Um Sistema de Arquivos Distribuído Peer-to-Peer Tolerante a Faltas usando Arquivos Transparentes

Marcelo Madruga, Sérgio Loest, Carlos Maziero, Altair Santin

[1]Programa de Pós-Graduação em Informática
Pontificia Universidade Católica do Paraná
Curitiba – PR – Brasil

{mcmadruga,sloest,maziero,santin}@ppgia.pucpr.br

Abstract. *The peer-to-peer computing model motivates the creation of new distributed file systems. This paper presents a peer-to-peer distributed file system which uses "transparent files" to improve its fault tolerance and file availability, without imposing much impact on the local peers' resources. Files are kept as transparent replicas, using the free disk space in each peer. However, a transparent file may be invalidated if the local filesystem needs the space it uses. When such a replica is invalidated, the peers should cooperate to restore it back. The proposed architecture was implemented and tested; experiments showed its feasibility and that its costs are proportional to the size of files being replicated. Also, the occurrence of multiple simultaneous replica invalidations do not impose a heavy burden on the system.*

Resumo. *O modelo* peer-to-peer *e a disponibilidade de largura de banda têm viabilizado a criação de novos sistemas de arquivos distribuídos. Este artigo apresenta um sistema de arquivos distribuído* peer-to-peer *que usa "arquivos transparentes" para melhorar a tolerância a faltas e a disponibilidade dos dados, sem impor um impacto significativo nos recursos locais de cada nó. Arquivos são salvos como réplicas transparentes no espaço livre em disco de cada nó. Entretanto, um arquivo transparente pode ser invalidado, se o sistema de arquivos local precisar do espaço ocupado por ele. Quando uma réplica é invalidada, os nós cooperam entre si para restaurá-la. Experimentos mostraram a aplicabilidade da proposta, e que seu custo é proporcional ao tamanho dos arquivos replicados. Além disso, invalidações múltiplas e simultâneas não causam impacto significativo no sistema.*

1. Introdução

O desenvolvimento do modelo *peer-to-peer (p2p)* e o crescimento da disponibilidade de largura de banda entre os computadores ligados à Internet [Payne and Woolnough 2004] estão fomentando a criação de sistemas de arquivos e armazenamento distribuídos, pois o modelo *p2p* possui algumas características interessantes, como auto-organização, escalabilidade e baixo custo de operação [Mauthe and Hutchison 2003, Milojicic et al. 2002]. Vários sistemas de arquivos e armazenamento que usam o modelo *p2p* foram propostos recentementes, como o FARSITE [Adya et al. 2002], OceanStore [Kubiatowicz et al. 2000], PAST [Druschel and Rowstron 2001b], Ivy [Muthitacharoen et al. 2002] e CFS [Dabek et al. 2001].

Tradicionalmente, arquivos pertencentes a aplicações distribuídas e arquivos pertencentes a aplicações locais são tratados da mesma forma pelo sistema de arquivos local, que armazena os dados no disco rígido. Assim, os dois tipos de arquivos contribuem igualmente para o consumo de espaço em disco. No entanto, essa falta de discriminação traz à tona alguns problemas e restrições para a ampla adoção de sistemas baseados no modelo *p2p*. Alguns sistemas *p2p* são vistos meramente como aplicações contributivas (*contributory applications*), nas quais usuários doam seu espaço livre em disco para outros usuários. Para esses sistemas, o comportamento dos usuários em relação à doação de espaço livre em disco varia muito: alguns têm receio de compartilhar ou doar muito espaço livre em disco, pois podem vir a precisar desse espaço para outros fins; já outros desejam doar o máximo possível de espaço livre em disco, porque sabem que em troca poderão acessar mais conteúdo [Golle et al. 2001].

Ao longo dos últimos anos, foram apresentadas várias idéias para estimular o compartilhamento de recursos locais [wan Ngan et al. 2003] [Leonard et al. 2002]. O conceito de "arquivos transparentes" introduzido em [Cipar et al. 2007] pode ser uma dessas soluções. O *Transparent File System* (TFS) é um sistema de arquivos local, capaz de usar o espaço livre nos discos locais de cada máquina para criar arquivos transparentes. Esses arquivos transparentes podem ser usados para armazenar dados na área livre do disco, já que blocos do disco ocupados por arquivos transparentes podem ter seu conteúdo sobrescrito e não são contabilizados para o cálculo do espaço ocupado.

Neste artigo, é proposto um sistema de arquivos distribuído baseado no modelo *p2p* que oferece alta disponibilidade dos dados e tolerância a faltas através do uso de arquivos transparentes. Todavia, arquivos transparentes impõem um novo desafio na implementação do sistema, porque eles podem ser removidos do disco local a qualquer momento e de forma imprevisível. O foco do projeto proposto é a capacidade de tratar essas remoções de forma eficiente, fazendo com que o sistema distribuído se ajuste automaticamente para manter constante o número de réplicas disponíveis de cada arquivo. Este artigo está assim organizado: a seção 2 apresenta sucintamente alguns sistemas de arquivos distribuídos que adotam o modelo *p2p*. A seção 3 descreve o *Transparent File System* e o conceito de arquivos transparentes. Já a seção 4 discorre sobre o projeto do sistema proposto e descreve os algoritmos usados pelo sistema para tratar de forma eficiente as exclusões dos arquivos transparentes. A seção 5 descreve a implementação do protótipo e apresenta os resultados dos experimentos. Por fim, a seção 6 discute os trabalhos relacionados e a seção 7 delineia conclusões e discorre sobre trabalhos futuros.

2. Sistemas de Arquivos Distribuídos baseados no modelo p2p

Sistemas *p2p* apresentam algumas vantagens em relação aos sistemas cliente/servidor, como maior flexibilidade, maior escalabilidade e menores custos. Inicialmente, sistemas *p2p* foram criados para compartilhar arquivos anonimamente na Internet, mas outras áreas têm se beneficiado das propriedades dos ambientes *p2p*. Existem sistemas *p2p* para distribuição de áudio e vídeo, trocas de mensagens instantâneas, e sistemas de arquivos e armazenamento distribuído [Milojicic et al. 2002, Mauthe and Hutchison 2003, Androutsellis-Theotokis and Spinellis 2004, Hasan et al. 2005]. Em sistemas *p2p*, os participantes (*peers*) atuam como clientes e servidores do serviço oferecido, e executam as mesmas tarefas. *Peers* interagem entre si para anunciar recursos disponíveis, localizar recursos e objetos e trocar mensagens. Esses sistemas são formados a partir de um subs-

trato que mantém uma tabela de *hash* distribuída (*Distributed Hash Table* ou DHT), que permite armazenar pares {*chave,valor*}. Os sistemas *Chord* [Stoica et al. 2001], *Pastry* [Rowstron and Druschel 2001] e *Tapestry* [Zhao et al. 2004] são exemplos de substratos *p2p* e DHT.

O *Cooperative File System* (CFS) [Dabek et al. 2001] é um sistema de arquivos distribuído somente de leitura (*read-only*), no qual todos os usuários podem ler os arquivos, mas somente os seus donos podem modificá-los. O CFS usa *Chord* como substrato e sua DHT para armazenar e buscar os meta-dados dos arquivos. Todos os *peers* usam o mesmo algoritmo para transformar os meta-dados existentes na DHT em uma abstração de sistema de arquivos, contendo uma única hierarquia de arquivos e diretórios. Arquivos inseridos no sistema são divididos em vários blocos de dados, que são replicados para vários *peers*, oferendo alta disponibilidade. O CFS tenta manter k réplicas de cada bloco; quando *peers* entram ou saem da rede, os blocos são redistribuídos entre os *peers*.

O Ivy [Muthitacharoen et al. 2002] é um sistema de arquivos distribuído baseado em arquivos de *logs*, que também foi construído sobre a infra-estrutura provida pelo *Chord*. Cada *peer* participante do sistema possui um arquivo de *log* para salvar todas as informações e modificações dos arquivos, que é armazenado na DHT. Qualquer *peer* pode buscar informações sobre os arquivos em quaisquer arquivos de *log* disponíveis na DHT, mas só pode escrever no seu próprio arquivo de *log*. O Ivy garante a consistência dos arquivos através de um *lock* de sessão, onde os dados serão atualizados e tornar-se-ão visíveis somente após o término da sessão de escrita.

OceanStore [Kubiatowicz et al. 2000] é um sistema global de armazenamento, projetado para alta escalabilidade. Ele fornece uma infra-estrutura para armazenamento de dados com alta disponibilidade, consistência e durabilidade, utilizando-se de um grupo de *peers* não confiáveis. Os dados no OceanStore podem migrar livremente entre os *peers* (chamados *nomatic data*) sem intervenção dos usuários. O OceanStore não controla ativamente o número de réplicas de cada arquivo no sistema, cada *peer* pode criar uma réplica de qualquer arquivo a qualquer momento. Para garantir a consistência dos dados entre essas diversas réplicas, é usado um protocolo de consenso bizantino durante as modificações dos arquivos.

O FARSITE [Adya et al. 2002] é um sistema de arquivos distribuído executado por diversas máquinas não confiáveis, que usa várias técnicas de tolerância a faltas para manter a alta disponibilidade dos dados: dispersão e replicação de arquivos, criptografia de arquivos e de comunicação e protocolos de consenso bizantino. Seu objetivo é que um grupo de computadores possa cooperar para criar um servidor de arquivos virtual, onde qualquer arquivo possa ser acessado de qualquer lugar por qualquer usuário. Assim, o FARSITE funciona como se fosse um único servidor central de arquivos. Por sua vez, o PAST [Druschel and Rowstron 2001b] é uma infra-estrutura de armazenamento *p2p* que oferece escalabilidade, disponibilidade dos dados e segurança. O sistema PAST oferece um espaço de nomes único e transparente, onde cada arquivo no sistema recebe um identificador único (*fileid*). O PAST tenta garantir que um número constante de réplicas dos arquivos esteja disponível, mesmo na presença de *peers* faltosos e *peers* entrando e saindo da rede. O sistema PAST foi construído sobre o substrato *p2p Pastry*.

3. O *Transparent File System* (TFS)

O *Transparent File System* (TFS) [Cipar et al. 2007] é um sistema de arquivos local que introduz o conceito de "arquivo transparente". Um arquivo transparente é um arquivo armazenado no disco local, mas que não é visível (ou percebido) pelos usuários e não é contabilizado no consumo do espaço em disco. O oposto ao arquivo transparente, ou seja, o arquivo que é visível ao usuário local, é denominado "arquivo opaco". No TFS, todo o espaço livre em um disco pode ser ocupado por arquivos transparentes. O modelo de alocação de arquivos adotado pelo TFS é o que o diferencia dos outros sistemas de arquivos, pois a persistência dos arquivos transparentes não é garantida. Arquivos opacos têm precedência sobre arquivos transparentes, isto é, arquivos transparentes são armazenados no espaço livre dos disco, mas podem ser removidos ou sobrescritos a qualquer momento por arquivos opacos. O TFS foi implementado como uma modificação do sistema de arquivos EXT2 do Linux e adiciona mais três estados possíveis aos blocos de disco: *Transparent*, *Allocated-and-Overwritten* e *Free-and-Overwritten*. O estado *Transparent* indica que o bloco está sendo usado por um arquivo transparente. O estado *Allocated-and-Overwritten* significa que o bloco estava sendo usado por um arquivo transparente, mas foi sobrescrito por um arquivo opaco. Já o estado *Free-and-Overwritten* indica que o bloco estava sendo usado por um arquivo transparente, foi sobrescrito por um arquivo opaco, e agora está livre novamente.

Através dos arquivos transparentes, o TFS permite que usuários armazenem dados em seu espaço livre em disco, ao mesmo tempo garantindo que esse espaço estará disponível para as aplicações locais que dele necessitarem. Essa propriedade provê uma base interessante para a criação de um sistema de arquivos distribuído que ofereça tolerância a faltas através de replicação: arquivos podem ser replicados no espaço livre de diversos computadores como arquivos transparentes, sem prejudicar a disponibilidade de espaço livre local. Todavia, as aplicações locais podem solicitar o espaço ocupado pelas réplicas para armazenar arquivos opacos a qualquer momento, de forma imprevisível. Assim, o sistema de arquivos distribuído deve reagir prontamente quando uma réplica transparente é removida, para que a disponibilidade dos dados seja mantida. No entanto, a implementação atual do TFS não provê um mecanismo para que as aplicações sejam notificadas quando isso acontece; a invalidação de um arquivo transparente somente é detectada quando o arquivo é aberto.

4. Um Sistema de Arquivos Distribuído usando Arquivos Transparentes

Nesta seção é proposto um sistema de arquivos distribuído *p2p* que utiliza arquivos transparentes para oferecer alta disponibilidade e tolerância a faltas de persistência de arquivos. O sistema proposto é capaz de tratar eficientemente a exclusão ou sobrescrita imprevisível de arquivos transparentes. Ele é composto por diversos nós (*peers*) conectados por uma rede *overlay*, onde cada um compartilha todo o seu espaço livre em disco de forma transparente, por meio do TFS. Nesse espaço compartilhado são armazenadas réplicas dos arquivos pertencentes ao sistema, no formato de arquivos transparentes. Todos os nós possuem as mesmas responsabilidades e realizam as mesmas tarefas: gerenciar as réplicas armazenadas em seu disco, interagir com os outros nós a fim de controlar o número de réplicas disponíveis para cada arquivo e oferecer uma interface para que aplicações possam acessar os arquivos e suas réplicas. O sistema é projetado para prover uma alta disponibilidade dos dados [Avizienis et al. 2004], similarmente ao PAST

[Druschel and Rowstron 2001b] e CFS [Dabek et al. 2001]. Desta forma, os arquivos (ou suas réplicas) estão disponíveis mesmo quando nós estiverem em um estado faltoso ou arquivos transparentes forem removidos. Outros aspectos de sistemas de arquivos distribuídos, como a semântica de consistência, não foram analisados pois o escopo desse trabalho é o estudo da viabilidade do uso de arquivos transparentes.

Adicionalmente, o conceito de "invalidação de réplica" é introduzido. Uma invalidação de réplica acontece quando um arquivo transparente, que é uma réplica de um arquivo disponível no sistema, é excluído ou sobrescrito em um nó. Quando isso acontece, esse nó informa aos demais nós que uma réplica foi removida. Então, o sistema tem a oportunidade de redistribuir as réplicas daquele arquivo, de forma que o número de réplicas disponíveis seja mantido, assegurando a disponibilidade dos dados.

4.1. Modelo Arquitetural

O modelo arquitetural proposto consiste de um conjunto de nós, conectados através de uma rede *p2p overlay*, que usa um sistema de arquivos local transparente para compartilhar seu espaço livre em disco. Todos os nós desempenham funções equivalentes, possuem características similares e trabalham de forma cooperativa, criando assim uma rede *p2p* não estruturada e pura. Cada nó da rede *p2p* é responsável por gerenciar o seu espaço de armazenamento, detectar possíveis remoções de arquivos transparentes, gerenciar a transferência de réplicas para/de outros nós e prover uma interface comum para os usuários. Cada nó possui sete componentes, ilustrados na Figura 1: *Aplicação Distribuída*, *Sistema de Arquivos Distribuído*, *Gestor de Armazenamento*, *Gestor de Replicação*, *Gestor de Transferência*, *Transparent File System Local* e *Substrato P2P*.

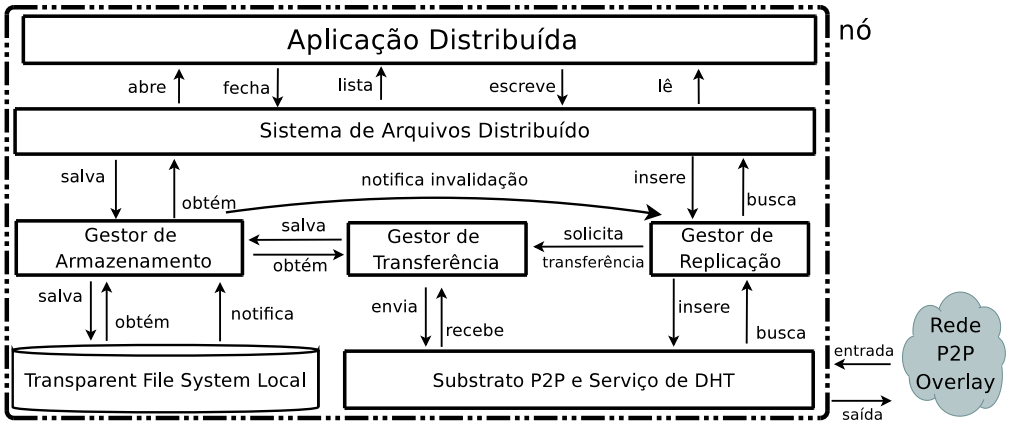

Figura 1. Modelo Arquitetural do Sistema Proposto

A *Aplicação Distribuída* utiliza a camada de sistema de arquivos distribuído para prover algum serviço específico aos seus usuários. Alguns exemplos de aplicações distribuídas que se beneficiariam desta proposta são discutidos na seção 4.5. O *Sistema de Arquivos Distribuído* é responsável por fornecer uma interface similar aos sistemas de arquivos comuns e por controlar o comportamento geral do sistema. Ele opera o *Gestor de Armazenamento* e o *Gestor de Replicação* para armazenar e buscar arquivos através do sistema de arquivos local ou da rede *p2p*. Vários tipos de aplicações podem ser construídos com essa camada, por isso sua semântica e suas operações dependem das características solicitadas pela camada superior.

O *Gestor de Armazenamento* é responsável por armazenar e buscar arquivos transparentes no disco local e monitorar o disco para identificar invalidações de arquivos transparentes, que devem ser informadas imediatamente ao *Gestor de Replicação*. O *Gestor de Replicação* possui a tarefa de manter constante o número de réplicas disponíveis de cada arquivo. Periodicamente, ele verifica o número de réplicas disponíveis para cada arquivo de sua posse; caso alguma insuficiência seja percebida, ele inicia um processo de replicação, a fim de restabelecer o número correto de réplicas. Além disso, ele executa o procedimento de recuperação de réplicas descrito na seção 4.4.

O *Gestor de Transferência* tem a função de enviar e receber réplicas através da rede *p2p overlay*. Ele entra em ação quando uma réplica não está disponível localmente, para transferí-la de outro nó, ou quando o *Gestor de Replicação* detecta alguma inconsistência no número de réplicas disponíveis e solicita transferências de réplicas. O *Transparent File System Local* é responsável por armazenar as réplicas localmente, como arquivos transparentes. Por fim, o *Substrato P2P* é responsável por conectar todos os nós em uma rede *p2p overlay*, gerenciar a entrada e saída de nós da rede e rotear mensagens entre eles.

4.2. Propriedades do Substrato p2p

Substratos *p2p*, como Chord [Stoica et al. 2001], Pastry [Rowstron and Druschel 2001] e Tapestry [Zhao et al. 2004], implementam técnicas de *Consistent Hashing* [Karger et al. 1997] para conseguir mapear uma chave a um nó. Normalmente, os substratos *p2p* definem três conceitos principais que governam o funcionamento do sistema distribuído:

- *Node Identifier* ($nodeId$): identificador único que pertence somente a um nó. Esse identificador é usado para localizar os nós e rotear mensagens entre nós;
- *Neighborhood* ($\mathbb{N}$): conjunto que contém os nós vizinhos mais próximos. Essa medida de proximidade é feita através de uma métrica de localidade, como por exemplo o número de *hops* entre os nós;
- *Leafset* ($\mathbb{L}$): conjunto que contém os nós cujos identificadores são numericamente mais próximos ao identificador do nó em questão.

Essas três definições ($nodeId$, $\mathbb{N}$ e $\mathbb{L}$) são fornecidas pelo substrato p2p e utilizadas pelo sistema proposto a fim de que as informações sobre os arquivos, o conteúdo dos arquivos e as mensagens do sistema possam trafegar pela rede de forma eficiente.

4.3. Acesso aos Arquivos

Sistemas de arquivos distribuídos devem tratar requisições de acesso aos arquivos espalhados em diversos nós de forma similar aos sistemas de arquivos locais. Este processo envolve a troca de mensagens entre os nós, para que os meta-dados e o conteúdo do arquivo possam ser acessados. Além disso, como o sistema proposto tem como premissa manter uma alta disponibilidade dos dados, é necessário então que os meta-dados e o conteúdo sejam replicados para vários nós. O sistema é composto por N nós $p_1 \dots p_N$. Um arquivo disponível no sistema de arquivos distribuído é denotado como f, o seu tamanho como $size(f)$, o seu conjunto de meta-dados é denominado $metadata(f)$, e o seu identificador de $fileid(f)$. Uma réplica do arquivo f armazenada no nó p_i é chamada de $r_i(f)$ e o número de réplicas que deve ser mantido para cada arquivo é chamado de fator de replicação k. O conjunto de nós que armazenam uma réplica de f é chamado de $\mathbb{R}(f)$.

Os nós vizinhos do nó p_i e o seu *leafset* são denominados $\mathbb{N}(p_i)$ e $\mathbb{L}(p_i)$, respectivamente. O espaço disponível em disco no nó p_i é chamado $fspace(p_i)$.

Quando é solicitada a inserção de um novo arquivo no sistema a partir do nó p_i, a *Aplicação Distribuída* usa a interface provida pela camada *Sistema de Arquivos Distribuído* para informar o nome e a localização do arquivo f. Então, um identificador único para o arquivo f ($fileid(f)$) é gerado e os seus meta-dados $metadada(f)$ são obtidos do sistema de arquivos local. A seguir, o *Gestor de Replicação* identifica quais nós $\mathbb{R}(f)$ devem armazenar as réplicas do arquivo, com base no fator de replicação k e na proximidade numérica entre o $fileid(f)$ e identificador dos nós *nodeId* existentes no *leafset* $\mathbb{L}(p_i)$. Assim, os meta-dados do arquivo são armazenados na DHT e os nós selecionados são informados da sua nova responsabilidade: manter uma réplica do arquivo f em seu disco local. O nó p_i envia uma mensagem de requisição de replicação $repl\_req(f)$ para todos os nós pertencentes a $\mathbb{R}(f)$. Então, cada nó solicita ao nó p_i a transferência do conteúdo do arquivo $file\_req(f)$ através do seu *Gestor de Transferência*. Após o fim da transferência, a réplica é armazenada localmente como um arquivo transparente. Essa seqüência de atividades está indicada no Procedimento 1.

Procedimento 1 Aplicação no nó p_i insere um arquivo f

1: Define $\mathbb{R}(f) \subseteq \mathbb{L}(p_i)$ com $|\mathbb{R}(f)| = k$
2: Armazena $metadata(f)$ na DHT
3: **for all** $p_j \in \mathbb{R}(f)$ **do**
4: Envia a mensagem $repl\_req(f)$ para p_j
5: Recebe a mensagem $file\_req(f)$ de p_j
6: Transfere $r_i(f)$ para p_j
7: **end for**

Quando a aplicação no nó p_i deseja acessar um arquivo f, o sistema executa os passos mostrados no Procedimento 2. O *Sistema de Arquivos Distribuído* primeiramente verifica se existe já uma réplica de f armazenada localmente. No caso positivo, a requisição é atendida imediatamente. Caso contrário, o *Gestor de Replicação* busca os meta-dados do arquivo f ($metadata(f)$) na DHT e busca um nó p_k que possua uma réplica do arquivo ($r_k(f)$)[1]. Então, o *Gestor de Transferência* solicita a transferência de uma réplica do arquivo f de p_k ($file\_req(f)$) e a armazena localmente, como um arquivo transparente. Após o término da transferência, a solicitação da aplicação é atendida.

4.4. Invalidações de Réplicas

Devido ao compromisso adotado pelo TFS, onde a persistência dos arquivos transparentes é preterida em relação à disponibilidade de disco para os arquivos locais, um problema surge com o uso de arquivos transparentes na criação de sistemas de arquivos distribuídos: réplicas podem desaparecer. Agora, além de tolerar faltas de nós, deve-se tolerar faltas de persistência nos dados, para que o sistema opere de forma adequada. Para tal, um procedimento de recuperação de réplicas é proposto. Esse procedimento tem como objetivo manter constante o número de réplicas de cada arquivo disponíveis no sistema, sendo

[1]O substrato p2p é responsável por garantir que, se pelo menos um nó que mantém uma réplica de f estiver disponível, essa busca será executada com sucesso.

Procedimento 2 Aplicação no nó p_i solicita um arquivo f

1: Busca $metadata(f)$ da DHT
2: **while** $\nexists r_i(f) \wedge (\mathbb{R}(f) \neq \phi)$ **do**
3: Busca $p_k \in \mathbb{R}(f)$ da DHT
4: Envia $file\_req(f)$ para p_k
5: Recebe $r_k(f)$ de p_k e salva como $r_i(f)$
6: **end while**
7: **if** $\exists r_i(f)$ **then**
8: Devolve $r_i(f)$ para a aplicação
9: **else**
10: **Erro:** nenhuma réplica disponível
11: **end if**

iniciado quando uma réplica é invalidada em um nó. Além disso, os nó trocam mensagens periodicamente para saber quantas réplicas existem; quando um nó não responde, o substrato P2P é ativado para verificar se ele deve ser classificado como faltoso.

Quando o *Gestor de Armazenamento* do nó p_i percebe que uma réplica local $r_i(f)$ foi removida ou sobrescrita, ele envia uma notificação de invalidação de réplica $repl\_inv(f, fspace)$ para o *Gestor de Replicação*. Se existir espaço no disco local para restaurar a réplica[2], o *Gestor de Replicação* procura na DHT um nó p_j que contenha uma réplica de f ($p_j \in \mathbb{R}(f)$) e envia uma solicitação de transferência de arquivo $file\_req(f)$ para p_j. Caso contrário, se não existe espaço livre suficiente, o *Gestor de Replicação* identifica o conjunto de nós que possuem uma réplica de f ($\mathbb{R}(f)$) e envia uma solicitação de replicação $repl\_req(f)$ para o primeiro nó p_k em seu *leafset* $\mathbb{L}(p_i)$ que não esteja armazenando uma réplica do arquivo f. Os passos executados pelo *Gestor de Replicação* no nó p_i são descritos no Procedimento 3.

Procedimento 3 Gestor de Replicação no nó p_i recebe uma notificação de invalidação de réplica $repl\_inv(f, fspace)$

1: **if** $fspace \geq size(f)$ **then**
2: Busca $p_j \in \mathbb{R}(f)$ da DHT
3: Envia mensagem $file\_req(f)$ para p_j
4: **else**
5: Busca $\mathbb{R}(f)$ da DHT
6: Busca $p_k = first(\mathbb{L}(p_i) \backslash \mathbb{R}(f))$ da DHT
7: Envia mensagem $repl\_req(f)$ para p_k
8: **end if**

Para completar o procedimento de restauração, o *Gestor de Replicação* do nó p_k executa o Procedimento 4 após receber a solicitação de replicação $repl\_req(f)$ do nó p_i. O primeiro passo é verificar se já existe uma réplica do arquivo f armazenada em p_k, através de uma consulta ao *Gestor de Armazenamento*. Se p_k não possui uma

[2]Arquivos transparentes podem ser invalidados mesmo se houver muito espaço livre no disco, devido ao algoritmo de alocação de blocos de disco usado no sistema de arquivos EXT2, que busca minimizar a fragmentação dos arquivos normais (opacos).

réplica de f e possui espaço livre em disco suficiente, uma busca na DHT é executada para localizar outro nó p_m que possua uma réplica de f ($p_m \in \mathbb{R}(f)$). Finalmente, uma requisição de transferência de arquivo $\mathit{file\_req}(f)$ é enviada para p_m. Assim, o nó p_k se torna responsável por armazenar uma replica de f, no lugar de p_i.

Entretanto, se p_k não tem espaço livre suficiente para armazenar uma réplica do arquivo, a solicitação de replicação $repl\_req(f)$ recebida de p_i é encaminhada para o próximo nó em seu *leafset* $\mathbb{L}(p_k)$ que não possua uma réplica de f. Por outro lado, se o nó p_k já possui uma réplica de f, isto significa que múltiplas invalidações de réplicas de f ocorreram simultaneamente e o nó p_k foi selecionado durante a execução do Procedimento 3 por outro nó. Então, a solicitação de replicação $repl\_req(f)$ deve ser re-encaminhada, para que o número de réplicas disponível seja mantido.

Procedimento 4 Nó p_k recebe uma requisição de replicação $repl\_req(f)$ do nó p_i

1: **if** $\nexists r_k(f) \wedge (\mathit{fspace}(p_k) \geq \mathit{size}(f))$ **then**
2: Busca $p_m \in \mathbb{R}(f)$ da DHT
3: Envia mensagem $\mathit{file\_req}(f)$ para p_m
4: **else**
5: Busca $p_m = \mathit{first}(\mathbb{L}(p_k) \backslash \mathbb{R}(f))$ da DHT
6: Encaminha mensagem $repl\_req(f)$ para p_m
7: **end if**

4.5. Aplicações de Exemplo

Esta seção discorre sobre duas aplicações-exemplo que se beneficiariam do sistema proposto, sendo que cada uma possui operações e semântica distintas. A primeira aplicação é uma biblioteca digital distribuída, na qual seus usuários podem publicar e pesquisar conteúdos. Uma biblioteca digital deve prover um espaço de nomes único e independente: a mesma hierarquia dos documentos, independentemente de onde eles estão armazenados. Dessa forma, cada documento deve ter um identificador único, pertencer a uma categoria e ser digitalmente assinado pelo seu autor, para que sua integridade seja garantida. Já a garantia de consistência dos dados pode ser obtida através um mecanismo de *lock* único para escrita, porque somente o autor poderá modificar os seus arquivos. Após a modificação, o *lock* do arquivo é removido e as mudanças são propagadas para as réplicas.

A segunda aplicação é um sistema de *backup* distribuído, no qual arquivos de vários usuários são armazenados em diversos computadores. Para esses sistemas, a segurança e privacidade dos dados e do usuário é fundamental. Um sistema de *backup* distribuído deve oferecer um espaço de nomes privado para cada usuário, no qual somente o proprietário poderá identificar, abrir e modificar os seus arquivos. Ainda, os meta-dados e o conteúdo dos arquivos devem ser cifrados. A consistência dos dados pode ser garantida de forma simples, onde os arquivos são atualizados no computador do usuário, são assinados e cifrados, sendo depois replicados para os outros nós.

5. Implementação e Avaliação

Foi implementado um protótipo do sistema proposto, com o intuito de avaliar o uso de arquivos transparentes na implementação de um sistema de arquivos distribuído e de verificar se os algoritmos propostos são capazes de manter a disponibilidade dos dados. Esse

protótipo serve como uma prova de conceito, possuindo um conjunto mínimo de funcionalidades que permitem avaliar o sistema proposto em condições de operação estável, ou seja, onde somente faltas de persistência dos arquivos transparentes ocorrem. Posteriormente, este protótipo pode ser estendido para implementar umas das aplicações descritas na seção 4.5.

5.1. Implementação do Protótipo

O protótipo foi implementado usando quatro projetos *open source*: FreePastry [Druschel and Rowstron 2001a], INotify [Love 2005], JNotify [Yadan 2005] e o TFS [Cipar et al. 2007]. Eles foram adaptados e integrados através de 2.500 linhas de código Java. FreePastry é uma implementação livre, escrita em linguagem Java, do protocolo de roteamento e do substrato *p2p* Pastry. Além disso, ele também contém uma implementação do PAST e de um serviço de DHT. O INotify é um módulo de núcleo do Linux que provê mecanismos para receber notificações do sistema de arquivos local: uma notificação é gerada se um arquivo ou diretório for modificado, apagado, criado ou renomeado. O JNotify é uma biblioteca que fornece uma interface em Java para a API do INotify. Através dessa biblioteca é possível criar aplicações em Java capazes de monitorar todos os eventos gerados pelo INotify.

O conjunto de classes implementadas pelo código em Java é responsável por integrar todos esses componentes e oferecer a base para o desenvolvimento de aplicações que utilizem arquivos transparentes em um sistema de arquivos distribuído. O protótipo implementa um sistema de arquivos simples e plano, onde existe somente um diretório e todos os arquivos estão armazenados dentro desse diretório. Além disso, três operações principais são implementadas: a inserção de um arquivo no sistema, a busca de arquivos e o tratamento das invalidações de réplicas. Essas operações implementam os quatro procedimentos apresentados na seção 4.

5.2. Avaliação do Protótipo

Os experimentos foram realizados em um computador HP AMD Turion 64 X2 2.1 Ghz, 3GB de memória RAM e 250 GB de disco rígido. Nesse computador foram instalados: o sistema operacional Linux Debian Sarge 3.1 com *kernel* 2.6.13, o módulo do TFS para o *kernel* e o *Java2SE Runtine Environment* da Sun na versão 1.6.06. Os nós participantes na rede *p2p* são executados nessa máquina, compartilham a mesma máquina virtual Java, possuem um espaço individual para armazenar o seus arquivos transparentes e são conectados entre si através de uma *bridge* virtual. Com essa configuração busca-se simular um ambiente real, onde cada nó do sistema de arquivos distribuído compartilhará recursos de disco, memória e processador com outras aplicações rodando no sistema local. O sistema é inicializado com 20 nós e o sistema de arquivos distribuído é populado com 25 arquivos de exemplo, cujos tamanhos variam entre 1 MB e 256 MB. Cada arquivo inserido no sistema possui o mesmo fator constante de replicação k igual a 4, isto significa que para cada arquivo existem 3 réplicas disponíveis e espalhadas no sistema, além do arquivo original.

Dois experimentos preliminares foram executados para investigar o comportamento do protótipo na presença de invalidações de réplicas. O valor mostrado foi calculado com base na média dos valores obtidos através de três execuções de cada experimento, e o coeficiente de variação dos resultados obtidos ficou abaixo de 5%. Com o

intuito de criar um ambiente controlado para a remoção e sobrescrita de arquivos transparentes, algumas réplicas são deliberadamente removidas. Essa remoção intencional gera exatamente a mesma cadeia de eventos que ocorreria se um arquivo transparente fosse sobrescrito por um arquivo opaco no TFS. Além disso, o sistema se mantém estável, não existem faltas de parada dos nós, e partições na rede.

O primeiro experimento visou medir o custo adicional causado pelos Procedimentos 3 e 4 de recuperação de réplicas. Nesse experimento foi medido o tempo gasto para que o sistema recupere uma réplica invalidada. Essa medição foi feita em duas situações: (1) quando o nó onde a réplica foi invalidada ainda possui espaço livre em disco para armazenar arquivos transparentes. Então, é suficiente que este nó apenas solicite uma nova réplica a um outro nó; (2) quando o nó onde a réplica foi invalidada não possui mais espaço livre em disco e o sistema precisa achar outro nó que possa armazenar a réplica. A Figura 2 mostra o resultado desse experimento. Pode-se observar que o custo para a execução dos procedimentos de recuperação de réplicas é baixo (inferior a 5%) se comparado com o tempo para transferir a réplica de um nó para outro. Verifica-se porém um custo proporcional e significativo (acima de 30%) somente quando o arquivo tem tamanho abaixo de 4 MB.

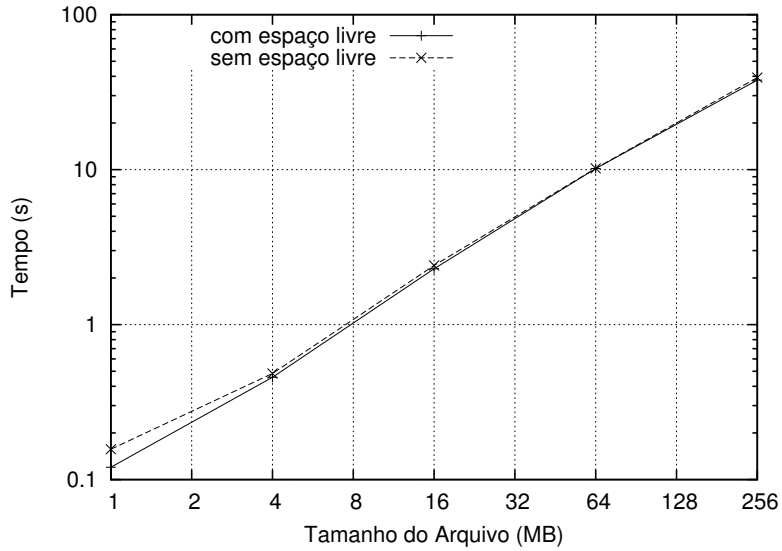

Figura 2. Tempo de recuperação de uma réplica invalidada em um nó com espaço livre em disco e em um nó sem espaço livre em disco.

O segundo experimento teve por objetivo verificar o comportamento do sistema caso várias invalidações de réplicas de um mesmo arquivo ocorram simultaneamente (1, 2 ou 3 invalidações simultâneas de réplicas de um mesmo arquivo). Como a manutenção da alta disponibilidade dos dados é uma premissa do sistema proposto, espera-se que após algum tempo todas as réplicas invalidadas sejam restauradas corretamente na rede. Para exercitar o procedimento completo de recuperação de réplicas, esse experimento foi executado somente no caso em que o nó onde a réplica foi invalidada não possui mais espaço livre em disco. Assim, é necessária uma atuação do mecanismo de auto-organização do sistema para identificar os nós onde serão restauradas as réplicas. A Figura 3 apresenta o resultado desse experimento. Observa-se que após um certo tempo, todas as réplicas destruídas voltam a estar disponíveis no sistema. Nesse experimento foi medido o

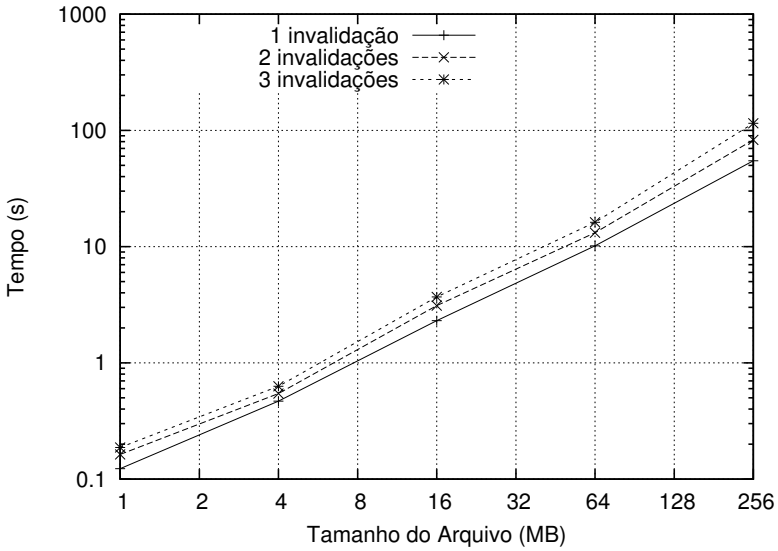

Figura 3. Tempo de recuperação de 1,2 e 3 réplicas invalidadas simultaneamente.

tempo gasto para recuperar todas as réplicas. Nota-se que o tempo gasto está proporcional ao tamanho do arquivo, o que era previamente esperado.

6. Trabalhos Relacionados

O TFS introduziu o conceito de arquivo transparente recentemente [Cipar et al. 2007], desta forma não foi encontrado nenhum trabalho similar que o utilize em um sistema de arquivos distribuído. Entretanto, o compartilhamento de recursos computacionais ociosos vem sendo explorado em outras áreas. Sistemas como o SETIHOME [Anderson et al. 2002] compartilham ciclos ociosos de CPU de computadores domésticos conectados à Internet. Esses ciclos são utilizados na resolução de cálculos científicos complexos, como simulações de modelos climáticos e reações químicas.

[Patterson et al. 1993] e [Karedla et al. 1994] descrevem o uso da memória RAM livre como cache de arquivos, de forma a melhorar o desempenho de aplicações que usam os discos rígidos intensivamente. A memória livre é utilizada para armazenar o conteúdo de arquivos de forma pró-ativa. Desse modo, aplicações poderão ler esses arquivos mais rapidamente, pois já foram copiados do disco para a memória. Quando as aplicações necessitarem de mais memória, a área usada como cache de disco poderá ser requisitada. Já em [Cipar et al. 2006], o uso da memória é controlado pelo sistema operacional de acordo com a finalidade da aplicação. Aplicações distribuídas têm uma prioridade menor do que aplicações locais, desta forma o desempenho das aplicações locais é maior, pois a taxa de faltas de páginas é reduzida.

7. Conclusão e Trabalhos Futuros

Este artigo apresentou e avaliou uma proposta de uso de arquivos transparentes [Cipar et al. 2007] em um sistema de arquivos distribuído baseado no modelo *p2p*. Na proposta, um arquivo a armazenar é replicado em vários nós de um sistema distribuído, usando arquivos transparentes. Como os arquivos transparentes podem desaparecer de forma imprevisível, foi proposto um mecanismo para detectar as invalidações de réplicas locais e algoritmos simples para manter o nível de replicação. Os testes preliminares

mostraram que o método proposto é viável, e o seu custo adicionado é proporcional ao tamanho do arquivo inserido no sistema. Além disso, a ocorrência de múltiplas exclusões simultâneas não impõe uma sobrecarga significativa ao sistema. Experimentos mais detalhados em um ambiente real e com situações dinâmicas (*churn*) estão sendo realizados.

Como este trabalho foi o primeiro a aplicar arquivos transparentes em um sistema de arquivos distribuído, vários pontos podem ser melhorados em trabalhos futuros. Inicialmente, o algoritmo de replicação utilizado é muito simples e não trata situações limítrofes, como a remoção simultânea de todas as réplicas de um mesmo arquivo. Para esse caso, poder-se-ia adicionar algum mecanismo de proteção, no qual uma ação seria iniciada se o número de réplicas disponíveis chegasse a um valor mínimo. Esta ação poderia ser a conversão temporária de um arquivo transparente em um arquivo opaco, de forma a preservá-lo até que suas réplicas sejam restabelecidas. O emprego de algoritmos de replicação mais robustos, como protocolos de consenso bizantino ou quóruns, também está sendo investigada. Por fim, a implementação atual do TFS suporta somente invalidação de arquivos completos. Se somente um bloco de disco pertencente a um arquivo transparente é sobrescrito por um arquivo opaco, o arquivo transparente inteiro é excluído. Os autores do TFS [Cipar et al. 2007] propuseram (mas não implementaram) mecanismos para tratar apenas os blocos sobrescritos. Nessa situação, somente os blocos perdidos poderiam ser restaurados, diminuindo consideravelmente o tempo gasto para a restauração de réplicas invalidadas.

Referências

Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. (2002). Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Operating Systems Review*, 36(SI):1–14.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. (2002). Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61.

Androutsellis-Theotokis, S. and Spinellis, D. (2004). A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

Cipar, J., Corner, M. D., and Berger, E. D. (2006). Transparent contribution of memory. In *USENIX Annual Technical Conference*.

Cipar, J., Corner, M. D., and Berger, E. D. (2007). TFS: a transparent file system for contributory storage. In *5th USENIX Conference on File and Storage Technologies*.

Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. (2001). Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles*.

Druschel, P. and Rowstron, A. (2001a). FreePastry. `http://freepastry.org`.

Druschel, P. and Rowstron, A. (2001b). PAST: A large-scale, persistent peer-to-peer storage utility. In *8th IEEE Workshop on Hot Topics in Operating Systems*.

Golle, P., Leyton-Brown, K., Mironov, I., and Lillibridge, M. (2001). Incentives for sharing in peer-to-peer networks. In *2nd Intl Workshop on Electronic Commerce*.

Hasan, R., Anwar, Z., Yurcik, W., Brumbaugh, L., and Campbell, R. (2005). A survey of peer-to-peer storage techniques for distributed file systems. In *Intl Conference on Information Technology: Coding and Computing (ITCC'05), Volume II*.

Karedla, R., Love, J. S., and Wherry, B. G. (1994). Caching strategies to improve disk system performance. *IEEE Computer*, 27(3):38–46.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *29th ACM Symposium on Theory of Computing*.

Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. (2000). Oceanstore: An architecture for global-scale persistent storage. In *9th ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Leonard, O., Nieh, J., Zadok, E., Osborn, J., Shater, A., and Wright, C. (2002). The Design and Implementation of Elastic Quotas: A System for Flexible File System Management. Technical Report CUCS01402, Computer Science, Columbia University.

Love, R. (2005). Kernel korner: Intro to INotify. *Linux Journal*, 2005(139):8.

Mauthe, A. and Hutchison, D. (2003). Peer-to-peer computing: Systems, concepts and characteristics. *Praxis in der Informationsverarbeitung und Kommunikation (PIK), Special Issue on Peer-to-Peer, Volume 26*.

Milojicic, D., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., and Xu, Z. (2002). Peer-to-Peer Computing. Technical Report 2002-57, HP Labs.

Muthitacharoen, A., Morris, R., Gil, T., and Chen, B. (2002). Ivy: A read/write peer-to-peer file system. In *5th Symposium on Oper. Systems Design and Implementation*.

Patterson, R., Gibson, G., and Satyanarayanan, M. (1993). A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34.

Payne, D. and Woolnough, P. (2004). Bandwidth drivers for future networks. *IEE Telecommunication Series*, 47:21–36.

Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–340.

Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. (2001). Chord: A scalable Peer-To-Peer lookup service for internet applications. In *ACM SIGCOMM*.

wan Ngan, T., Wallach, D., and Druschel, P. (2003). Enforcing fair sharing of peer-to-peer resources. In *2nd Intl Workshop on Peer-to-Peer Systems*.

Yadan, O. (2005). JNotify. `http://jnotify.sourceforge.com`.

Zhao, B., Huang, L., Stribling, J., Rhea, S., Joseph, A., and Kubiatowicz, J. (2004). Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53.