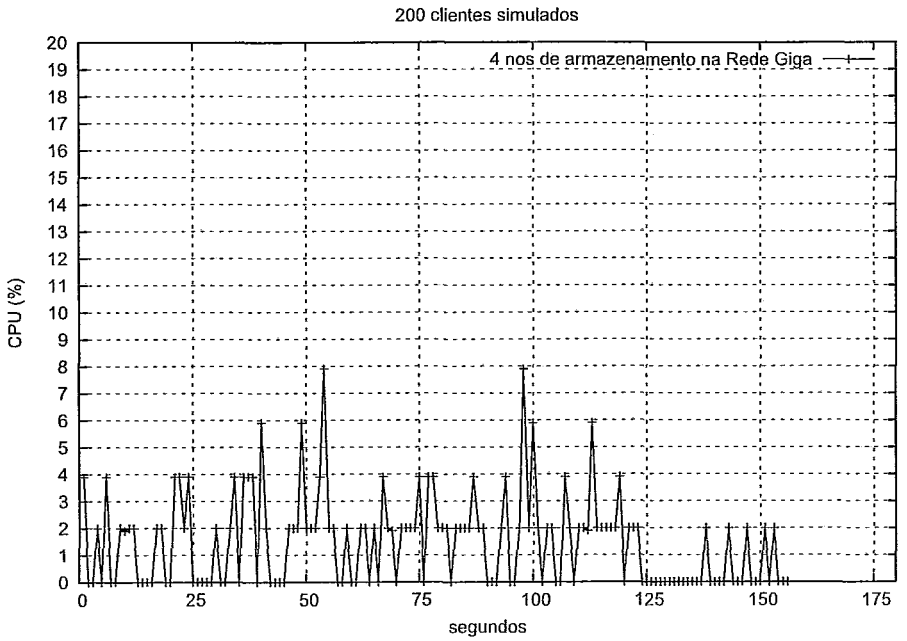

Figura 4.29: Gráfico da taxa de ocupação do processador x tempo de simulação, com o RIO com 4 nós na rede Giga e 200 clientes.

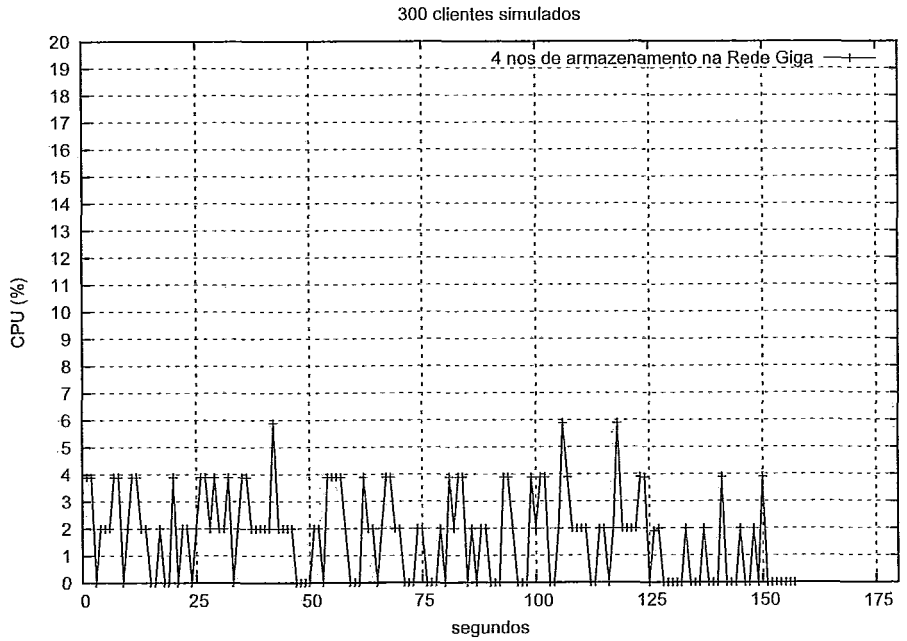

Figura 4.30: Gráfico da taxa de ocupação do processador x tempo de simulação, com o RIO com 4 nós na rede Giga e 300 clientes.

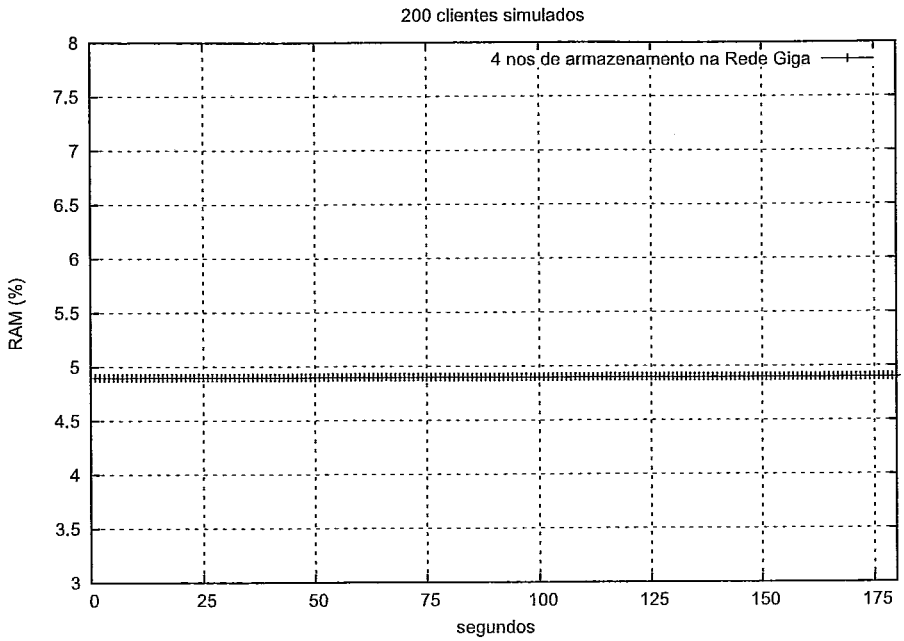

Figura 4.31: Gráfico da taxa de ocupação da memória RAM x tempo de simulação, com o RIO com 4 nós na rede Giga e 200 clientes.

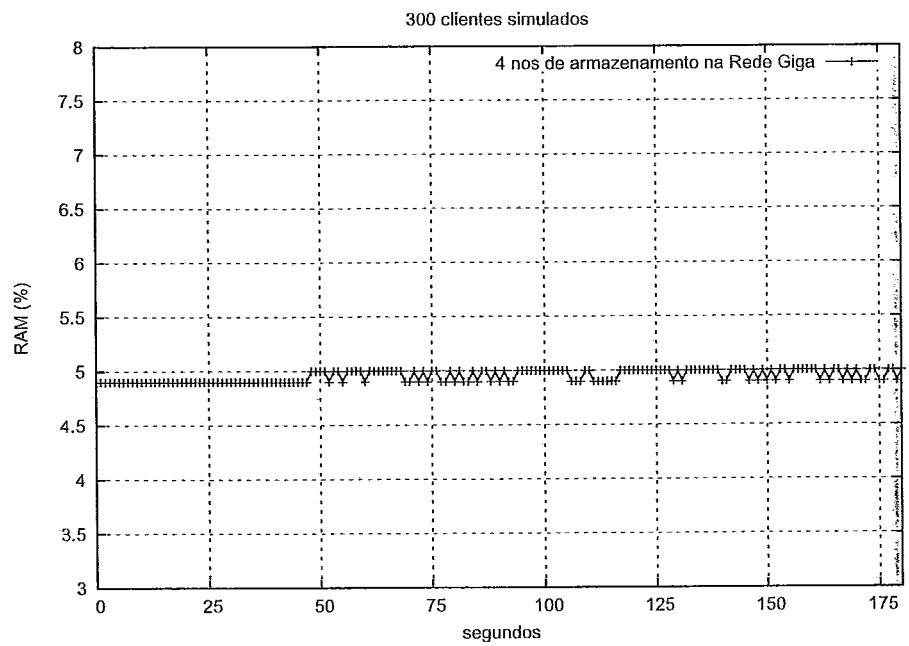

Figura 4.32: Gráfico da taxa de ocupação da memória RAM x tempo de simulação, com o RIO com 4 nós na rede Giga e 300 clientes.

Capítulo 5

Conclusão e Trabalhos Futuros

EDUCAÇÃO a distância será a forma de democratização do acesso à informação e da igualdade de oportunidades no futuro. E a Internet já é o veículo pelo qual esta revolução silenciosa ocorrerá. Portanto, todo e qualquer esforço por melhorar ou tentar compreender os artefatos computacionais pelos quais isto se dará é da mais extrema necessidade. Muito já se tem feito na literatura para se compreender e incrementar o desempenho de uma ferramenta importante para esse fim: o servidor multimídia. Entretanto, pouco se estudou quando sua aplicação é para ensino a distância e menos ainda foi feito quando se trata de experimentos com sistemas reais. Este trabalho se encaixa neste último ponto, com o objetivo de estudar o comportamento de um servidor multimídia através de experimentos reais.

Assim, foi realizado um apanhado na literatura das diversas formas propostas de se construir um servidor multimídia escalável. As propostas foram organizadas de forma a se obter uma visão geral das vantagens e desvantagens das diversas arquiteturas possíveis. Concluiu-se, então, que um servidor DVoD é uma das arquiteturas mais apropriadas para um sistema de educação a distância. Utilizamos o servidor RIO em um ambiente real para medir o seu desempenho em face a diversas configurações diferentes a fim de estudar os muitos fatores de impacto existentes que muitas vezes são negligenciados em modelos teóricos. Um cenário específico foi determinado, construiu-se toda uma infraestrutura tanto física quanto computacional e métricas de desempenho foram escolhidas. O objetivo maior dos experimentos foi verificar o comportamento de um servidor multimídia em uma rede com cam-

inhos heterogêneos e, assim, fazer uma relação deste problema com o problema de diversidade de caminhos. Para cada experimento proposto, os resultados obtidos foram estudados e o gargalo do sistema foi determinado. Possíveis soluções para a ampliação deste gargalo foram sugeridas e novos experimentos realizados.

Por fim, concluiu-se que caminhos heterogêneos influenciam na qualidade de serviço do servidor quando consideramos taxas de goodput próximas de 100%. Também verificamos que as perdas se distribuem entre fragmentos atrasados e perdidos, sem predominância de nenhum na maior parte dos casos. Portanto, é possível auferir-se ganhos consideráveis mesmo com canais com taxas de perda da ordem de 4 a 5% tal qual o caminho entre a UFRJ e a UFMG estudado.

Também foi averiguado que a replicação somente é util para a qualidade de serviço do sistema quando temos uma baixa utilização da capacidade de suas interfaces de rede: com a tecnologia atual, estas são os principais pontos de gargalo de um sistema multimídia. Portanto, a replicação somente trás o aumento da confiabilidade ao sistema. Também mostramos que o servidor RIO tem muito espaço para escalar mais clientes do que o experimentado neste trabalho, pois averiguamos que a taxa de ocupação do processador e da memória RAM pelo servidor tem uma alteração marginal quando o número de clientes aumenta. Assim, poderíamos aumentar o número de clientes no sistema com a adição de novos nós de armazenamento até o limite da capacidade de comutação do summit.

5.1 Trabalhos Futuros

É interessante a alteração do mecanismo de balanceamento de carga do servidor para que contemplasse mais variáveis do que apenas o tamanho das filas em disco. Por exemplo, ele poderia levar em consideração a taxa de perda do canal e o tempo de propagação daquele nó de armazenamento até o cliente: somente com estas mudanças a utilização de replicação poderia tornar-se mais recompensadora para a qualidade de serviço do sistema. Com este fim, poderia ser possível utilizar-se de [54], que contém um algoritmo de predição da taxa de perda de curto prazo com a utilização de uma Cadeia de Markov Oculta. Compondo um algoritmo de

balanceamento de carga com este algoritmo, poderíamos otimizar o roteamento de pedidos para o nó com a maior probabilidade de receber o pedido e com menor fila, o que é relacionado com o problema de diversidade de caminhos.

Também é válido sugerir a repetição dos experimentos deste trabalho com a utilização de um modelo de comportamento de usuário para a geração dos pedidos feitos ao servidor. Este modelo poderia ser conforme o estudado em [63]. Isto porque, neste trabalho, supomos inicialmente ter usuários que visualizariam os vídeos do início ao fim, como um filme, para facilitar a execução dos experimentos, pois existem modelos na literatura para geração de carga sintética com interatividade, mas não existia nenhum modelo implementado para ser usado ao início dos experimentos. Entretanto, este comportamento linear não é o comportamento de um usuário em face a uma aplicação interativa como ensino a distância. Portanto, a carga gerada por estes usuários seria diferente da carga gerada neste trabalho, o que poderia alterar os resultados aqui obtidos e, portanto, é uma hipótese válida para ser investigada.

Um trabalho futuro que pode ser implementado é a mudança da política de armazenamento do servidor RIO. Com o objetivo de se atender a grupos locais de clientes, o servidor RIO poderia ser adaptado para funcionar como uma coleção de mini-servidores e um servidor central, onde parte do conteúdo ficaria armazenado de acordo com a demanda local. Quando algum cliente fizesse uma requisição que não se encontrasse em seu mini-servidor, este faria uma requisição ao servidor central e, então, copiaria o que não foi encontrado para seu armazenamento local.

Em andamento temos a criação de modelos matemáticos que prevejam os resultados experimentais obtidos neste trabalho. Estes modelos poderiam ser úteis na criação de novos algoritmos de balanceamento de carga conforme o descrito acima, por exemplo.

Referências Bibliográficas

[1] Tran, D., Hua, K., and Do, T. A peer-to-peer architecture for media streaming. In *IEEE Journal on Selected Areas in Communications* (2003), vol. 22, pp. 121–133.

[2] Velos, E., Almeida, V., Meira, W., Bestavros, A., and Jin, S. A hierarchical characterization of a live streaming media workload. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment* (2002), pp. 117–130.

[3] Berson, S., Golubchik, L., and Muntz, R. Fault tolerant design of multimedia servers. In *Proc. SIGMOD* (1995), pp. 364–375.

[4] Birk, Y. Random raids with selective exploitation of redundancy for high performance video servers. In *Proc. NOSSDAV* (May 1997), pp. 13–23.

[5] de Souza e Silva, E., Leao, R., Ribeiro-Neto, B., and Campos, S. Performance issues of multimedia applications, performance evaluation of complex systems: Techniques and tools. *Springer 2459Lec* (2002), 374–405.

[6] Hefeeda, M., Bhargava, B., and Yau, D. A hybrid architecture for cost-effective on-demand media streaming. *Computer Networks: The International Journal of Computer and Telecommunications Networking 44*, 3 (February 2004), 353–382.

[7] Barnett, S., and Anido, G. A cost comparision of distributed and centralized approaches to video-on-demand. *IEEE Journal on Selected Areas in Communications 14* (August 1996), 1173–1183.

[8] CORES, F., RIPOLLI, A., QAZZAZ, B., SUPPI, R., YANG, X., HERNANDEZ, P., AND LUQUE, E. Exploiting traffic balancing and multicast efficiency in distributed video-on-demand architectures. In *Proc. 9th International Euro-Par Conference* (Klagenfurt, Austria, August 2003), pp. 859–869.

[9] WANG, B., SEN, S., ADLER, M., AND TOWSLEY, D. Optimal proxy cache allocation for efficient streaming media distribution. In *Proc. INFOCOM* (2002), vol. 3, pp. 1726–1735.

[10] WANG, Y., ZHANG, Z., DU, D., AND SU, D. A network-conscious approach to end-to-end video delivery over wide area networks using proxy servers. In *Proc. INFOCOM* (San Francisco, CA, April 1998), vol. 2, pp. 660–667.

[11] MIAO, Z., AND ORTEGA, A. Proxy caching for efficient video services over the internet. In *Proc. of 9th International Packet Video Workshop (PV'99)* (New York, USA, April 1999).

[12] CHAN, S., AND TOBAGI, F. Distributed servers architecture for networked video services. *IEEE/ACM Transactions on Networking 9*, 2 (April 2001), 125–136.

[13] Akamai. *http://www.akamai.com.*

[14] Digital Island. *http://www.digitalisland.com.*

[15] JOHNSON, K., CARR, J., DAY, M., AND KAASHOEK, M. The measured performance of content distribuition networks. *Computer Communications 24*, 2 (2001), 18–27.

[16] GADDE, S., CHASE, J., AND RABINOVICH, M. Web caching and content distribution: A view from the interior. *Proc. of the 5th International Web Caching and Content Delivery Workshop 24*, 2 (May 2000), 222–231.

[17] DO, T., HUA, K., AND TANTAOUI, M. P2VOD: Providing fault tolerant video-on-demand streaming in peer-to-peer environment. In *IEEE Communications Society* (2004), vol. 3, pp. 1467–1472.

[18] GUO, Y., SUH, K., KUROSE, J., AND TOWSLEY, D. P2CAST: peer-to-peer patching scheme for VOD service. In *Proc. of the 12th international conference on World Wide Web (WWW'03)* (New York, NY, USA, 2003), ACM Press, pp. 301–309.

[19] PADMANABHAN, V., WANG, H., AND CHOU, P. Distributing streaming media content using cooperative networking. In *Proc. NOSSDAV* (2002), pp. 177–186.

[20] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl. 15*, 3 (2001), 200–222.

[21] ANDROUTSELLIS-THEOTOKIS, S., AND SPINELLIS, D. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys 36*, 4 (2004), 335–371.

[22] BEIRIGER, J., JOHNSON, W., BIVENS, H., HUMPHREYS, S., AND RHEA, R. Constructing the ASCI Grid. In *Proc. 9th IEEE Symposium on High Performance Distributed Computing* (2000).

[23] JOHNSTON, W., GANNON, D., AND NITZBERG, B. Grids as production comp uting environments: The engineering aspects of nasa's information power grid. In *Proc. 8th IEEE Symposium on High Performance Distributed Computing* (1999), p. 34.

[24] BRUNEO, D., GUARNERA, M., ZAIA, A., AND PULIAFITO, A. A grid-based architecture for multimedia services management. In *Proc. of the 2003 Annual Crossgrid Project Workshop and 1st European Across* (2003).

[25] FOSTER, I., AND IAMNITCHI, A. On death, taxes, and the convergence of peer-to-peer and grid computing. *Lecture Notes of Computer Science: Peer-to-Peer Systems II: Second International Workshop, IPTPS 2735* (February 2003), 118–128.

[26] ZHANG, L., AND LO, K. A peer-to-peer architecture for on-demand video streaming on internet. In *Proc. of the 2004 International Conference on Communications, Circuits and Systems(ICCCAS'04)* (2004), pp. 525–528.

[27] LEE, J., AND LEUNG, R. Study of a server-less architecture for video-on-demand applications. In *Proc. of the IEEE International Conference on Multimedia and Expo (ICME '02)* (2002), pp. 233–236.

[28] KALAPRIYA, K., AND NANDY, S. Throughput driven, highly available streaming stored playback video service over a peer-to-peer network. In *Proc. of the 19th IEEE International Conference on Advanced Information Networking and Applications (AINA'05)* (March 2005), vol. 2, pp. 229–234.

[29] WICKER, S. B. *Error Control Systems for Digital Communication and Storage.* Prentice-Hall.

[30] NETTO, B. Patching interativo: um novo método de compartilhamento de recursos para transmissão de vídeo com alta interatividade. Tese de Mestrado, Federal University of Rio de Janeiro, February 2004.

[31] DA SILVA RODRIGUES, C. K. *Mecanismos de Compartilhamento de Recursos para Aplicações de Mídia Contínua na Internet.* Tese de Doutorado, Federal University of Rio de Janeiro, 2006.

[32] Gnutella. *http://www.gnutella.com.*

[33] GANESH, A., KERMARREC, A., AND MASSOULIÉ, L. Peer-to-peer membership management for gossip-based protocols. In *Proc. of the IEEE Transactions on Computers* (February 2003), pp. 139–149.

[34] ADAR, E., AND HUBERMAN, B. Free riding on gnutella. *First Monday 5*, 10 (2000).

[35] MARKATOS, E. Tracing a large scale peer-to-peer system: An hour in the life of Gnutella. *2nd IEEE/ACM Int. Symp. on Cluster Computing and the Grid* (2002).

[36] POUWELSE, J., GARBACKI, P., EPEMA, D., AND SIPS, H. A measurement study of the bittorrent peer-to-peer file-sharing system. *Tech. Rep. PDS-2004-007 Delft University of Technology* (2004).

[37] KALOGERAKI, V., DELIS, A., AND GUNOPULOS, D. Peer-to-peer architectures for scalable, efficient and reliable media services. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium* (April 2003), p. 10 pp.

[38] KELLER, R., CHOI, S., DASEN, M., DECASPER, D., FANKHAUSER, G., AND PLATTNER, B. An active router architecture for multicast video distribution. In *Proc. INFOCOM* (December 2000), vol. 3, pp. 1137–1146.

[39] BURCHARD, L., AND LULLING, R. An architecture for a scalable video-on-demand server network with quality-of-service guarantees. In *7th International Workshop on Interactive Distributed Multimedia Systems and Telecomunications Services (IDMS)* (2000), pp. 132–143.

[40] VENKATASUBRAMANIAN, N., AND RAMANATHAN, S. Load management in distributed video servers. In *17th IEEE International Conference on Distributed Computer Systems (ICDS)* (1997), p. 528.

[41] LEE, J. Y., AND WONG, P. Performance analysis of a pull-based parallel video server. *IEEE Transactions on Parallel and Distributed Systems 11*, 12 (December 2000), 1217–1231.

[42] CHOU, C., GOLUBCHIK, L., LUI, J., AND CHUNG, I. Design of scalable continuos media servers. *Multimedia Tools And Applications 17*, 2-3 (July 2002), 181–212.

[43] GHANDEHARIZADEH, S., AND MUNTZ, R. Design and implementation of scalable continuos media servers. *Parallel Computing 24*, 1 (January 1997), 91–122.

[44] KORST, J. Random duplicated assignment: An alternative to striping in video servers. In *Proc. of ACM Multimedia* (1997), pp. 219–226.

[45] FABBROCINO, F., SANTOS, J., AND MUNTZ, R. An implicitly scalable, fully interactive multimedia storage server. In *Proc. of the 2nd International Workshop on Distributed Interactive Simulation and Real Time Applications* (1998), p. 92.

[46] SANTOS, J., AND MUNTZ, R. Performance analysis of the rio multimedia storage system with heterogeneous disk configurations. In *Proc. of the sixth ACM international conference on Multimedia* (1998), pp. 303–308.

[47] MUNTZ, R., SANTOS, J., AND FABBROCINO, F. Design of a fault tolerant real-time storage system for multimedia applications. In *IEEE Internation Computer Performance and Dependability Symposium (IPDS)* (1998), p. 174.

[48] SANTOS, J., MUNTZ, R., AND RIBEIRO-NETO, B. Comparing random data allocation and data striping in multimedia servers. In *Proc. ACM Sigmetrics* (2000), pp. 44–55.

[49] MITZENMACHER, M. *The power of two choices in randomized load balancing.* Tese de Doutorado, University of California at Berkeley, Computer Science Department, 1996.

[50] MUNTZ, R., SANTOS, J., AND BERSON, S. A parallel disk storage system for realtime multimedia applications. *International Journal of Intelligent Sciences, Special Issue on Multimedia Computing Systems 13*, 12 (December 1998), 1137–1174.

[51] LIE, P., LUI, J., AND GOLUBCHIK, L. Threshold-based dynamic replication in large-scale video-on-demand systems. *Multimedia Tools And Applications 11*, 1 (May 2000), 35–62.

[52] TAO, S., XU, K., XU, Y., FEI, T., GAO, L., GUÉRIN, R., KUROSE, J., TOWSLEY, D., AND ZHANG, Z. Exploring the performance benefits of end-to-end patch switching. In *Proc. of the 12th IEEE International Conference on Network Protocols (ICNP)* (2004).

[53] ABDOUNI, B., CHENG, W., CHOW, A., GOLUBCHIK, L., AND LUI, J. Picture-perfect streaming over the internet: Is there hope? *IEEE Communications Magazine 42*, 8 (August 2004), 72–79.

[54] FILHO, F. J. S. Previsão de estatísticas de perdas de pacotes usando modelos de markov ocultos. Tese de Mestrado, Federal University of Rio de Janeiro, May 2006.

[55] RIBEIRO, B., DE SOUZA E SILVA, E., AND TOWSLEY, D. Estudo da eficácia do método de diversidade de caminhos para aplicações multimídia. In *Anais do XXII Simpósio Brasileiro de Redes de Computadores* (May 2004), vol. 1, pp. 307–320.

[56] DE QUEVEDO CARDOZO, A. Mecanismos para garantir qualidade de serviço de aplicações de vídeos sob demanda. Tese de Mestrado, Federal University of Rio de Janeiro, 2002.

[57] Gereoffy, A. MPlayer - Movie Player for Linux. *http://www.mplayerhq.hu/homepage/*.

[58] PROJETO GIGA, subprojeto Distribuição de Vídeo em Larga Escala sobre Redes Giga, com Aplicações a Educação (DIVERGE). Instituições Participantes: COPPE/UFRJ, UFF, UFMG, FIOCRUZ, PUC-RJ e UMass.

[59] ROSS, S. M. *A Course in Simulation.* Prentice-Hall, 1990.

[60] MARTINELLO, M., AND DE SOUZA E SILVA, E. A testbed for network performance evaluation and its application to connection admission control algorithms. *J. Braz. Comp. Soc. 7*, 2 (2001), 39–51.

[61] ROCHA, A., LEÃO, R., AND DE SOUZA E SILVA, E. Metodologia para estimar o atraso em um sentido e experimentos na internet. In *Anais do XXII Simpósio Brasileiro de Redes de Computadores* (May 2004), vol. 1, pp. 589–602.

[62] KESHAV, S. A control-theoretic approach to flow control. In *ACM SIGCOMM* (1991).

[63] TOMIMURA, D. Caracterização e modelagem do comportamento de usuários acessando um vídeo de ensino a distância. Tese de Mestrado, Federal University of Rio de Janeiro, March 2006.

Apêndice A

Detalhes das Implementações Realizadas

NESTE apêndice iremos descrever com mais detalhes algumas das melhorias no código implementadas no servidor RIO e os algoritmos modificados, os quais ainda não foram descritos anteriormente, no corpo deste trabalho.

A.1 Algoritmo de Cópia com Redundância

O RIO faz a cópia com redundância que foi codificada no objeto DiskManager (/server/DiskMgr.cpp), através do método cuja assinatura é int *DiskMgr* :: *AllocMult*(int numReplicas, RioDiskBlock *rep). Este método recebe como parâmetros o número de réplicas a serem feitas, na variável numReplicas e qual bloco deve ser gravado, no ponteiro *rep. O algoritmo, simplificadamente, tem estes passos:

1. calculo o número de nós de armazenamento e armazeno na variável numStorages;

2. certifico-me de que o número de réplicas da assinatura do método não é maior do que o número de nós de armazenamento. Se for, reduzo-a para o número de nós de armazenamento.

3. inicializo algumas variáveis, entre elas os inteiros que irão armazenar quais discos (discos_disponiveis_rep) e nós de armazenamento (storages_disp) ainda

tenho disponíveis para alocar uma das réplicas do bloco a ser gravado;

4. faço um laço com o número de vezes que devo copiar o bloco da assinatura do método;

5. o primeiro passo do laço é me certificar que existem discos disponíveis para se alocar uma cópia do bloco;

6. depois entro em um laço onde sorteio um número pseudo-aleatório, uniformemente distribuído. Só saio do laço quando este número corresponde a um disco que não possui uma cópia deste bloco nem esta localizado em um nó de armazenamento que contém uma cópia deste bloco.

7. em posse de um disco sorteado nas condições acima, sorteio uma posição qualquer dentro dele e verifico se a mesma está livre. Caso não esteja, retorno ao passo 5;

8. retorno ao passo 4 até ter alocado todas as cópias;

Algumas considerações sobre como algumas variáveis são calculadas:

- a variável numStorages é calculada como sendo o número total de discos do sistema pelo número de discos por nó de armazenamento;

- a variável discos_disponiveis_rep é uma potência de 2 elevado ao número de discos do sistema. Isto porque cada bit desta variável indica a disponibilidade de um disco para se alocar uma réplica de um bloco, lembrando que um disco só pode ter, no máximo, uma cópia de um mesmo bloco. Por isso que, inicialmente, esta variável recebe 2 elevado ao número de discos do sistema: todos os bits 1 desta variévei indicam que todos os discos estão disponíveis para serem selecionados. Então, a cada disco selecionado, eu transformo o bit correspondente do disco selecionado para zero, a fim de indicar que aquele disco já possui uma cópia do bloco que está sendo gravado. Portanto, para saber se posso usar este disco, eu faço uma comparação de bit a bit com o operador &.

Por exemplo, no caso de 4 storages e 3 réplicas, discos_disponiveis_rep é inicializado com 15 (2^4 - 1, todos os bits setados). Suponhamos que, na primeira cópia, eu seleciono randomicamente o terceiro disco. Faço a = 2^3 para descobrir qual bit tenho que tornar zero e faço isso com discos_disponiveis_rep = discos_disponiveis_rep - a.

- a variável storages_disp garante que só terei uma cópia por nó de armazenamento. Isto é feito não permitindo que sejam escolhidos discos no mesmo nó de armazenamento. Uma vez que eu sei que cada nó de armazenamento tem limite máximo de 4 discos, faço a mesma estratégia acima de configuração dos bits para indicar quais nós de armazenamento ainda tenho disponíveis. Depois disso, para saber em qual nó de armazenamento esta o disco sorteado, eu divido o número do disco por 4 e arredondo para o inteiro acima. Tendo posse do número do nó de armazenamento sorteado, aplico a mesma técnica acima para verificar se o nó de armazenamento está disponível e gravar zero em sua posição para indicar seu uso.

A.2 Melhorias no código

Nesta seção iremos listar algumas das melhorias realizadas no código do RIO. Iremos agrupá-las por arquivo alterado, em conjunto com uma breve descrição.

server/DiskMgr.cpp:

- Correção do *segmentation fault* quando se copiava um arquivo após o nó de armazenamento estar cheio e abortar a cópia anterior, que acontecia durante a cópia de diretórios, por exemplo;
- Adaptações para o código compilar no Mandrake 10.1;
- Retirado o valor 2 fixo como o número máximo de replicações. Agora o servidor lê este valor do system.cfg;

server/DiskMgr.h:

- Alterado o nome da variável "num"para "numReplicas"para maior clareza do codigo;

- Removido o valor 2 fixo como máximo número de replicações. Agora o servidor lê este valor dos system.cfg;

server/SessionManager.cpp:

- Tornei legível para depuração o endereco IP de uma mensagem de erro de conexão com o nó de armazenamento;

server/SystemManager.cpp:

- Removido o valor 2 fixo como máximo número de replicações. Agora o servidor lê este valor dos system.cfg;

Makefile.inc, libraries/RioNeti.cpp:

- Correção de um problema no envio dos fragmentos de um bloco, onde o envio era bloqueado no meio quando ocoria um problema com o socket, no caso do erro EAGAIN. Com isso os problemas de perda de fragmentos ainda dentro do nó de armazenamento estão resolvidos.

libraries/RioNeti.cpp, vsi/vsisocket.cpp, clients/riommclient/src/RioMMIntSimul.cpp:

- Mudanca no socket TCP, o qual agora passa parâmetro MSG_WAITALL

clients/riommclient/src/RioMMVideo.cpp:

- Inserção de comentário no método *WaitPrefetch* justificando o seu comportamento

etc/system.cfg:

- Comentário explicativo do uso da variável MaxDisks

Estes próximos arquivos são uma nova ferramenta criada para extrair dos nós de armazenamento qualquer vídeo gravado no servidor RIO. Funcionam com qualquer replicação. O seu uso é explicado ao se chamar o dumpRIOobj sem parâmetros. Algumas de suas características seguem abaixo:

tools/riofs/discos.conf,
tools/riofs/Attic/dumpRIOobj.c,
tools/riofs/Attic/dumpRIOobj.cpp:

- Lê configuração de onde estão os discos do arquivo discos.conf;
- Funciona com qualquer redundancia, mas apenas com a configuracao de 1 disco por storage;
- Imprime tabela de metadados no final da extração, para um objeto com até 5 réplicas;
- Gera arquivos de saída no formato RioObjX.mpg, onde X é o número da réplica;

storage/routerinterface.cpp:

- Nó de armazenamento agora informa, em sua inicialização, o IP do computador em que está sendo executado e em qual porta está configurado;

A.3 Formato do log gerado pelo cliente

Nesta seção, descreveremos o formato do log gerado pelo cliente. Abaixo segue um exemplo de um log:

From salinas.land.ufrj.br ==========

Quantidade de blocos 259

1144634226.122899 Pedido: 0

1144634226.123073 Pedido: 1

1144634226.123219 Pedido: 2

1144634226.123338 Pedido: 3

1144634226.123466 Pedido: 4

[RioMMEmul] Open() - valor retornado: 240

[RioMMEmul] Retornando da acao 240

RioMMClient - testando wait_time: 240

1144634226.354065 Recepção: 0

1144634226.356970 Execução: 0

1144634226.394118 Pedido: 5

1144634226.800434 Recepção: 1

A.3 Formato do log gerado pelo cliente

1144634226.803881 Execução: 1

1144634227.699501 Recepção: 3

1144634227.809096 Pedido: 6

Esperando bloco 2

Perdido: 2

1144634228.814891 Pedido: 7

1144634228.815045 Recepção: 7

1144634228.818906 Execução: 3

1144634229.685903 Pedido: 8

Esperando bloco 4

Recuperado : 4 tamanho: 125216 Bytes

[RioNeti] Fragmento 0 da requisição 3 atrasado!

[RioNeti] Fragmento 1 da requisição 3 atrasado!

[RioNeti] Fragmento 2 da requisição 3 atrasado!

[RioNeti] Fragmento 3 da requisição 3 atrasado!

<...>

1144634434.705919 Execução: 257

1144634435.461253 Execução: 258

RioMMClient - Finalizando emulador

Blocos completamente recebidos: 252

Blocos parcialmente recebidos: 0

Blocos perdidos: 7

Bytes exibidos: 33030144

Goodness: 97.297297

Blocos Completamente Atrasados: 5

Blocos Parcialmente Atrasados: 0

Blocos Realmente Completamente Perdidos: 2

Blocos Realmente Parcialmente Perdidos: 0

Fragmentos Atrasados: 450

Fragmentos Perdidos: 180

Goodput (%): 97.2973

A.3 Formato do log gerado pelo cliente

Segue agora uma descrição de seu significado:

- Na primeira linha temos o caminho do computador onde o cliente está sendo executado. No exemplo acima, a execução se realizou na máquina *salinas.land.ufrj.br*;
- Logo após é impresso o número de blocos do vídeo requisitado, informado pelo servidor. No exemplo, esta quantidade é de 259;
- Todas ações do cliente são logadas. As ações de pedir um bloco, recebê-lo por inteiro e executá-lo são logadas com o tempo em que isto ocorreu. No exemplo, no segundo 1144634226.122899, que são quantos segundos passaram desde 1 de janeiro de 1970, o cliente pediu o bloco inicial 0, o qual foi recebido no instante 1144634226.354065, ou seja, 231.166 milisegundos depois, e executado no instante 1144634226.356970, 2.905 milisegundos após o recebimento;
- As ações de interatividade do cliente são logadas com o marcador [RioMMEmul]. No exemplo, vemos que o cliente executou uma abertura de sessão;
- Quando um bloco chega atrasado, a linha do log vem com um marcador [RioNeti], onde indica qual o fragmento de qual bloco que chegou atrasado, ou seja, após o momento de sua execução;
- Quando um bloco não chega até ser o próximo a ser executado, é impresso uma linha escrita "Esperando bloco X", onde X é o número do próximo bloco a ser executado pelo cliente e que ainda não chegou completamente;
- Caso chegue no momento de sua execução e o bloco não tenha chegado completamente, o cliente executa os fragmentos que chegaram a tempo e imprime: Recuperado : X tamanho: Y Bytes, onde X é o número do bloco recuperado e Y é o número de bytes recuperados do bloco. Caso não tenha chegado nenhum fragmento, é impresso Perdido: Z, onde Z é o número do bloco perdido;
- Nas últimas linhas dos arquivos, são impressos estatísticas da sessão. O conceito mais importante é a linha com o nome *Goodness*, que é o *Goodput* calculado pelo cliente. Entretanto, como foi verificado que havia pequenas discrepâncias entre o que era impresso e este cálculo, montamos um outro script

em AWK que varre este log e recalcula o *Goodput* com base no que foi impresso inicialmente. Assim, este novo cálculo é impresso na última linha do arquivo, com o nome *Goodput* e o seu porcentual nesta sessão;