MARCELO CHEMINN MADRUGA

USO DE ARQUIVOS TRANSPARENTES NA CONSTRUÇÃO DE UM SISTEMA DE ARQUIVOS DISTRIBUÍDO TOLERANTE A FALTAS

Dissertação submetida ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para a obtenção do título de Mestre em Informática.

Curitiba PR
Dezembro de 2008

MARCELO CHEMINN MADRUGA

USO DE ARQUIVOS TRANSPARENTES NA CONSTRUÇÃO DE UM SISTEMA DE ARQUIVOS DISTRIBUÍDO TOLERANTE A FALTAS

Dissertação submetida ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para a obtenção do título de Mestre em Informática.

Área de concentração: *Ciência da Computação*

Orientador: Prof. Dr. Carlos Alberto Maziero

Curitiba PR
Dezembro de 2008

Dados da Catalogação na Publicação
Pontifícia Universidade Católica do Paraná
Sistema Integrado de Bibliotecas – SIBI/PUCPR
Biblioteca Central

M183u
2008

Madruga, Marcelo Cheminn.

Uso de arquivos transparentes na construção de um sistema de arquivos distribuído tolerante a faltas / Marcelo Cheminn Madruga ; orientador, Carlos Alberto Maziero. – 2008.

xvi, 66 f. : il. ; 30 cm

Dissertação (Mestrado) - Pontifícia Universidade Católica do Paraná, Curitiba, 2008.

Bibliografia: f. 63-66

1. Arquitetura de computador 2. Sistemas operacionais distribuídos 3. Arquitetura não-hierárquica (Rede de computador) I. Maziero, Carlos Alberto II. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática III. Título.

CDD 20. ed. – 004.22

Esta folha deve ser substituída pela ata de defesa devidamente assinada, que será fornecida pela secretaria do programa após a defesa.

Agradecimentos

Ao Prof. Dr. Carlos Alberto Maziero que, com sabedoria e dedicação, mostrou-me caminhos, apontou-me erros e encorajou-me nos momentos difíceis; assim tornando possível a realização deste trabalho.

Agradeço a minha família e minha namorada, que souberam compreender minhas ausências, minhas angústias e minhas longas noites de estudo. Contudo mantiveram-me focado e deram-me energia para continuar sempre.

Resumo

O modelo *peer-to-peer* e a disponibilidade de largura de banda têm viabilizado a criação de novos sistemas de arquivos distribuídos. Esta dissertação 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.

Palavras-chave: sistemas de arquivos distribuído, arquivos transparentes, peer-to-peer.

Abstract

The peer-to-peer model and the bandwidth availability are fostering the creation of new distributed file systems. This work 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 a replica is invalidated, peers cooperate to restore it. 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.

Keywords: distributed file system, transparent files, peer-to-peer.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

ACID	Atomicity, Consistency, Isolation, Durability
AFS	Andrew File System
API	Application Programming Interface
CBS	Charles Block Service
CPU	Central Processing Unit
CCET	Centro de Ciências Exatas e de Tecnologia
DHT	Distributed Hash Table
GUID	Global Unique Identifier
MS	Metadata Service
NFS	Network File System
P2P	Peer-to-Peer
PPGIa	Programa de Pós-Graduação em Informática
PUCPR	Pontifícia Universidade Católica do Paraná
RAID	Redundant Array of Inexpensive Disks
SFS	Self-Certifying File System
TFS	Transparent File System
XFS	Serverless File System

Capítulo 1

Introdução

1.1 Contexto e Motivação

Com o crescimento da conectividade entre os computadores trazida pela Internet e da largura de banda disponível, sistemas de arquivos e armazenamento distribuído estão migrando de arquiteturas baseadas no modelo cliente/servidor para arquiteturas baseadas no modelo *peer-to-peer* [Mauthe and Hutchison, 2003]. Sistemas baseados no modelo *peer-to-peer* funcionam de forma descentralizada, e possuem a característica de se adaptarem a diferentes situações através de suas propriedades de auto-organização. Sistemas de arquivos e armazenamento distribuído podem ser beneficiados por essas características para fornecer uma melhor tolerância a faltas, maior disponibilidade de espaço para armazenamento e alta disponibilidade dos dados [Androutsellis-Theotokis and Spinellis, 2004].

Tradicionalmente, arquivos pertencentes a aplicações distribuídas e arquivos pertencentes a aplicações locais são tratados/manuseados da mesma forma pelo sistema de arquivos local que armazena os dados no disco. Assim, os dois tipos de arquivos contribuem do mesmo modo no consumo de espaço de armazenamento em disco. No entanto, essa falta de discriminação entre arquivos pertencentes a aplicações locais ou distribuídas traz à tona algumas preocupações e problemas para a implementação e utilização de sistemas baseados no modelo *peer-to-peer*. Existem alguns sistemas *peer-to-peer* de armazenamento distribuído que são vistos meramente como aplicações contributivas, nas quais usuários doam seu espaço livre em disco para o uso dos membros pertencentes ao sistema. Contudo, esse espaço doado não será necessariamente utilizado pelo próprio doador. Para esses sistemas, o comportamento dos usuários em relação à doação de espaço livre em disco é totalmente diferente e varia de acordo com os objetivos de cada usuário. Alguns têm receio de compartilhar ou doar grandes quantidades de espaço livre em disco, pois eles eventualmente precisarão desse espaço compartilhado para fins individuais e essa doação pode afetar significantemente o desempenho da máquina. Já outros desejam doar o máximo possível de espaço livre em disco, porque sabem que poderão pesquisar e utilizar um maior número de arquivos existentes no sistema, conforme é explicado em [Golle et al., 2001].

Ao longo dos últimos anos, foram apresentadas várias idéias para resolver ou mitigar esse problema do comportamento variável dos usuários [wan Ngan et al., 2003] [Leonard et al., 2002]. O conceito de "arquivos transparentes" introduzido em [Cipar et al., 2007] é uma dessas soluções. O *Transparent File System* (TFS) é um sistema de arquivos local que é capaz de doar 100% do espaço livre em disco de uma máquina

a um sistema de arquivos distribuído, sem adicionar um custo muito alto ao desempenho do sistema e do espaço disponível para aplicações locais. O TFS atinge esse objetivo através do uso de "arquivos transparentes". Os arquivos transparentes armazenam dados e seu espaço ocupado no disco pode ser solicitado a qualquer momento por aplicações locais. Desta forma, o usuário não percebe a existência desses arquivos, já que eles não são contabilizados na ocupação do espaço livre.

Este trabalho propõe a utilização de "arquivos transparentes" por sistemas de arquivos distribuído, de forma que a falta de persistência desses arquivos não seja um problema. Além disso, o sistema deve manter um número constante de réplicas de cada arquivo disponíveis. Desta maneira, o espaço livre em disco que pode ser doado é maximizado, o usuário não precisa adotar um comportamento defensivo quanto ao compartilhamento do espaço e o sistema oferece uma alta disponibilidade dos dados.

1.2 Objetivos

Este trabalho tem como objetivo a avaliação do uso de "arquivos transparentes" para a construção de sistemas de arquivos distribuído baseado no modelo *peer-to-peer*, os quais oferecem tolerância a faltas e alta disponibilidade dos dados.

1.2.1 Objetivos Específicos

Os objetivos específicos deste trabalho são:

1. Avaliar um modelo, como prova de conceito, de um sistema de arquivos distribuído mínimo, baseado no modelo *peer-to-peer*, que use o conceito de "arquivos transparentes", o qual não possui impacto no espaço livre para o usuário local;

2. Avaliar o custo adicionado quando "arquivos transparentes" são utilizados.

1.3 Escopo

Esse trabalho é uma prova de conceito, que visa demonstrar a viabilidade no uso de "arquivos transparentes" para criação de sistemas de arquivos distribuído. Por se tratar de uma prova de conceito, a implementação e avaliação do protótipo focou-se apenas nas funções mínimas oferecidas por um sistema de arquivos distribuído, na definição de algoritmos para tratar a remoção involuntária e descontrolada das réplicas dos arquivos, e averiguação do custo adicional ao sistema.

1.4 Organização deste Documento

O trabalho está dividido e organizado da seguinte forma: o capítulo 2 apresenta um estudo comparativo dos sistema de arquivos distribuídos mais relevantes, mostrando as características, a arquitetura, a semântica e as propriedades de cada um. Ao final do capítulo é feita uma comparação das características dos sistemas. O capítulo 3 introduz o conceito de "arquivos transparentes", bem como as suas vantagens e desvantagens; além disso, descreve

o funcionamento do *Transparent File System* (TFS), que é um sistema de arquivos locais que utiliza arquivos transparentes para armazenar dados pertencentes a aplicações distribuídas. O capítulo 4 apresenta a proposta desse trabalho, descrevendo sua arquitetura, suas características e seus algoritmos. Já o capítulo 5 contém um descrição detalhada do protótipo implementado, as principais tecnologias utilizadas e os resultados obtidos durante a execução dos experimentos de avaliação. Por fim, o capítulo 6 traz a conclusão desse trabalho e uma análise sobre possíveis trabalhos futuros.

Capítulo 2

Sistemas de Arquivos Distribuídos

2.1 Introdução

Este capítulo apresenta uma visão geral sobre sistemas de arquivos distribuídos relevantes. Um sistema de arquivos distribuídos armazena dados e informações dos usuários em diversas máquinas, ou servidores, mas sem demonstrar isso aos usuários [Tanenbaum, 1994]. Isto é, para o usuário não existe diferença lógica entre um sistema de arquivos local e um distribuído.

Nas próximas seções alguns sistemas de arquivos distribuídos mais relevantes, tendo em vista o tópico deste trabalho, são apresentados. Foram selecionados dez sistemas de arquivos distribuídos, os quais fornecem uma visão geral do estado da arte nesse assunto, e possibilitam o vislumbramento da evolução tecnológica e complexidade dos sistemas distribuídos. No início criaram-se sistemas baseados no modelo cliente-servidor, e posteriormente esses sistemas foram aprimorados seguindo a evolução na área de sistemas distribuídos, como modelo *peer-to-peer*, tolerância a faltas e segurança. O restante do capítulo discorre em maior detalhes sobre cada um desses dez sistemas em questão, e por fim faz uma comparação entre eles.

2.2 Network File System (NFS)

O *Network File System* (NFS) foi desenvolvido e introduzido pela Sun Microsystems em 1985 [Sandberg et al., 1985]. Desde então, o NFS é o sistema de arquivos distribuídos mais popular e mais difundido no mundo. Com esta imensa popularidade, o NFS sofreu evoluções gerando várias versões. Pawlowski et al descreve em [Pawlowski et al., 1994] o NFSv3, e em [Pawlowski et al., 2000] o NFSv4, além disso a especificação completa do NFSv3 foi publicada pela Sun Microsystems em 1994 [Microsystems, 1994]. Este documento não pretende entrar em detalhes e/ou diferenças entre o NFSv3 e o NFSv4, mas sim explicar em termos gerais a arquitetura e o funcionamento do NFS.

2.2.1 Arquitetura

O NFS foi desenvolvido no paradigma cliente-servidor, no qual existe um servidor, que contém todos os arquivos, e vários clientes que lêem e escrevem arquivos. O servidor exporta (ou disponibiliza) arquivos, diretórios ou subdiretórios para os clientes, que por sua vez

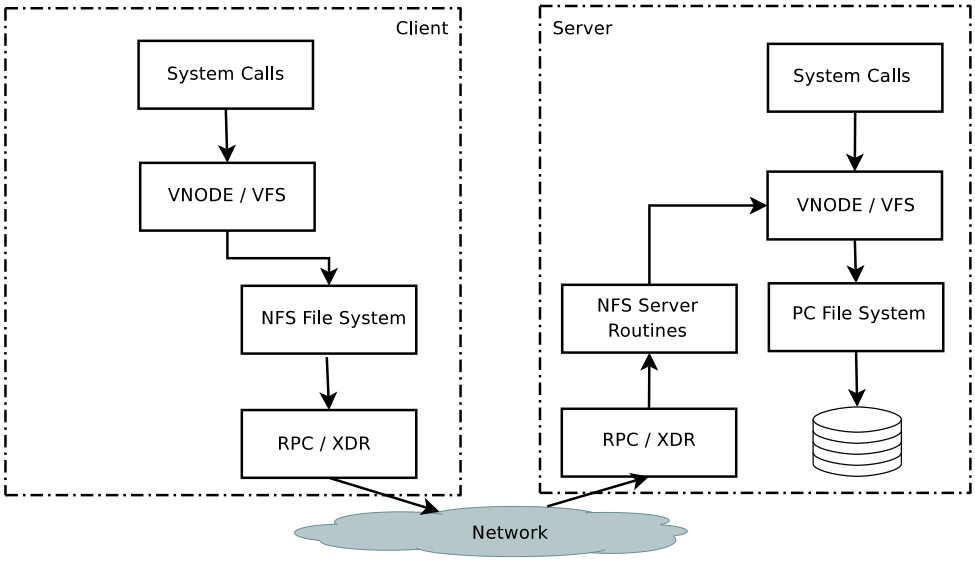

Figura 2.1: Arquitetura NFS (adaptado de [Sandberg et al., 1985])

“montam"(ou mapeiam) os dados localmente. Tanto clientes como servidores precisam saber *a priori* quem exportará quais diretórios e onde serão montados os dados. Desta forma, o NFS precisa de uma estrutura de dados que informe quem são os servidores aos clientes, e indique onde os dados dos clientes serão montados na máquina local.

A comunicação entre cliente e servidor é feita através de *Remote Procedures Calls* (RPCs). Estas RPCs definem todos os tipos de ações que os clientes podem executar em um servidor, como por exemplo: navegar na árvore de diretórios, pesquisar por um arquivo, ler o conteúdo de um diretório, definir atributos de arquivos e diretórios, ler e escrever arquivos.

A figura 2.1 ilustra a arquitetura e o modo de funcionamento do NFS. Todas as chamadas de sistema feitas pelo cliente para o sistema de arquivos ocorrem através do *Virtual File System* (VFS, sistema de arquivos virtual), que é uma camada acima do sistema de arquivos real do sistema operacional onde o NFS é executado. O VFS intercepta todas as chamadas de sistema do cliente destinadas aos arquivos contidos no diretório no qual é montado o sistema remoto. Assim, estas chamadas são redirecionadas ao cliente NFS, que posteriormente as transfere para o servidor NFS adequado. Quando a requisição chega no servidor NFS, o VFS redireciona para o sistema de arquivos real do servidor, e este responde para o VFS, que devolve para o servidor NFS. Todo o NFS foi desenhado para ser *stateless*, ou seja, cada requisição do cliente possui todas informações necessárias para ser atendida pelo servidor imediatamente. Desta forma, a implementação do servidor NFS é simplificada, pois ele não precisa manter informações sobre o estado de cada requisição. No entanto, o NFSv4 prevê alguma modificação nesse protocolo para suportar operações de *locking*, assim o NFSv4 não é completamente *stateless*.

2.2.2 Espaço de Nomes

Cada servidor NFS pode exportar o sistema de arquivos completo, ou somente uma subárvore; além disso deve indicar as restrições de acesso e permissões. O cliente NFS, por sua vez, deve indicar onde os dados exportados serão montados, desta forma cada cliente poderá ter os mesmos arquivos (ou a mesma subárvore) em estruturas de diretórios totalmente diferentes. O NFS também permite que o cliente use pontos de montagem recursivos. Isto é, uma subárvore

de um servidor poderá possuir como ponto de montagem um lugar onde outra subárvore de outro servidor NFS já estiver montada. Contudo, toda essa flexibilidade pode criar caminhos e um espaço de nomes totalmente desestruturados. Por exemplo, um servidor NFS exporta duas subárvores de diretórios `/diretorio1` e `/diretorio2`. Um cliente pode montar esta subárvore em `/usr/local/diretorio1` e aquela em `/var/diretorio2`; e outro cliente poderia montar em `/local/nfs/diretorio1` e `/local/nfs/diretorio2`.

Portanto, o NFS possui um espaço de nomes independente, mas não transparente. É independente pois os mesmos arquivos e subárvores serão montadas em todos os clientes; mas não transparente, porque cada cliente selecionará o ponto de montagem da melhor forma para si.

2.2.3 Tolerância a Faltas

Tolerância a Faltas não foi um aspecto considerado na criação do NFS. Ele não possui nenhum mecanismo intrínseco para replicação de arquivos e garantia de uma alta disponibilidade dos dados. Sendo assim, se um servidor NFS se tornar indisponível, todas as requisições para ele serão terminadas com um erro e novas requisições não serão possíveis. Somente no NFSv4 especificaram-se operações primárias e rudimentares para tolerância a faltas, como por exemplo, uma resposta contendo o endereço do novo servidor caso os dados migrem de um servidor para outro.

2.2.4 Semântica de Consistência

O NFS possui uma característica interessante, não suporta nenhum mecanismo para garantia de consistência, no entanto usa intensivamente cache. Quando um cliente solicita a leitura de um arquivo, os atributos (meta-dados) e alguns blocos de dados deste arquivo são trazidos para o cache. O cache segue uma técnica de leitura-avançada (*read-ahead*), na qual os blocos solicitados e mais alguns são copiados do servidor para o cliente. Desta forma, na próxima leitura, o cliente verifica se os próximos blocos já estão no cache, senão solicita ao servidor. O cache é monitorado a cada 60 segundos, ou seja, se arquivos neste intervalo não são usados, eles serão descartados do cache.

O NFS usa um modelo de “escrita-atrasada"(*write-back*), no qual o cache do cliente NFS é verificado em intervalos de tempos definidos, e quando são encontradas mudanças elas são enviadas ao servidor. Uma atividade de escrita é considerada completa somente após os dados chegarem no disco do servidor. O NFS não oferece um controle de concorrência e consistência, pois se dois clientes estiverem com o mesmo arquivo aberto, somente a última escrita será armazenada.

2.3 Andrew File System (AFS)

O *Andrew File System* ou AFS teve o seu desenvolvimento iniciado em 1983 em uma parceria entre a *Carnegie Mellon University* (CMU) e a IBM. O objetivo primordial do AFS foi promover um ambiente computacional adequado para que pesquisadores, professores e estudantes pudessem compartilhar informações e aumentar a colaboração entre seus usuários. A meta foi que até 10.000 computadores pudessem ser conectados transparentemente no sistema de arquivos proposto pelo AFS.

Assim, com este grandioso objetivo em mente, o foco da arquitetura do AFS é a escalabilidade e o desempenho, pois os usuários não poderiam perceber grandes diferenças entre usar um sistema de arquivo local ou o AFS. Desta forma o AFS ganhou um grande número de adeptos e foi instalado em várias universidades do EUA, como *Massachuchets Institute of Technology* (MIT), *University of Michigan*, entre outras. Em 2001 foi criado o OpenAFS, um projeto de código aberto e livre, graças ao anúncio da IBM de disponibilizar o código fonte do AFS.

O artigo [Howard, 1988] explica como é o AFS de maneira geral, já em [Howard et al., 1988], Howard explica em detalhes o funcionamento e a arquitetura do AFS, e mostra os resultados detalhados dos testes de desempenho.

2.3.1 Arquitetura

O AFS foi construído de acordo com o modelo cliente-servidor. Existe uma diferença bem clara e definida entre clientes e servidores. Os servidores são máquinas dedicadas e totalmente confiáveis, e os clientes são vários e não tão confiáveis assim. Desta forma o AFS define duas entidades, o *Vice* e o *Venus*. *Vice* é o componente que se localiza no servidor e é responsável por controlar toda a operação. *Vênus* é o componente que se encontra no cliente e faz interface com o *Vice* para realizar tarefas como ler, escrever ou buscar arquivos. Além disso, os *Vices* se comunicam com outros *Vices* também.

O componente principal do *Vênus* é o *Andrew Cache Manager*, responsável por controlar todos os acessos do cliente aos arquivos e garantir um bom desempenho através de algoritmos de cache. Já no *Vice*, existem quatro componentes: *File Manager, Authentication Manager, Update Manager e Status Manager*. O *File Manager* é responsável por controlar todas as operações nos arquivos, leitura, escrita, *locks*, etc. O *Authentication Manager* mantém uma base de dados de usuários e deve autenticar os usuários antes que eles possam acessar os arquivos. O *Update Manage*r mantém a consistência da base de usuários e do sistema de arquivos. O *Status Manager* é o ponto central de informações, no qual o estado dos vários servidores de arquivos é mantido.

O funcionamento do AFS é bem direto: quando um usuário deseja abrir um arquivo (em um *Vênus*) ele faz uma chamada ao sistema operacional. Esta chamada é redirecionada ao *Andrew Cache Manager*, que inicia a comunicação com o *File Manager* em um Vice. O *File Manager* responde e começa a transmitir os dados do arquivo. Quando o usuário deseja escrever em um arquivo, ele escreve localmente e o *Cache Manager* solicita um *lock* para o servidor. Após o reconhecimento do servidor, o *Cache Manager* transmite os novos blocos do arquivo modificado. Portanto, percebe-se que o AFS possui um protocolo que controla os estados dos clientes e servidores, contrariamente ao NFS que é *stateless*.

O AFS divide as partições do sistema de arquivos dos vários servidores em volumes. Cada volume é um conjunto pequeno e consistente de diretórios e arquivos. Essa abstração é o ponto principal do AFS, que o torna flexível e eficiente. Uma prova disso é que os volumes podem ser replicados e armazenados em outros servidores sem perda de disponibilidade para os clientes.

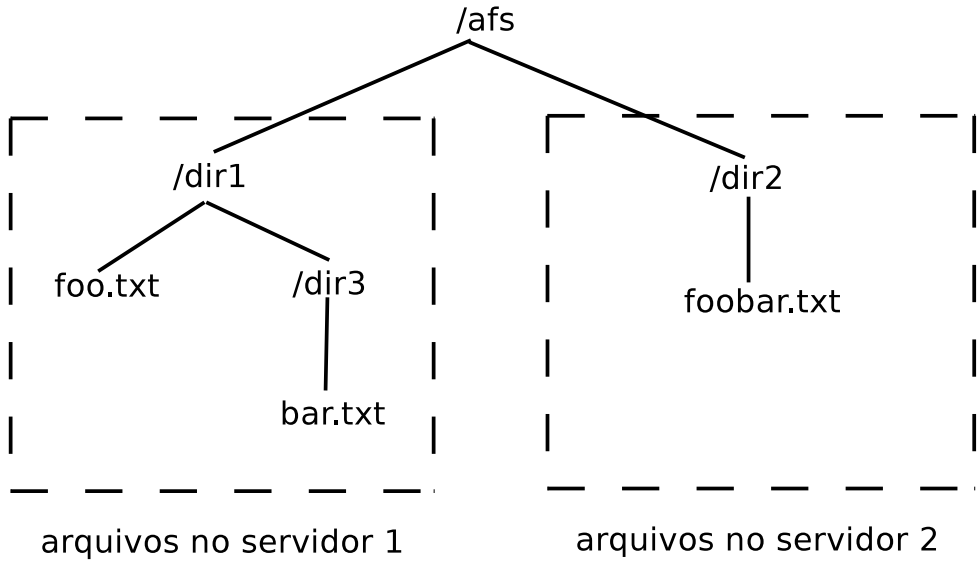

Figura 2.2: Transparência de espaço de nomes no AFS (adaptado de [Howard et al., 1988])

2.3.2 Espaço de Nomes

Os clientes montam os arquivos dos servidores do AFS em um único ponto `/afs`. Desta forma, o AFS implementa para os clientes uma transparência de localização e independência de local; pois os arquivos estão sempre no mesmo caminho independentemente onde eles se encontram. Supondo a estrutura de um sistema de arquivos ilustrado na Figura 2.2, os diretórios `dir1` e `dir2` estão abaixo de `/afs` no cliente, mas estão fisicamente em dois servidores distintos. Isto é, para o cliente existirá sempre o diretório `dir1` abaixo de `/afs` independentemente de onde os dados estejam realmente armazenados.

2.3.3 Tolerância a Faltas

Como já dito, volumes são a chave para a arquitetura do AFS; assim os volumes permitem que o AFS seja um sistema de arquivos distribuído tolerante a faltas. Volumes podem ser livremente copiados e replicados para diversos Vices, ou servidores, dentro a rede do AFS, já que existe uma independência no espaço de nomes. Desta forma, se o AFS identificar que um servidor está em estado de falta, ele redireciona a solicitação do cliente para o servidor réplica mais próximo. Existe um único detalhe para a replicação dos volumes: somente os volumes marcados como *read-only* podem ser copiados livremente. Portanto, não é necessário implementar um controle de concorrência para os arquivos replicados.

2.3.4 Semântica de Consistência

O AFS provê uma semântica de sessão, ou seja, todas as operações são controladas e validadas enquanto a sessão estiver disponível. Quando um cliente abre um arquivo, ele é salvo no seu cache. Todas as operações de leitura e escrita são feitas na cópia local do arquivo. Somente após a solicitação de fechar o arquivo que os dados modificados são enviados ao servidor.

A forma com que o AFS implementa este comportamento é baseado em um controle de estados dos diversos clientes. Quando um cliente abre o arquivo, o servidor registra essa operação, ou seja, que um determinado cliente está com um certo arquivo aberto. Então, quando outro cliente abre o mesmo arquivo, o servidor saberá que outro cliente está com o mesmo arquivo aberto. Logo que algum cliente feche o arquivo, todos os registros de quem está com o arquivo aberto são apagados. Assim, o cliente que está com o arquivo aberto deve solicitar novamente a abertura do arquivo para garantir que está lendo a última versão. Portanto, o AFS garante a consistência dos dados através da sessão e de forma otimista, pois supõe que o número de escritas simultâneas é baixo.

2.4 Serveless File System (xFS)

Anderson em [Anderson et al., 1995] apresenta um sistema de arquivos em rede sem um servidor central, o xFS. O xFS é um protótipo feito com o intuito de demonstrar a viabilidade e usabilidade de um sistema de arquivos distribuídos sem um servidor principal, confrontando com todos os sistemas de arquivos distribuídos da época que usam o modelo cliente-servidor.

Os sistemas usuais, ou seja, com um servidor principal controlando todas as operações, possuem dois pontos de contenção bem claros: desempenho e confiabilidade, porque todos os serviços são providos pelo servidor, e esse pode ser comprometido, ou não atender a todos os clientes suficientemente. Já um sistema sem este servidor, mas no qual as tarefas são distribuídas entre todos os elementos participantes, tenta resolver esses problemas.

Este conceito de *serveless system* ou sistema sem um servidor central de controle na época era novo e dizia-se que seria muito útil nas futuras redes, pois novos tipos de aplicação (como multimídia, processamento em paralelo, grande quantidades de dados, etc) iriam necessitar de sistemas onde o desempenho das chamadas de E/S (entrada/saída) fossem bem maior. Sabia-se que operações de E/S são as mais demoradas para um sistema computacional. Desta forma, o xFS foi o pioneiro na criação de sistemas de arquivos sem um servidor central, ou seja, que seguem o modelo *peer-to-peer*, criado posteriormente.

2.4.1 Arquitetura

O xFS [Anderson et al., 1995] foi desenvolvido tendo como visão uma simples frase: *"anything, anywhere"*. Em consequência disso, sua arquitetura é bem modular e permite que cada um de seus elementos seja executado em qualquer elemento da rede. Qualquer informação do sistema, como metadados, dados e controle podem estar em qualquer lugar da rede, e ainda pode migrar de local a qualquer momento.

Cada componente do xFS pode estar em qualquer elemento da rede. A Figura 2.3 contém dois exemplos de sua instalação, um com elementos dedicados (mais abaixo) e outro com todos os elementos compartilhando as mesmas máquinas. O xFS é composto por 4 tipos de entidades: *Clients, Managers, Cleaners e Storage Servers*. Além disso toda operação realizada por algumas destas entidades é gravada em *log*. Desta maneira o xFS pode executar operações de recuperação baseada em todos os *logs* do sistema.

Managers são responsáveis por controlar todos os metadados do sistema, sendo que cada um controla um espaço de endereçamento de arquivos. Este endereçamento é controlado por uma estrutura chamada *Manager Map*. Em outra palavras, o *Manager Map* contém um mapeamento de quais endereços de arquivos estão em poder de quais *Managers*. Assim, quando

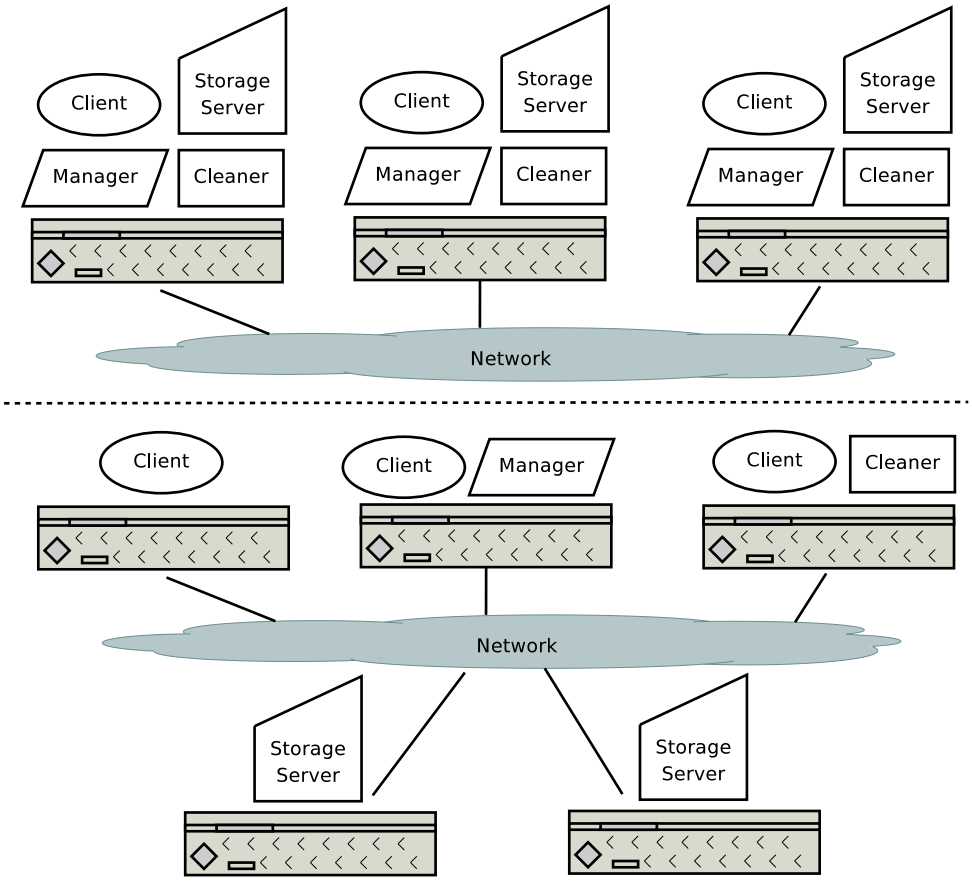

Figura 2.3: Arquitetura e componentes do xFS (adaptado de [Anderson et al., 1995])

um *Client* pede acesso a um determinado arquivo, o sistema procura o *Manager* que possui o arquivo.

Outra tarefa dos *Managers* é controlar o cache dos clientes de uma forma cooperativa. Isto é, informações de cache podem ser migradas de cliente para cliente, sem a necessidade de passar por um servidor de cache central. Isto facilita o acesso e aumenta o desempenho em alguns casos.

Storage Servers têm a função de armazenar os dados em disco, armazenar os logs e prover informações para operações de recuperação. Estes servidores são controlados por uma estrutura chamada de *Stripe Group Map*, que segue o modelo RAID (*Redundant Array of Inexpensive Disks*) [Patterson et al., 1988] de larga escala. Isto é, o espaço de armazenamento nos *Storage Servers* são divididos em grupos e faixas; e cada arquivo está escrito em um fragmento de log e em um grupo de faixas.

Cleaners fazem o papel de verificar e limpar todos os logs do sistema. Quando novos blocos dos arquivos são escritos no sistema, novos segmentos do log são utilizados, no entanto poder-se-ia utilizar segmentos antigos mas que estão livres pois seus conteúdos já foram deletados. A função dos *Cleaners* é manter estes logs consistentes e sem fragmentação.

2.4.2 Espaço de Nomes

O xFS oferece um espaço de nomes único para todos os clientes. Como os arquivos e diretórios estão espalhados por diversos *Storage Servers*, a unicidade do espaço de nomes é uma solução elegante e que atende todos os clientes. Todos os diretórios e arquivos, a partir do ponto de montagem, são sempre os mesmos. Portanto, o xFS oferece ao usuário transparência de localização e independência de local.

2.4.3 Tolerância a Faltas

Como o xFS é um sistema no qual não existe um servidor controlador central, tolerância a faltas é uma premissa básica para o funcionamento do sistema. Clientes não podem ter suas operações terminadas somente porque um elemento do xFS está com falhas. O xFS usa várias técnicas para garantir isso.

Primeiramente, o xFS utiliza a técnica de RAID nos *Storage Servers*. Todos os logs são armazenados localmente, mas com uma cópia de redundância. Existem *checkpoints* que são executados por todos os elementos do sistema em determinadas situações. Além disso, estruturas importantes como os *Manager Maps* e o *Stripe Group Map* são replicados globalmente pelo sistema.

O xFS também possui um modelo bem definido de recuperação, assim qualquer componente que entre em um estado inválido e faltoso poderá voltar ao estado correto. Todo o processo de recuperação é baseado nos logs e nos *checkpoints*.

2.4.4 Semântica de Consistência

Todas as escritas no xFS são inicialmente feitas no cache do cliente. No entanto, para que ele possa escrever, deverá solicitar ao *Manager* responsável pelo arquivo um *lock* de escrita. Quando o *Manager* recebe esta solicitação ele invalida todos os outros caches que possuem este arquivo. Somente depois que o *lock* é liberado pelo *Manager*, o cliente pode escrever no

arquivo. O cliente que solicitou a permissão de escrita a mantém até que outro cliente a solicite. Com este simples modelo de gerência de cache, o xFS oferece a condição mínima para garantir a consistência dos dados no sistema.

2.5 Google File System

Nos últimos anos, a demanda por um novo modelo de sistema de arquivos tem crescido exponencialmente. Esse novo sistema de arquivos tem como requisitos uma maior confiabilidade e a capacidade de mauseio eficente grandes quantidades de dados (na ordem de terabytes). O *Google File System* [Ghemawat et al., 2003] propõe um novo sistema de arquivos que atende a esses requisitos. O Google File System (GoogleFS) foi desenvolvido a partir de observações na carga e necessidades de aplicações específicas da Google Incorporated.

2.5.1 Arquitetura

O GoogleFS foi desenvolvido com base nas seguintes premissas: falhas nos componentes devem ser tratadas como usuais e não como exceções; arquivos serão maiores que a média, em torno de algumas dezenas de MBs; a maioria dos arquivos serão modificados através da adição de novos dados e não da alteração dos dados; e os mesmos dados poderão ser acessados simultaneamente por vários clientes. Assim, o GoogleFS implementa uma arquitetura composta de um servidor mestre (*single master*) e vários escravos (*chunckservers*). Cada arquivo é dividido em *chunks* (ou pedaços) de tamanho fixo, sendo que cada pedaço possui uma identidade única (*unique 64 bit ID*) que é atribuída pelo *single master*. O servidor mestre administra todas as informações do sistema, *namespaces*, controle de acesso, localização dos *chunks*, etc.

O *single master* possui um papel muito importante, já que ele é o responsável por garantir um método sofisticado de alocação e replicação de *chunks* nos diversos *chunkservers*. No entanto, o acesso aos arquivos pelos clientes é feito diretamente através dos *chunkservers*. Isto é, o cliente pergunta ao mestre onde estão os determinados arquivos, que informa quais *chunkservers* os possuem. Assim, o cliente solicita os arquivos diretamente aos *chunkservers*. A figura 2.4 ilustra a arquitetura do GoogleFS e como é feito o acesso aos dados.

2.5.2 Espaço de Nomes

O *singlemaster* é responsável por controlar todo o espaço de nomes do GoogleFS. Isto é, todas as operações que envolvem acesso ao espaço de nomes, como localização, leitura e escrita, passam impreterivelmente pelo *singlemaster*. Ao contrário de outros sistemas de arquivos, o GoogleFS não armazena informações por diretório, ou seja, cada diretório não contém informação sobre quais arquivos e/ou sub-diretórios estão nele contidos. O GoogleFS mantém uma tabela de procura que contém toda a informação do espaço de nomes. Esta tabela de procura é um mapeamento entre caminhos completos e metadados; e cada entrada nessa tabela possui informação de bloqueio/travamento (*lock*) para escrita e leitura.

Quando se deseja escrever em um arquivo chamado `foo.txt` que está no diretório `/dir1/foo/bar/foo.txt`, o *singlemaster* adquire *locks* em `/dir1`, `/dir1/foo`, `/dir1/foo/bar` e um *lock* de escrita em `/dir1/foo/bar/foo.txt`. Desta forma é

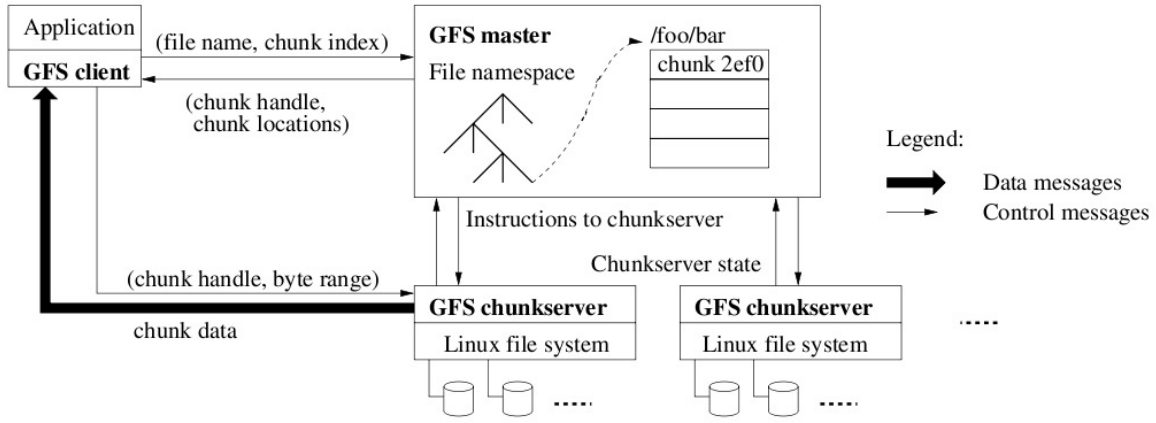

Figura 2.4: Arquitetura e forma de operação do Google File System (extraído de [Ghemawat et al., 2003])

possível executar mudanças simultâneas no mesmo diretório, pois *locks* diferentes serão usados.

Portanto, o GoogleFS apresenta um espaço de nomes transparente e independente, porque o *singlemaster* controla todo o processo e fornece um mesmo modo de operação para todos os clientes.

2.5.3 Tolerância a Faltas

Uma das premissas do GoogleFS é que faltas são comuns ao sistema e ocorrem com extrema freqüência. Ele implementa duas técnicas simples que são utilizadas para garantir um alto grau de disponibilidade: replicação e recuperação rápida.

Tanto os *chunkservers* como o *singlemaster* foram criados de forma a suportar uma recuperação rápida e independente da falha que aconteceu. Não existe diferença em um desligamento normal e um anormal. Quando um destes elementos volta a funcionar, ele está apto para entrar em operação após alguns segundos.

Técnicas de replicação são usadas nos *chunkservers* e no *singlemaster*. Quando um arquivo é inserido no GoogleFS ele é dividido em vários pedaços, e estes pedaços são copiados algumas vezes (normalmente 3) e enviados a diversos *chunkservers*. Desta forma garante-se uma alta disponibilidade, pois mais cópias dos pedaços dos arquivos existem e estas cópias estão espalhadas pelo sistema todo. Como o ponto central do sistema é o *singlemaster*, ele necessita também de um certo grau de replicação. Todas as informações, como tabelas e logs, são replicadas para diversas máquinas, chamadas de “sombras do master". Estas sombras fornecem um acesso somente de leitura às suas estruturas de dados, desta forma caso um *master* fique indisponível, o acesso é garantido por aquelas até que um novo *master* seja iniciado e restabeleça o serviço completamente. Operações que modificam as estruturas do *master* são consideras completas somente após a sua replicação para todos os “sombras".

2.5.4 Semântica de Consistência

O GoogleFS fornece um suporte confiável para garantir a consistência dos dados. *Locks* são suportados em todo o espaço de nomes. *Locks* podem ser de leitura ou de escrita, garantindo operações simultâneas de leitura/escrita em diretórios e arquivos. Além disso, como o GoogleFs propõe-se a trabalhar com hardware de baixo custo, falhas nos discos rígidos são comuns. Então uma verificação de *checksum* para cada pedaço dos arquivos é usada. Para cada pedaço é calculado um *checksum* de 32 bits. Isto garante que pedaços corrompidos por falhas de hardware não serão enviados aos clientes.

2.6 Self-certifying File System (SFS)

O *Self-certifying File System* (SFS) [Mazières et al., 1999] é um sistema de arquivos distribuídos seguro, criado para as necessidades atuais na Internet. O SFS fornece uma solução para as vulnerabilidades encontradas hoje em sistemas distribuídos de compartilhamento de informações. A proteção para essas vulnerabilidades é feita de uma forma diferente no SFS.

Os sistemas de arquivos atuais utilizam chaves para proporcionar um maior grau de segurança, desta forma todo o pesado processamento das chaves é feito pelos sistemas de arquivos. O SFS gerencia as chaves de maneira diferenciada e não no contexto do sistema de arquivos. Ele introduz o conceito de nomes e caminhos assinados, ou seja, no caminho do arquivo ou diretório as chaves públicas estão presentes. Assim, todo o processamento das chaves pode ser feito em uma camada acima do sistemas de arquivos.

O objetivo do SFS é criar um sistema de arquivos único ao redor do mundo, não importando de qual cliente é originada a solicitação de acesso, mas sim do seu usuário. Desta forma, o SFS utiliza intensamente de chaves de criptografia para aumentar o grau de segurança do sistema.

2.6.1 Arquitetura

O SFS tem por objetivo criar um sistema de arquivos altamente seguro, portanto ele divide a segurança do sistema em duas áreas: segurança do sistema de arquivos e gerenciamento de chaves. O SFS garante e oferece somente a segurança do sistema de arquivos, assegurando que o conteúdo do sistema de arquivos não será lido ou modificado por um intruso. Como SFS usa a criptografia para controlar o acesso aos seus arquivos, usuários não podem ler, modificar, deletar ou escrever no sistema sem ter passado por todo o processo de autenticação e deter um segredo. Já o gerenciamento de chaves é feito na camada acima ao sistema de arquivos por agentes escolhidos pelo usuário. O SFS assume que esses agentes são confiáveis.

A figura 2.5 ilustra os componentes do SFS: *SFS Client*, *SFS Server*, *Agent* e *Auth-Server*. Resumidamente, a arquitetura do SFS é composta por cliente e servidores que se comunicam por conexões TCP. O SFS oferece uma semântica similar ao NFS, desta forma o usuário executa chamadas de sistema que atingem o *NFS Client*, e este as redireciona para o *SFS Client*. Todas as mensagens trocadas entre os clientes e servidores são efetuadas através de chamadas RPC.

O *SFS Client* é responsável por toda operação realizada na máquina no usuário, fazendo a intermediação entre o *NFS Client, SFS Server e Agent*. O *SFS Server* é responsável

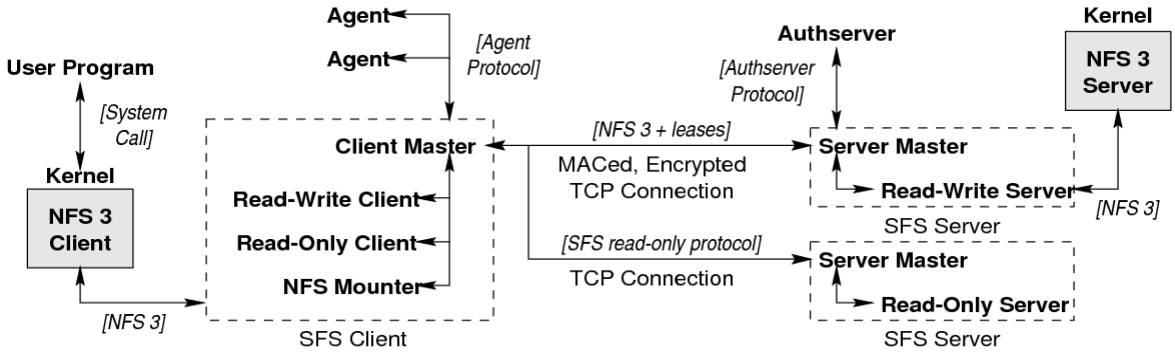

Figura 2.5: Componentes do SFS (extraído de [Mazières et al., 1999])

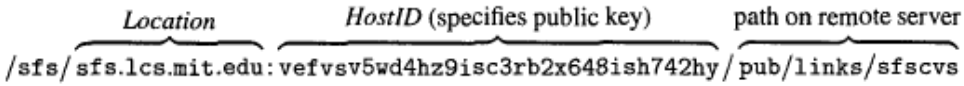

Figura 2.6: Caminho e nomes no SFS (extraído de [Mazières et al., 1999])

por armazenar os arquivos e controlar o acesso aos arquivos. O *Agent* é responsável por implementar um método de gerenciamento de chaves, juntamente com o *AuthServer* que provê métodos para a autenticação dos usuários. O *AuthServer* possui um mapeamento entre chaves e credenciais de usuários. Assim, quando um determinado usuário deseja acessar um arquivo, o *Agent* se comunica com o *AuthServer* para validar o usuário e, após o término da autenticação, o usuário recebe as suas credenciais de acesso.

2.6.2 Espaço de Nomes

O espaço de nomes oferecido pelo SFS é único, porque junto com o caminho e nome do arquivo estão o nome e o endereço do servidor que hospeda o arquivo. A figura 2.6 mostra um exemplo, nela o caminho é dividido em três partes: local, identificação do servidor (*hostID*) e o caminho completo no servidor. O *HostID* é a identificação do servidor representado por sua chave pública, desta forma somente clientes autenticados e certificados pelo servidor poderão acessar os aquivos contidos nele. Portanto, caso um cliente deseje acessar um determinado arquivo ele deverá especificar o caminho completo: localização do servidor (no exemplo da figura usa-se a nomenclatura do DNS), o *hostID* e o caminho do arquivo no servidor nos moldes dos sistemas de arquivos comuns.

O SFS oferece um espaço de nomes transparente. Isto é, como a identificação (*hostID*) do servidor faz parte do caminho, o usuário identifica onde o arquivo se encontra e o ponto de montagem é sempre o mesmo, o que permite que dois arquivos iguais possam existir em servidores diferentes sem que exista confusão por parte do usuário.

2.6.3 Tolerância a Faltas

O SFS segue o modelo cliente-servidor, e portanto depende da disponibilidade da rede e dos servidores para garantir a funcionalidade do sistema de arquivos. Ele não tolera faltas de parada, pois se um servidor cair, os arquivos lá armazenados não estarão mais disponíveis na

rede, no entanto assim que o servidor voltar ao seu estado de funcionamento os arquivos estarão disponíveis.

Contudo, o SFS usa técnicas de criptografia e gerenciamento de chaves para codificar os dados e controlar o acesso aos dados. Em outras palavras, somente após de um processo de autenticação e autorização será permitido o acesso aos dados que estão criptografados.

2.6.4 Semântica de Consistência

Como o foco do SFS é a segurança, ele garante que sempre que algum arquivo for modificado, ele estará visível e válido na próxima operação, porque toda a operação é feita com criptografia e através de canais seguros de comunicação. No entanto, a partir da referência bibliográfica [Mazières et al., 1999] não foi nos possível descobrir se o SFS suporta uma garantia de consistência dos dados para múltiplos acessos de escrita.

2.7 Cooperative File System (CFS)

O *Cooperative File System* (CFS) [Dabek et al., 2001] é um sistema de arquivos distribuídos baseado no modelo *peer-to-peer*. Ele é um sistema somente-leitura (*read-only*), pois somente quem publica os dados pode alterá-los. O CFS oferece garantias para a criação de um sistema de armazenamento de dados eficiente, robusto e com distribuição de carga, através de uma arquitetura totalmente descentralizada que possibilita uma alta escalabilidade.

O CFS usa as tabelas de hash distribuídas (*Distributed Hash Tables* - DHT) providas pelo Chord [Stoica et al., 2001] e técnicas de criptografia para criar um sistema de arquivos distribuído de alta escalabilidade. O Chord é um subtrato *peer-to-peer* que permite a comunicação entre diversos *peers* de forma robusta e permite a localização de dados em qualquer *peer* na rede através de tabelas de hash distribuídas. Outra característica do CFS é que os arquivos são particionados em blocos, e esses blocos são armazenados e distribuídos entre os participantes da rede *peer-to-peer*.

2.7.1 Arquitetura

O CFS contém duas entidades principais na sua arquitetura: o *CFS Client* e o *CFS Server*. Cada *CFS Client* contém três camadas de software: um cliente para o sistema de arquivos (FS), uma camada de armazenamento usando tabelas de hash distribuídas (DHash) e uma camada de busca (Chord). A camada cliente usa a camada logo abaixo, DHash, para adquirir blocos de arquivos. Então a camada DHash usa o sistema de busca oferecido pela próxima camada, Chord, para identificar quem possui os blocos de arquivos desejados.

O *CFS Server* possui duas camadas de software, uma camada de armazenamento usando tabelas de hash distribuídas (DHash) e uma camada de busca (Chord). A camada DHash é responsável por armazenar os blocos de dados dos arquivos, manter um nível adequado de replicação dos dados, e gerenciar um sistema de cache para arquivos mais populares. A camada Chord é responsável por executar as solicitações de buscas dos clientes, verificando no armazenamento local e no cache. A figura 2.7 ilustra essa arquitetura.

Com essa arquitetura, o CFS possui as seguintes propriedade sistêmicas:

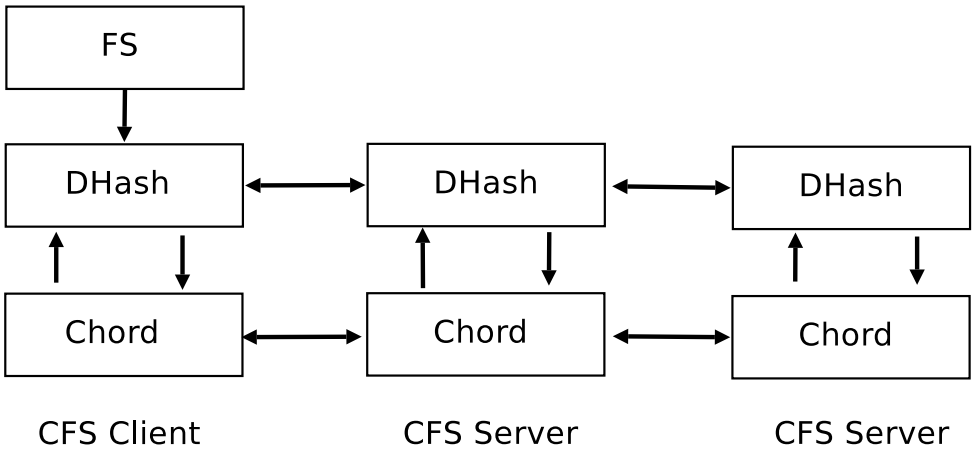

Figura 2.7: Arquitetura do CFS (extraído de [Dabek et al., 2001])

1. **Controle Descentralizado**, porque não exige um controle de configuração e acesso externo. O CFS segue o modelo *peer-to-peer* puro.

2. **Escalabilidade**, pois o algoritmo de busca implementado pelo Chord é eficiente, usa $O(logN)$ mensagens para executar uma busca completa (N é o número de nós na rede).

3. **Disponibilidade**, por existir um modelo de replicação de todos os dados. Assim clientes terão acesso aos dados mesmo na presença de partições de rede ou *peers* faltosos.

4. **Distribuição de Carga**, porque o sistema garante que o trabalho de armazenamento e busca dos dados é igualmente distribuído entre todos os nós.

5. **Persistência**, porque os dados ficarão disponíveis por um tempo limitado, mas acordado no inicio da operação.

6. **Quotas**, já que o CFS limita o número de dados que podem ser armazenados no sistema para cada endereço IP.

7. **Eficiência**, pois clientes adquirem arquivos na mesma velocidade que adquiririam em servidores comuns de FTP.

2.7.2 Espaço de Nomes

O espaço de nomes do CFS é único, já que é responsabilidade do cliente interpretar as informações contidas nas tabelas de hash distribuídas e transformá-las em um formato de sistema de arquivo para o usuário final. Assim, todos os clientes usam o mesmo algoritmo para realizar essa transformação, garantindo que o mesmo espaço de nomes é apresentado aos diversos usuários.

O *CFS Client* usa um modelo similar ao UNIX v7, mas ao invés de usar blocos e endereços do disco para representar o arquivo, usa blocos do DHash e identificadores dos blocos. Cada bloco é um pedaço de um arquivo ou um metadado (como por exemplo um diretório). O sistema executa uma função de hash no conteúdo do bloco para gerar o identificador. Então

o editor assina o bloco raiz do sistema de arquivos com a sua chave privada e gera o identificador da raiz com a sua chave pública. Portanto, o CFS oferece um espaço de nomes único, transparente e independente para os clientes.

2.7.3 Tolerância a Faltas

Como o CFS implementa um sistema de arquivos distribuídos através do modelo *peer-to-peer*, um mínimo de tolerância a faltas já é garantido pelo modelo. Além disso, o CFS possui algoritmos de replicação e controle de cache que aumentam essa tolerância. Os blocos dos arquivos são replicados para *n* servidores e o sistema mantém essas *n* réplicas ativas de acordo com a entrada de saída de nós na rede. O cache é utilizado com um objetivo apenas: controle de carga. Desta forma, os nós que possuem arquivos populares não são penalizados com mais trabalho, já que estes arquivos estão no cache dos outros nós.

No entanto o CFS não tolera faltas maliciosas. Estas faltas podem ocorrer em duas camadas: DHash e Chord. Um servidor malicioso pode publicar dados inválidos no sistema e fazer com que alguns outros dados sumam através da modificação das suas tabelas de hash, modificando assim o caminho onde os dados seriam encontrados.

2.7.4 Semântica de Consistência

O CFS oferece uma semântica similar ao UNIX e ao NFS. Também usa chaves e criptografia para identificar blocos e servidores. Desta forma o CFS garante que os dados estarão corretos mesmo na presença de faltas de parada. Outro fator importante é que todo o cache do sistema opera de acordo com o modelo de uso menos recente. Assim, os dados apresentados poderão ser antigos, mas serão consistentes. O *CFS Client* ainda possui uma opção para executar uma operação de limpeza e atualização das entradas de cache.

2.8 Ivy

Ivy [Muthitacharoen et al., 2002] é um sistema de arquivos baseado em um modelo *peer-to-peer* puro, que oferece suporte a múltiplos usuários e proteção de integridade dos dados. A base do Ivy são os *logs*, sendo que cada participante da rede possui um. Os logs são armazenados em tabelas de hash distribuídas – *distributed hash tables* ou *DHT*. Cada participante consulta todos os *logs* para obter alguma informação, mas adiciona informações somente ao seu próprio *log*. O sistema ainda fornece um controle de versões, para possibilitar a resolução de conflitos. Além disso, o sistema Ivy fornece uma interface com semântica similar ao NFS. Testes mostraram que o Ivy é três vezes mais lento que o NFS.

2.8.1 Arquitetura

Quando se deseja criar um sistema de arquivos em uma rede *peer-to-peer* enfrenta-se os seguintes desafios:

1. Acessos de escrita múltiplos e distribuídos tornam a manutenção consistente dos metadados do sistema de arquivos complexa;

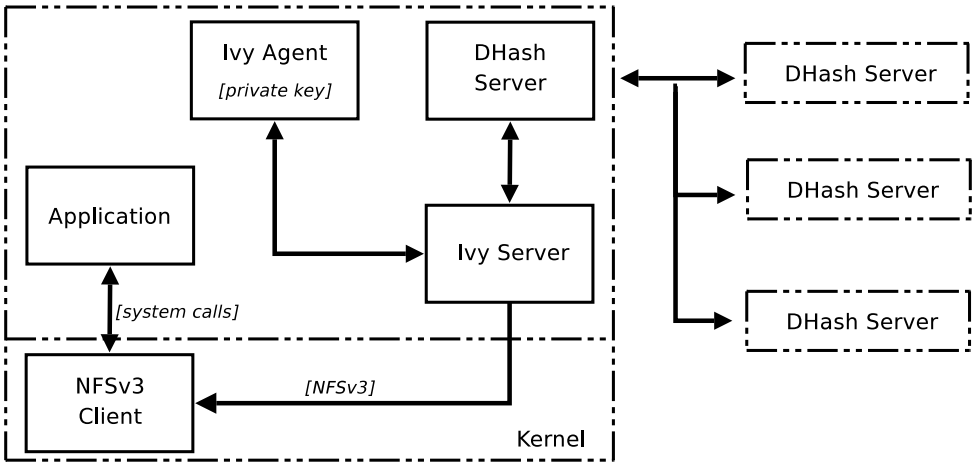

Figura 2.8: Arquitetura do Ivy (extraído de [Muthitacharoen et al., 2002])

2. Um grande número de participantes espalhados pela rede fazem com que técnicas de *locking* não sejam uma boa escolha para assegurar a consistência dos dados;

3. Nem todos os participantes são confiáveis, eles podem ter sido corrompidos. Desta forma é necessário criar uma maneira de reverter a ação destes participantes faltosos;

4. Como os dados estão distribuídos entre os diversos participantes, o sistema está particionado. Assim, deve-se oferecer uma forma de resolver problemas inerentes ao particionamento dos dados e da rede.

Todos esses problemas são solucionados pelo Ivy através de seus *logs* e suas *distributed hash tables*. Além disso, alguma proteção contra ataques maliciosos externos é oferecida através da utilização de criptografia.

O sistema Ivy oferece uma interface similar ao NFS, com suporte a consistência. A figura 2.8 ilustra a arquitetura do Ivy. O *Ivy Server* é responsável por controlar o sistema de arquivos montado na máquina local e interceptar as chamadas de sistema dos clientes. O *DHash Server* é responsável por controlar e localizar todos os arquivos no sistema através das suas tabelas de hash distribuídas. Assim, o *Ivy Server* age como um cliente para o *DHash Server*, solicitando as informações dos metadados e dados. Existe também um agente que é responsável por controlar a chave privada de cada *Ivy Server*, chamado de *Ivy Agent*. Esse agente é chamado antes de qualquer operação no *log*, pois toda entrada no *log* deve ser assinada.

2.8.2 Espaço de Nomes

O sistema Ivy oferece aos seus participantes um espaço de nomes único para cada sistema de arquivos que for criado. Quando o Ivy cria um sistema de arquivos, a chave é usada para identificar esse conjunto de arquivos e suas respectivas tabelas de hash distribuídas, como por exemplo `/ivy/9RYBbWyeDVEQnxeL95LG5jJjwa4`. Então, o *Ivy Server* encarrega-se de criar a raiz do sistema e iniciar a população dos *logs*.

Portanto, o Ivy oferece uma independência de local, pois o sistema de arquivos será montado sempre no mesmo ponto, e os caminhos para os seus arquivos serão sempre os mes-

mos. Há também uma transparência de localização pois está baseado em uma rede *peer-to-peer*, em que os arquivos estão distribuídos na rede e somente o sistema sabe como encontrá-los.

2.8.3 Tolerância a Faltas

Os conceitos de visão (ou *view*) e *snapshot* são implementados pelo Ivy para possibilitar uma certa tolerância a faltas de parada e maliciosas. Cada participante mantém um *snapshot* do estado atual de todos os logs do sistema. Esse *snapsho*t é feito a cada 200ms. Desta forma, caso alguns participantes saiam do sistema, ainda é possível operar em arquivos modificados por este participante. Para o Ivy tudo é gravado nos logs e nas tabelas de hash distribuídas, inclusive os dados dos arquivos.

O conceito de *view* é utilizado pelo Ivy para suportar múltiplos acessos ao mesmo sistema de arquivos. Os participantes do mesmo sistema de arquivos entram em um acordo de quais *logs* farão parte do sistema que eles irão utilizar. O mesmo conceito é utilizado para dar uma tolerância mínima a faltas maliciosas. Quando um participante entra em um estado de falta maliciosa, ele é detectado pelas inconsistências nas operações dos logs que usam chaves criptográficas. Portanto, os outros participantes podem criar uma nova *view* para o sistema de arquivos que exclua os *logs* desse participante malicioso.

2.8.4 Semântica de Consistência

O Ivy oferece uma semântica de consistência similar ao NFS. Após qualquer operação de atualização no sistema de arquivos, toda informação já está disponível para os outros participantes que queiram consultar os mesmos dados. Esta disponibilidade rápida das informações é obtida no Ivy devido a dois pontos:

1. Antes de responder à solicitação do cliente, Ivy espera que toda a informação esteja disponível nas suas tabelas de hash distribuídas, e que a nova entrada do *log* esteja pronta e verificada;

2. Logo no início de qualquer operação, o *Ivy Server* solicita aos *DHash Servers* as entradas mais recentes nos logs. Além disso, estes *logs* são armazenados em cache, mas nunca precisam ser invalidados pois suas entradas são imutáveis.

Para operações de leitura e escrita em arquivos, o Ivy oferece uma consistência *close-to-open*. Quando um cliente escreve novos dados em um arquivo `arq1` e logo após fecha o arquivo, o próximo cliente que ler `arq1` receberá todos os novos dados inseridos pelo primeiro cliente. Em outras palavras, as atualizações dos arquivos são disponibilizadas para os demais clientes somente após uma solicitação de fechamento do arquivo. Caso dois clientes desejem escrever no mesmo arquivo, eles gerarão entradas conflitantes nos logs. Desta forma o Ivy oferece ferramentas para a resolução de entradas conflitantes usando o vetor de versões contido em todas as entradas dos logs.

Um ponto em que o Ivy difere dos sistemas de arquivos distribuídos baseados no modelo *peer-to-peer*, é o modo de gerenciar modificações e criações de diretórios. O *Ivy Server*, implementa um algoritmo de “Exclusive Create"ou criação exclusiva para diretórios e arquivos. Desta forma, ele garante a consistência da unicidade da criação de diretórios e arquivos. No entanto, este algoritmo só funciona se a conectividade entre todos os participantes estiver garantida pela rede; qualquer operação de criação desconectada não é assegurada pelo Ivy.

2.9 Eliot

Eliot [Stein et al., 2002] é um sistema de arquivos mutável (isto é, permite leitura e escrita) baseado em um substrato *peer-to-peer*. Nessas redes existem nós constantemente falhando, outros deixando a rede e ainda outros entrando na rede (tal fenômeno é chamado de *churn*). A base para o seu funcionamento é um sistema de endereçamento adequado, só depois as aplicações poderão ser construídas e desfrutar das vantagens das redes *peer-to-peer*. Na maioria dos casos são utilizadas tabelas de hash distribuídas (DHT) para compor esse endereçamento. A utilização destas DHTs traz algumas vantagens:

1. Pelo mapeamento de dados e nós dentro do mesmo espaço, estes dados podem ser distribuídos entre os nós. Além disso a proximidade dos dados na DHT não significa proximidade física, então potenciais problemas de certas localidades não influenciarão o sistema;

2. Como os valores usados como chave possuem um baixo índice de colisão, com uma alta probabilidade um mesmo endereço não será mapeado para dois recursos diferentes.

2.9.1 Arquitetura

Tendo como base o endereçamento via DHTs, Eliot implementa um sistema de arquivos mutável em uma estrutura *peer-to-peer* imutável. O sistemas é composto por:

- Um substrato confiável de armazenamento *peer-to-peer* chamado *Charles*;
- Uma base de dados replicada e confiável chamada *Metadata Service* (MS);
- Um conjunto de clientes;
- Um ou mais Servidores de Cache para melhorar o desempenho.

Charles é o componente do sistema que provê uma estrutura *peer-to-peer* confiável e tolerante a faltas. A sua entidade principal é o *Charles Block Service* (CBS) que é responsável pelo endereçamento, busca e controle dos nós e dados na rede *peer-to-peer* do Charles. O *Metadata Service* (MS) controla todos os meta-dados do sistema de arquivos, como diretórios, nomes dos arquivos e outros. Também controla onde estão armazenados os blocos e réplicas dos dados.

Assim, quando um cliente deseja ler ou escrever em um arquivo, ele solicita ao MS informações do meta-dados e localização dos dados. Com a resposta do MS, ele então começa a procura dos dados no substrato *peer-to-peer* (*Charles*). Resumindo, os meta-dados e informações sobre os blocos de dados são armazenados no MS e os dados no *Charles*.

O Eliot também implementa um método mínimo de autenticação, que faz com que o MS e os clientes Eliot sejam autenticados entre si. Cada cliente possui uma chave pública registrada no MS, assim cada acesso ao MS é assegurado pela verificação da assinatura de cada cliente. Eliot também oferece uma ferramenta de *cooperative caching*, cuja solução é a utilização de alguns nós como servidor de cache para os arquivos mais solicitados. No quesito implementação, Eliot oferece uma interface POSIX para as aplicações.

2.9.2 Espaço de Nomes

O sistema de arquivos de Eliot é montado localmente pelos clientes, e possui um estrutura de diretórios hierarquizada. Isto faz com que o espaço de nomes do cliente seja mantido. Portanto, Eliot oferece independência de local, porque os dados estão armazenados em uma estrutura *peer-to-peer*. Mas não oferece uma transparência de localização, pois cada cliente poderá escolher o ponto de montagem para o sistema de arquivos.

2.9.3 Tolerância a Faltas

Eliot depende inteiramente da disponibilidade do serviço *Charles*. Assim, *Charles* implementa um suporte a tolerância a faltas de parada e maliciosas, garantindo que se algum dado for mudado maliciosamente ele será detectado. O algoritmo que o Eliot usa para tolerar faltas maliciosas é o seguinte: a modificação maliciosa é detectada através da verificação do hash do endereço do servidor e do dado. Quando um servidor está em um estado de falta maliciosa, o processo de autenticação não ocorre com sucesso e assim o hash do endereço do servidor não será igual ao hash do endereço original e correto. Além disso, se o MS sair do ar, Eliot não poderá modificar os meta-dados, por isso o MS possui redundância e replicação.

2.9.4 Semântica de Consistência

A semântica de consistência oferecida por Eliot depende de como são implementados os clientes. Eliot é capaz de fornecer semântica similar ao NFS e ao AFS. No caso de um cliente NFS, Eliot suportará concorrência em nível de blocos. No caso de um cliente AFS, o Eliot oferecerá uma consistência *close-to-open*. Mais detalhes sobre as semânticas de consistência do NFS e do AFS podem ser vistas nas seções 2.2 e 2.3, respectivamente.

2.10 OceanStore

Com a popularização da Internet e a criação de dispositivos pequenos e portáteis que possuem acesso à Internet, como por exemplo PDAs e aparelhos celulares, cria-se a demanda para a computação ubíqua. Isto é, o usuário tem acesso a toda informação que ele precisa independente da localização e de como ele acessa (computador, telefone celular, ...). Seguindo essa tendência, Kubiatowicz et al idealizaram em [Kubiatowicz et al., 2000] uma infra-estrutura capaz de fornecer aplicações para a computação ubíqua chamada *OceanStore*.

OceanStore não é propriamente um sistema de arquivos distribuídos, mas sim uma estrutura global que oferece acesso contínuo às informações persistentes. Contudo, seus conceitos permitem que seja criada um sistema de arquivos distribuídos usando essa infra-estrutura.

O sistema *OceanStore* foi criado com dois objetivos bem claros:

1. Ser desenvolvido em uma infra-estrutura não confiável. O OceanStore assume que a rede de comunicação e seus elementos falhem frequentemente. Portanto, o sistema deve gerenciar corretamente e de maneira usual essas situações de falha;

2. Ter suporte para que dados possam ser migrados facilmente entre os participantes da rede, esses dados são chamados *nomadic data*, ou dados nômades.

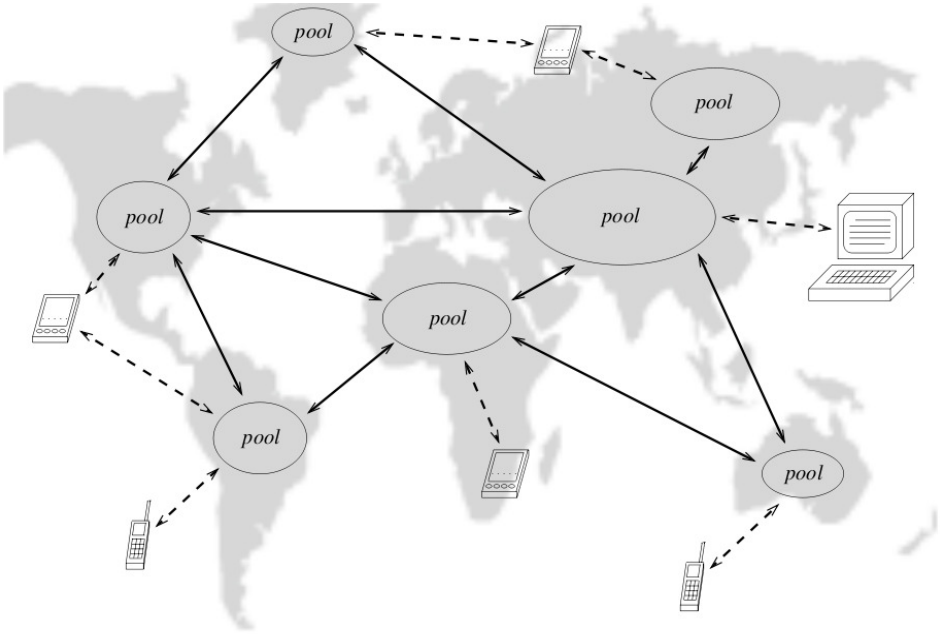

Figura 2.9: Modelo do OceanStore (extraído de [Kubiatowicz et al., 2000])

2.10.1 Arquitetura

A figura 2.9 ilustra a infra-estrutura proposta pelo OceanStore. O OceanStore é composto por *pools* de servidores localizados geograficamente próximos. Os dados passam de um *pool* para outro de maneira simples e livre, e os clientes se conectam a esses *pools* independentemente.

A base para o funcionamento do OceanStore é o objeto persistente (ou *persistent object*), que nada mais é do que um tipo de informação, pode ser por exemplo um arquivo, uma mensagem de e-mail, etc. Cada objeto persistente possui uma identificação única chamada de GUID, *global unique identifier*. Desta forma, estes objetos são replicados e armazenados em diversos servidores, para garantir uma alta disponibilidade dos dados. Ainda, estas réplicas são independentes do servidor onde estão armazenadas, criando-se assim um conceito de *floating replicas*. Isto é, as réplicas também podem ser movimentar entre os diversos servidores do OceanStore.

As réplicas e os objetos são localizados dentro da rede de duas formas. Primeiramente, usa-se um algoritmo probabilístico para tentar localizar o objeto (ou suas réplicas) em servidores geograficamente mais próximos. Caso a busca não tenha sucesso, parte-se para a segunda tentativa de busca, mas agora com um algoritmo determinístico e conseqüentemente mais lento. Este algoritmo é capaz de procurar em todas as máquinas da rede, de forma que, se o objeto existe, ele será encontrado.

Os objetos no OceanStore são atualizados através de operações de *update*, que contêm informações das mudanças ocorridas nos objetos. Para cada mudança, o OceanStore cria uma nova versão do objeto, garantindo assim uma recuperação rápida caso uma operação de *update* seja equivocada. Além disso, o OceanStore separa os objetos em dois tipos: *active* (ou ativos) e *archival* (ou arquivados). Objetos ativos nada mais são do que a última versão de cada objeto, a qual pode ser atualizada por uma operação de *update*. Objetos arquivados são as versões antigas e são utilizados somente para leitura ou recuperação.

2.10.2 Espaço de Nomes

O OceanStore oferece um espaço de nomes transparente, independente e único, caso seja implementada uma aplicação de arquivos distribuídos sobre sua infra-estrutura. Como já mencionado, os objetos são identificados unicamente pelo seu GUID, dessa forma todos os usuários terão os mesmos arquivos disponíveis. Além disso, os objetos e suas réplicas podem migrar entre servidores livremente, deixando claro a independência do espaço de nomes.

2.10.3 Tolerância a Faltas

Como o OceanStore tem por objetivo ser um sistema de informação e armazenamento global, tolerância a faltas é uma de suas premissas básicas. Com a extensa utilização da replicação de objetos, e a independência de armazenamento das réplicas, faltas de parada são toleradas facilmente no OceanStore.

Todavia, caso faltas ocorram na camada que executa a localização e roteamento dos objetos e suas réplicas, a disponibilidade do sistema é ameaçada. O OceanStore resolve este problema através da criação de métodos para constantemente verificar os estado dos links e das rotas para as réplicas. Assim o sistema suporta um certo número de links e rotas corrompidas, mas que serão corrigidas rapidamente.

Ainda, o OceanStore oferece uma tolerância a faltas maliciosas através da utilização de algoritmos de consenso bizantinos durante as operações de *update* dos objetos.

2.10.4 Semântica de Consistência

O OceanStore possibilita atualizações simultâneas em seus objetos. O OceanStore implementa atualizações concorrentes baseado no modelo de resolução de conflitos, possibilitando assim uma alternativa ao modelo clássico de *locks*. Seu modelo de resolução de conflitos suporta vários tipos de semântica de consistência, inclusive semânticas de transação que respeitam as propriedades de atomicidade, consistência, durabilidade e isolamento (ACID).

2.11 FARSITE

O FARSITE é um sistema de arquivos distribuídos que funciona logicamente como um servidor central de arquivos, mas que na verdade é composto de inúmeros computadores dispersos em uma rede. Ele foi criado pela *Microsoft Research* com o intuito de elevar a disponibilidade dos dados em um ambiente corporativo ou acadêmico, mas sem a necessidade de grandes investimentos em servidores dedicados, através do uso dos computadores já existentes no ambiente.

Este sistema de arquivos foi concebido para combinar e oferecer as mesmas características de um servidor central de arquivos e de um sistema de arquivos local, conforme descrito por Adya et al em [Adya et al., 2002]. Estas características são:

- Espaço de Nomes compartilhado;
- Acesso transparente e independente de localização;
- Armazenamento confiável;

- Baixo custo;
- Privacidade;
- Tolerância a faltas geograficamente localizadas.

Além disso, FARSITE usa computadores em rede que não são confiáveis o que implica na adoção de técnicas de segurança e tolerância a faltas, como por exemplo, criptografia, replicação e algoritmos de consenso bizantinos.

2.11.1 Arquitetura

Como premissa para o desenvolvimento do FARSITE, assume-se que o sistema será executado em computadores de mesa em grandes corporações ou universidades. Portanto, o sistema deve ter uma escalabilidade na faixa de 10^5 computadores, sendo que nenhum é um servidor dedicado para armazenamento de dados, e que todos estão conectados através de uma rede local com banda larga e baixa latência. Também assume-se que estas máquinas possuem uma disponibilidade relativamente alta, maior que computadores conectados na Internet e menor que servidores dedicados, e que os períodos de *downtime* estão relacionados com falhas permanentes.

Cada nó (ou computador) pertencente ao sistema executa três papéis distintos: é um cliente, é membro de um grupo (*Directory Group*), é um servidor de arquivos (*File Host*). O cliente é uma máquina que interage com o usuário, nos mesmos moldes de um sistema de arquivos local. O Grupo de Diretório é um conjunto de máquinas que gerenciam, coletivamente, todas as informações dos arquivos (metadados) contidos no sistema por meio de um protocolo tolerante a faltas maliciosas. O Servidor de Arquivos contém réplicas dos arquivos existentes, sendo utilizado para armazenamento e recuperação de dados dentro do sistema.

Para ilustrar de forma genérica o funcionamento do FARSITE, considera-se um sistema composto por vários clientes e um grupo de diretório (*Directory Group*), sendo que esse grupo gerencia todas as informações relativas ao sistema de arquivos (dados e meta-dados). O Grupo de Diretório armazena essas informações de forma redundante em cada membro do grupo. O protocolo de replicação utilizado garante a consistência das informações mesmo em situações de falha. Quando um cliente deseja ler um arquivo, ele envia uma requisição ao grupo, que responde com o conteúdo do arquivo solicitado. Se um cliente atualiza ou modifica um arquivo, ele envia uma mensagem de atualização para o grupo. No caso de um segundo cliente solicitar a abertura de um arquivo que já está sendo utilizado pelo primeiro cliente, o grupo verifica a semântica aplicada pelos dois clientes a fim de determinar a concessão ou não do acesso ao segundo cliente.

O FARSITE é implementado para a plataforma Windows e possui dois componentes: um serviço em nível de usuário e um *driver* em nível de *kernel*. O *driver* implementa funções que necessitam interface com o sistema de arquivo local e com o *kernel*, são elas: exportar uma interface adequada do sistema de arquivos, interação com o gerenciador de *cache* e cifrar/decifrar o conteúdo dos arquivos. Já o serviço implementa todas as outras funções contidas no FARSITE: gerenciamento do *cache* de arquivos local, busca de arquivos em máquinas remotas, validação do conteúdo dos arquivos, replicação de arquivos, gerenciamento dos metadados e execução do protocolo de replicação bizantino.

A figura 2.10 ilustra a arquitetura do FARSITE dentro da plataforma Windows. O RDBSS é um *driver* que atua como um arcabouço genérico para a implementação de sistemas

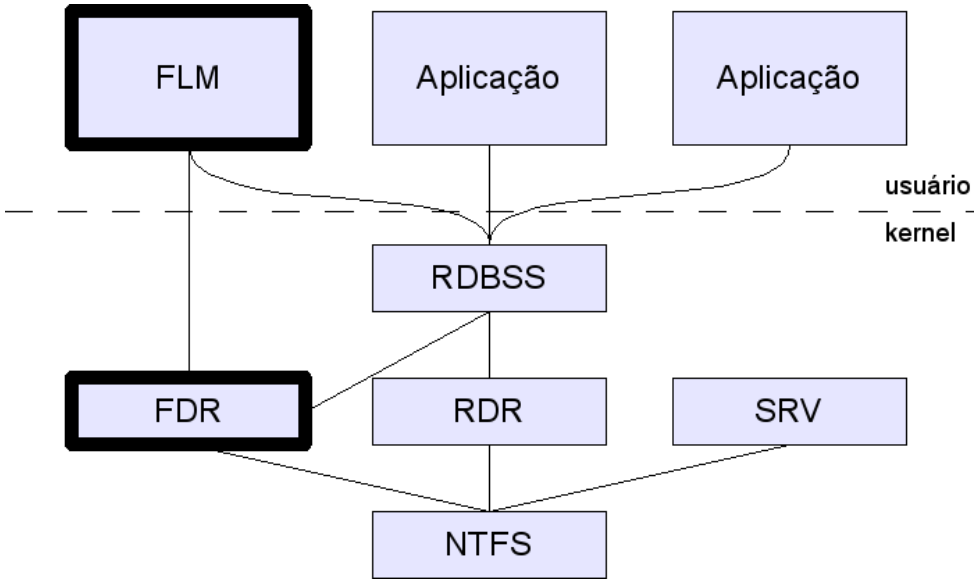

Figura 2.10: Arquitetura do FARSITE (adaptado de [Adya et al., 2002])

de arquivos em rede. O NTFS é o sistema de arquivos local disponível. Já o RDR e o SRV são dois *drivers* nativos responsáveis pela implementação do sistema de arquivos em rede padrão da plataforma Windows, ele implementam as interfaces especificadas pelo RDBSS. A implementação do FARSITE usa este arcabouço já disponível para adicionar seus dois módulos: o serviço, *Farsite Local Manager (FLM)* e o *driver*, *Farsite Mini Redirector (FMR)*. O FMR recebe as chamadas do sistema de arquivos, interage com o FLM, usa o NTFS como repositório local de persistência, e executa as operações de criptografia nas chamadas de E/S durante a chamadas de escrita ou leitura. Já o FLM controla as conexões de controle entre os nós do FARSITE, e estabelece as conexões de acesso remota aos arquivos através dos *drivers* RDR e SRV.

2.11.2 Espaço de Nomes

O Espaço de Nomes no FARSITE é transparente, independente e gerenciado por um ou mais Grupos de Diretório *(Directory Group)*. O Grupo de Diretório é responsável por administrar o espaço de nomes, garantindo que não existam conflitos. Como o sistema foi projetado para suportar um grande número de máquinas, algumas modificações no modo como o espaço de nomes é gerenciado foram feitas, a fim de que problemas de consistência possam ser resolvidos mais rapidamente e que não existam pontos de contenção no seu desempenho.

O Espaço de Nomes pode ser gerenciado por um ou mais Grupos de Diretórios a medida que o sistema cresce. É permitido que um Grupo de Diretório delegue para outro Grupo de Diretório a administração de um subconjunto de nomes dentro do seu espaço de nomes. No entanto, do ponto de vista do cliente nada é alterado, pois os Grupos de Diretórios encaminham mensagens uns para os outros quando estão gerenciando o mesmo espaço de nomes. Outra modificação é na definição da raiz do espaço de nomes, o FARSITE permite que o administrador do sistema crie múltiplas raízes no espaço de nomes, formando assim servidores de arquivos virtuais. Isto torna o sistema mais flexível e tolerante a modificações frequentes no espaço de nomes.

Além disso, o FARSITE implementa listas de controle de acesso (*ACL - Access Control List*) aos arquivos e diretórios e um método para garantir a consistência do espaço de nomes chamado *name leases*. *Name Leases* são permissões para um cliente para um certo nome dentro do espaço de nomes. Caso o nome do diretório ou arquivo não exista, essa permissão possibilita que o cliente crie uma entrada no sistema com esse nome, garantindo que nenhum outro cliente usará o mesmo nome. Caso o nome já exista, a permissão propicia que o cliente crie arquivos ou subdiretórios abaixo do nome contido na permissão.

2.11.3 Tolerância a Faltas

Nesse quesito o FARSITE se destaca em relação aos outros sistemas de arquivos distribuído. Como ele foi projetado para ser executado em máquinas não confiáveis, ele implementa técnicas avançadas para tolerar faltas. Três técnicas principais são utilizadas: criptografia, replicação dos dados e protocolo de consenso bizantino.

Todas as conexões, meta-dados e dados são criptografados através de uma estrutura de chaves públicas. As conexões entre clientes e grupos de diretórios são criptografadas, permitindo assim que somente clientes confiáveis se conectem ao sistema. Todas as operação nos meta-dados e nos dados (arquivos) são também criptografadas, isso permite a implementação da lista de controle de acesso, conforme indicado na seção anterior. Para a criptografia do conteúdo dos arquivos (dados) um método chamado criptografia convergente [Adya et al., 2002] é aplicado. Este método permite que blocos de dados dos arquivos possam ser cifrados a medida em que são escritos no sistema. Possibilitando assim a adição de novos conteúdos sem a necessidade de cifrar o arquivo inteiro a cada nova escrita.

Todos os dados e meta-dados presentes no sistema são replicados para diversas máquinas. Isso garante que, se algumas máquinas sofrerem faltas de parada, os dados ainda estarão disponíveis para os usuários. Os meta-dados são replicados entre máquinas pertencentes ao mesmo grupo de diretórios, e os dados são replicados entre os servidores de arquivos. Estas réplicas são constantemente recolocadas (movidas de uma máquina para outra) com o intuito de aumentar a disponibilidade e diminuir a possibilidade de particionamento de um espaço de nomes.

Para a replicação de meta-dados dentro de um grupo de diretório, um protocolo de replicação tolerante a faltas maliciosas é utilizado. Através do uso deste método, o FARSITE é capaz de tolerar faltas maliciosas (ou maliciosas) provocadas por máquinas ou usuários fraudulentos participantes do sistema. O sistema é robusto o suficiente para manter a disponibilidade de seus serviços para os meta-dados e dados da seguinte forma:

- para um grupo de diretório com R_D membros, os metadados são preservados e acessíveis se até $(R_D - 1)/3$ membros estiverem em um estado faltoso;
- para arquivos replicados em R_F servidores de arquivos, os dados são preservados e acessíveis se pelo menos um servidor estiver ativo e não-faltoso.

2.11.4 Semântica de Consistência

A plataforma Windows suporta consistência provendo controle sobre a semântica de compartilhamento de arquivos para as aplicações. Quando uma aplicação abre um arquivo, ela especifica dois parâmetros: modo de acesso e modo de compartilhamento. Este especifica o

tipo de acesso que será dado a outras aplicações que tentarem acessar o mesmo arquivo, já aquele especifica qual o tipo de acesso solicitado pela aplicação. Do ponto de vista de um sistema de arquivos distribuído, estes tipos de acesso são resumidos em três modos de acesso: acesso de leitura, acesso de escrita e acesso de deleção; e em três modos de compartilhamento: compartilhamento de leitura, compartilhamento de escrita e compartilhamento de deleção.

Para suportar essa semântica provida pela plataforma, o FARSITE implementa seis tipos de permissões (*mode leases*): leitura, escrita, deleção, exclusão de leitura, exclusão de escrita, exclusão de deleção. Quando um cliente abre um arquivo, o FARSITE traduz os modos de acesso e de compartilhamento para os tipos de permissões adequados. O Grupo de Diretório é responsável por verificar se pode emitir as permissões solicitadas sem conflitar com outras permissões existentes para o arquivo em questão. Caso ele não possa emitir as permissões solicitadas, ele entra em contato com o(s) cliente(s) detentores das permissões a fim de verificar se estas podem ser retiradas ou modificadas. Este processo implica em um tempo maior na operação de abertura de arquivos quando existem conflitos de permissões no sistema.

Além disso, a plataforma Windows possui uma semântica complexa para executar a deleção de arquivos, o que forçou o FARSITE a implementar mais tipos de permissões de acesso: acesso público, acesso protegido, e acesso privado. *Acesso Público* indica que o detentor da permissão está com o arquivo aberto, *Acesso Protegido* indica que o arquivo está aberto e que nenhum outro cliente receberá permissões de acesso sem antes contatar o detentor do acesso, e *Acesso Privado* indica que o arquivo está aberto e que ninguém receberá permissão de acesso ao arquivo enquanto o detentor atual não retirar a permissão atual. Desta forma, o Grupo de Diretório sabe que a permissão de acesso o cliente possui, possibilitando assim uma deleção sem conflitos.

2.12 Conclusão do Capítulo

Além dos sistemas de arquivos distribuído apresentados neste capítulo, existem ainda muitos outros. Este campo de estudo e pesquisa é vasto e cheio de desafios e problemas. Satyanarayanan em [Satyanarayanan, 1989] faz uma análise de diversos sistemas de arquivos distribuídos que usam o modelo cliente-servidor como base, como o CODA, Sprite, RFS, entre outros. Zahid também faz uma análise detalhada em [Hasan et al., 2005] de sistemas que usam o modelo *peer-to-peer* como base, como o Kellips e o Freenet.

Esse capítulo descreveu e avaliou diversos pontos de alguns dos sistemas de arquivos distribuídos existentes. A tabela 2.1 compara os principais pontos dos dez sistemas apresentados.

Adicionalmente, a ordem das seções tenta retratar a evolução destes sistemas: observa-se claramente que inicialmente o modelo cliente-servidor era dominante, logo após surgiram os primeiros sistemas usando o modelo *peer-to-peer* e finalmente constata-se a consolidação do modelo *peer-to-peer*. Contudo, percebe-se que existem muitos desafios para tornar os sistemas de arquivos distribuído baseados no modelo *peer-to-peer* realmente disponíveis para todos e tão estáveis e confiáveis como os sistemas que usam modelos cliente-servidor. Um destes desafios é o gerenciamento e controle das redes *peer-to-peer*, já que as funções estão distribuídas pela rede. Outro é o suporte a tolerância a intrusões e segurança, porque no mundo altamente conectado de hoje a probabilidade de intrusões ocorrerem é significativa. E por fim, a disposição do usuário em compartilhar espaço em disco para arquivos desconhecidos de pessoas desconhecidas.

Sistema	**Modelo**	**Espaço de Nomes**	**Tolerância a Faltas**	**Semântica de Consistência**
NFS	cliente-servidor	independente	nenhuma	nenhuma
AFS	cliente-servidor	independente e transparente	parada	por sessão
xFS	*peer-to-peer*	independente e transparente	parada	locks
GoogleFS	*peer-to-peer* híbrido	independente e transparente	parada	locks
SFS	cliente-servidor	transparente	-	-
CFS	*peer-to-peer* puro	independente e transparente	parada	cache, uso menos recente
Ivy	*peer-to-peer* puro	independente e transparente	parada	close-to-open
Eliot	*peer-to-peer* híbrido	independente	parada	close-to-open
OceanStore	cliente-servidor e *peer-to-peer*	independente e transparente	maliciosas	ACID
FARSITE	*peer-to-peer* puro	independente e transparente	arbitrárias e maliciosas	baseado em permissões

Tabela 2.1: Resumo dos Sistemas de Arquivos Distribuídos apresentados

Capítulo 3

O Conceito de Arquivos Transparentes

3.1 Introdução

A palavra "transparência" tem sido usada largamente em sistema computacionais para descrever algumas características do sistema ou algumas de suas funcionalidades. No entanto, existem várias possíveis interpretações para "transparência" dentro da ciência da computação. A primeira segue a interpretação da física, na qual tudo no interior do sistema pode ser visto com precisão e claridade. Uma outra segue o ponto de vista do usuário ou outra entidade que tem interface com o sistema, em que o comportamento ou funcionalidade dele não é percebido pelo usuário ou entidade. Ainda uma outra pode indicar um nível de abstração, cujo funcionamento detalhado de alguns componentes do sistema não têm importância para explicar o funcionamento de outros.

Gorn em [Gorn, 1965] e Parnas em [Parnas and Siewiorek, 1975] mostram o uso do termo "transparência" no contexto da ciência da computação. O primeiro artigo, datado de 1965, explica como usar os caracteres da tabela ASCII de controle para que comunicação entre dois computadores seja transparente, ou seja, possa enviar e receber caracteres não pertencentes à tabela ASCII de forma correta. O segundo, datado de 1975, mostra o conceito de transparência na criação de sistemas organizados hierarquicamente. Isto é, usa transparência como um nível de abstração adicional para a criação de sistemas.

Este capítulo tem como objetivo introduzir o conceito de Arquivos Transparentes dentro do contexto de sistemas de arquivos distribuído, e explicar o funcionamento de *Transparent File System* (TFS), e discorrer sobre trabalhos relacionados que usam o conceito de transparência. O TFS é um sistema de arquivo local que implementa esse conceito de arquivos transparentes para que possam ser aplicados a diversas situações [Cipar et al., 2007].

3.2 Definição de Arquivos Transparentes

Em um sistema de arquivos distribuído, o arquivos (ou dados) são armazenados em diversos computadores pertencentes a uma rede. Isto significa que diversos usuários podem ter seus arquivos armazenados nestes locais comuns que compartilham o espaço livre em disco. Portanto, qualquer usuário tem seu desempenho e espaço livre em disco afetado por arquivos pertencentes a outros usuários, já que o espaço compartilhado é comum a um grupo de usuários. Isto leva à aplicação de algumas políticas, como restrição de uso por quotas, para que esse

espaço compartilhado tenha um efeito restrito no desempenho e disponibilidade de espaço livre no sistema como um todo.

”Arquivos Transparentes” são arquivos de outrem (com origem desconhecida) armazenados em uma determinada máquina de forma a não impor custos adicionais de desempenho e disponibilidade de espaço livre em disco para o usuário ou processos locais. Em outras palavras, os “arquivos transparentes” são armazenados no disco, mas os usuários locais têm a sensação de que eles não existem, pois não causam impacto de desempenho perceptível. Portanto, do ponto de vista dos usuários ou processos locais eles são “transparentes” perante ao sistema local.

3.3 Transparent File System (TFS)

Em [Cipar et al., 2007], Cipar et al introduzem o *Transparent File System* (TFS), que é um sistema de arquivos local com o intuito de disponibilizar todo o espaço livre em disco para outros sistemas não relacionados com o computador local, de uma forma transparente para o usuário e sem penalidade no desempenho local. Assim, o TFS opera de forma que aplicações distribuídas possam compartilhar o espaço livre em disco em cada máquina participante.

O TFS armazena os dados sendo utilizados pelo grupo de compartilhamento no espaço livre do disco, de forma que esses dados e arquivos não sejam visíveis para os processos locais. Assim, ele permite que os processos locais continuem a trabalhar da mesma forma, sem sequer notar a presença de dados e arquivos compartilhados pelo grupo. Portanto, do ponto de vista do usuário local, o seu espaço disponível em disco continua o mesmo, apesar de poder estar sendo utilizado por aplicações distribuídas.

3.3.1 Arquitetura

O desempenho do sistema de arquivos local que usa o TFS é minimamente impactado, pois o aspecto principal de sua arquitetura reside no processo de alocação de blocos. Sabe-se que o processo de alocação de blocos em sistemas de arquivos é fundamental para a obtenção de um bom desempenho do sistema de arquivos.

No TFS existem dois tipos de arquivos: *arquivos transparentes* e *arquivos opacos*. *Arquivos Transparentes* são arquivos existentes no sistema mas invisíveis aos usuários, porque fazem parte de um grupo de recursos compartilhados. *Arquivos Opacos* são os arquivos locais e visíveis aos usuários, ou seja, são os arquivos convencionais. Da mesma forma, os blocos são definidos como transparentes ou opacos. Portanto, o TFS implementa um outro processo de alocação de blocos dentro do sistemas de arquivos, onde existem arquivos e blocos opacos e arquivos e blocos transparentes. As próximas seções explicam em mais detalhes o funcionamento este processo diferenciado de alocação de blocos.

O TFS introduz um compromisso de manter um bom desempenho para os arquivos locais (*arquivos opacos*), e por isso não garante a persistência dos arquivos e blocos transparentes. Isto é, arquivos e blocos opacos têm prioridade em relação a arquivos e blocos transparentes e podem ocupar qualquer espaço do disco a qualquer momento, sobrescrevendo blocos ou arquivos transparentes.

O TFS é implementado como um módulo do *kernel* 2.6.13 do Linux e é baseado no sistema de arquivos Ext2 [Card et al., 1994]. Assim ele oferece todas as facilidades de um arquivo local (operações de leitura, escrita, abertura e fechamento), e adiciona uma chamada de sistema para informar se arquivos ou diretórios são transparentes ou não.

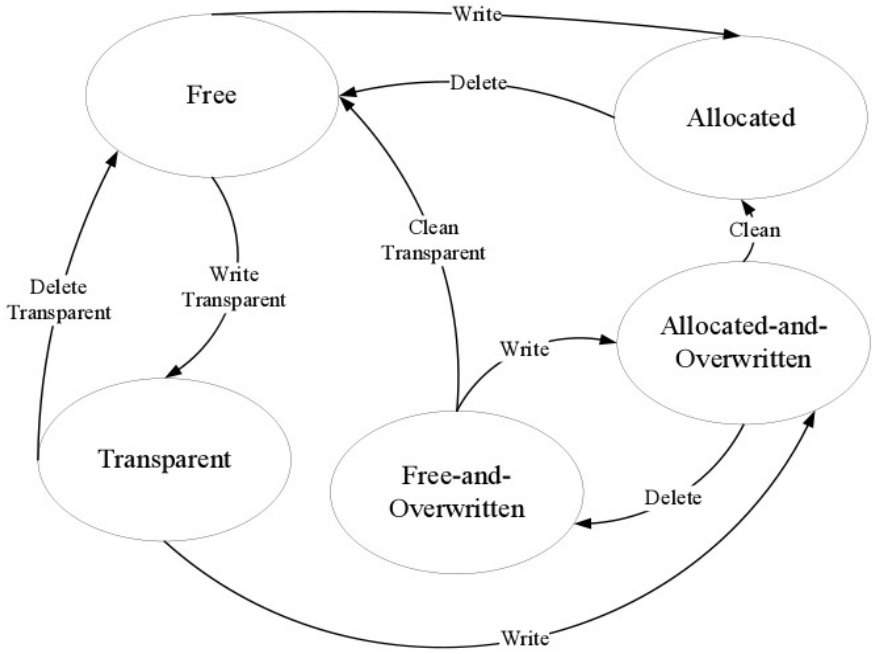

Figura 3.1: Máquina de estados de alocação de blocos no TFS (extraído de [Cipar et al., 2007])

3.3.2 Alocação dos Blocos

O TFS modifica o procedimento de alocação de blocos do EXT2 de forma que blocos transparentes sejam vistos como blocos livres e seu conteúdo possa ser facilmente sobrescrito. Permitindo assim que o processo de alocação de arquivos para arquivos opacos aconteça como se não existissem arquivos e blocos transparentes no sistema.

No TFS, os blocos podem assumir cinco estados, três novos estados adicionados pelo TFS: *Transparent, Free-and-Overwritten e Allocated-and-Overwritten*; e dois existentes no EXT2: *Free* e *Allocated*. O estado *Free* significa que o bloco está livre e pode ser usado por arquivos opacos ou transparentes. Em contraste, o *Allocated* significa que o bloco está sendo usado por um arquivo opaco. Já o estado *Transparent* significa que o bloco está sendo usado por um arquivo transparente. Os estados *Allocated-and-Overwritten* e *Free-and-Overwritten* significam, respectivamente: que o bloco estava sendo usado para um arquivo transparente e foi sobrescrito por dados de um arquivo opaco, e que o bloco foi liberado pelo arquivi opaco mas antes estava sendo usado por um arquivo transparente. Estes dois últimos estados indicam que o bloco foi usado por um arquivo transparente mas foi sobrescrito em algum momento, desta forma o TFS garante que blocos de arquivos opacos não estejam disponíveis para aplicações que usam os arquivos transparentes.

A Figura 3.1 ilustra o diagrama de transição para os vários estados do blocos no TFS. *Arquivos Opacos* podem usar blocos *Free* ou *Transparent* (*Write*), enquanto os arquivos transparentes somente blocos *Free* (*Write Transparent*). *Arquivos Opacos* podem sobrescrever qualquer bloco sendo usado por arquivos transparentes (blocos no estado de *Transparent*), quando isso acontece o bloco passa ao estado de *Allocated-and-Overwritten*. E por fim, quando blocos sobrescritos por arquivos opacos são liberados (*Delete*), eles mudam para o estado de *Free-and-Overwritten*.

3.3.3 Comportamento das Operações de Arquivos

No TFS, algumas das ações realizadas pelas operações usuais sobre arquivos (as chamadas *read*, *close*, *read* e *write*) têm seu comportamento modificado, devido às mudanças nos procedimentos de alocação de blocos.

A operação de escrita em arquivos transparentes somente busca blocos no estado *Free*, já em arquivos opacos busca blocos nos estados *Free*, *Transparent* e *Free-Overwritten*. Assim, se o número de blocos disponíveis (no estado *Free*) acabar, arquivos transparentes não serão mais criados, mas arquivos opacos sim.

A operação de leitura em arquivos opacos busca por blocos nos estados *Allocated* e *Allocated-and-Overwritten*; já em arquivos transparentes somente por blocos no estado *Transparent*. Desta forma, blocos que antes eram usados por arquivos transparentes e foram sobrescritos não terão seu conteúdo acessado de forma errônea.

A operação de abertura para arquivos opacos não muda, mas para arquivos transparentes sim. Quando um arquivo transparente é aberto, o TFS verifica se nenhum bloco foi sobrescrito. Em caso positivo, esta operação retorna erro indicando que o arquivo transparente foi deletado, já que o seu conteúdo não está mais consistente. Então o TFS marca todos os blocos pertencentes ao arquivo transparente recém deletado como *Free* ou *Allocated*, dependendo do estado anterior.

Por fim, a operação de fechamento não foi alterada. Quando o arquivo é fechado, modificações ainda residentes em memória são transferidas para o disco e os campos de metadados do arquivo são atualizados.

3.4 Trabalhos Relacionados

O conceito de transparência vem sendo usado em computação de várias maneiras. Sendo que uma delas é comparável à maneira aqui proposta: o uso de transparência, sem impacto no desempenho do sistema local, para compartilhamento de recursos entre computadores, como espaço em disco e ciclos de processamento.

A capacidade de processamento das CPUs, a velocidade de acesso à memória e o espaço de armazenamento em disco vêm aumentando significativamente nos últimos anos, mas a velocidade de acesso aos discos de armazenamento vem aumentando em um ritmo mais lento. Isto causa uma necessidade de uso de disco grande, mas o seu acesso ainda é mais lento. Desta forma, foram feitos estudos que comprovam um desempenho melhor para certas aplicações através da utilização de técnicas de cache em memória para arquivos no disco [Karedla et al., 1994]. Nestes sistemas, arquivos contidos no disco são trazidos para a memória fazendo com que o acesso da aplicação aos arquivos seja diretamente na memória. Assim, a maior parte do tempo o acesso é feito na memória e posteriormente todas as modificações são passadas para o disco. Patterson em [Patterson et al., 1993] propõe um sistema que conhece *a priori* o padrão de uso de arquivos de certas aplicações e carrega determinados arquivos em memória antes da aplicação solicitá-lo. Portando, através do uso de memória como cache de arquivos em disco aumenta-se o desempenho de certas aplicações e compartilha-se a memória de forma transparente e sem deterioração do sistema local.

Existe também uma outra forma de compartilhamento de memória proposta por Cipar em [Cipar et al., 2006], onde a máquina local controla de forma transparente o uso de memória de algumas aplicações. Estas aplicações normalmente têm o intuito de executar tarefas para

o bem de um grupo, como por exemplo o compartilhamento de arquivos. Através de uma melhoria do sistema de gerenciamento de memória do sistema operacional, é possível fazer com que estas aplicações possam ter uma prioridade menor no uso de memória. Em outras palavras, aplicações mais importantes do sistema local têm preferência ao acesso à memória, em detrimento às outras aplicações, diminuindo assim a taxa de faltas de página *(page faults rate)* para as aplicações locais.

Muitos computadores atualmente estão ligados à Internet e suas CPUs permanecem a maior parte do tempo ociosa, e existem certas tarefas específicas que exigem muito processamento (ou ciclos de CPU). Portanto, é possível usar essas CPUs ociosas para contribuir em computações e cálculos mais complexos e demorados. Isto é exatamente o que é proposto por Anderson em [Anderson et al., 2002] e por Larson em [Larson et al., 2002]. O primeiro propõe um sistema no qual quando a CPU está ociosa ela é usada para ajudar nos cálculos da análise de sinais de rádio vindas de outras galáxias. O sistema é composto por diversos agentes, em um modelo *peer-to-peer*, que fazem cálculos e enviam seus resultados para a rede a fim de que sejam continuados. O segundo usa o tempo ocioso de CPU para simular as reações químicas entre enzimas, a qual se fosse feito por um único computador poderia levar décadas. Portanto, estes dois sistemas compartilham o uso de CPU de forma transparente, e sem impacto para o sistema local já que a CPU será utilizada somente quando está no estado ocioso.

O conceito de transparência em compartilhamento de recursos também é muito utilizada em sistemas de *cluster* de computadores. Os computadores pertencentes ao *cluster* trabalham de forma cooperativa e com distribuição de carga para atingir um resultado específico, que pode ser uma simulação, cálculos astro-físicos, etc. O MOSIX [Barak and La'adan, 1998] é um sistema que provê uma infra-estrutura para montar *clusters*, usando sistema com um ou mais processadores de forma homogênea e como se fossem um único computador. O MOSIX compartilha recursos entre os computadores membros do grupo ou *cluster* de forma transparente e fácil. Isto é feito através da migração, de forma preemptiva, de processos entre os diversos computadores com o objetivo de melhorar o desempenho total do sistema em *cluster*. Portanto, no MOSIX, processos são compartilhados de forma transparente e sem prejuízo ao processamento final do *cluster*.

3.5 Conclusão do Capítulo

Neste capítulo foi definido o conceito de transparência usado amplamente na ciência da computação e o conceito de arquivos transparentes introduzido pelo TFS. O TFS tem como objetivo fomentar o uso do espaço livre em disco para armazenamento de dados de um sistema distribuído com nenhum impacto para o sistema local. Em outras palavras, o TFS minimiza o efeito psicológico dos usuários durante o compartilhamento de espaço em disco, já que os arquivos compartilhados são transparentes e não afetam o desempenho do sistema local, pois sua persistência não é garantida.

No próximo capítulo é proposto um sistema de arquivos distribuído capaz de tolerar, de forma simples, a falta de persistência dos arquivos transparentes. Com isso, os usuários não terão problemas em compartilhar 100% do seu espaço livre em disco em uma rede.

Capítulo 4

Uso de Arquivos Transparentes na Construção de um Sistema de Arquivos Distribuído Tolerante a Faltas

4.1 Introdução

Neste capítulo apresenta-se como o conceito de arquivos transparentes pode ser aplicado para a criação de um sistema de arquivos distribuído tolerante a faltas, no qual os usuários podem compartilhar o espaço em disco disponível, sem a sensação de diminuição do seu espaço livre. Desta forma, pode-se usar o máximo do espaço disponível em disco de diversos usuários para a criação de sistemas de arquivos que melhorem a disponibilidade dos dados, a tolerância a faltas e aumentem o espaço em disco disponível para o grupo.

Inicialmente é apresentada a motivação para a criação de tal sistema e em seguida, a proposta é explicada detalhadamente com uma listagem de seus objetivos específicos. A seguir, a arquitetura da proposta é apresentada, juntamente com os algoritmos para um manuseio adequado dos arquivos transparentes. Por fim, é feita uma discussão sobre três tipos de aplicações que poderiam se beneficiar do sistema proposto, e como as suas arquiteturas seriam impactadas.

4.2 Motivação

Tradicionalmente, sistemas de arquivos distribuídos não diferem em nada de sistemas de arquivos locais do ponto de vista de utilização de espaço e modo de operação local, conforme visto no capítulo 2. Isto é, quando parte do disco é usada para armazenar arquivos locais e arquivos compartilhados, eles são tratados da mesma forma pelo sistema de arquivos. Assim, tanto os arquivos estritamente locais e os compartilhados diminuem o espaço livre disponível, são visíveis aos usuários locais, seu acesso é controlado pelas mesmas regras de controle de acesso, etc.

Com a crescente expansão da conectividade entre computadores através de redes locais ou da Internet, a demanda para criação de aplicações capazes de compartilhar dados e espaço em disco está aumentando, conforme Androutsellis-Theotokis comenta em [Androutsellis-Theotokis and Spinellis, 2004]. Todavia, os usuários de sistemas que compartilham espaço em disco não se sentem à vontade em compartilhar o máximo possível para o

grupo, pois têm a impressão de que outras pessoas estão usando o seu espaço em disco, o qual poderá ser necessário no futuro, e porque também temem uma diminuição no desempenho de seu computador. Portanto, a atitude em relação ao compartilhamento de espaço em disco varia para cada usuário: alguns se sentem ameaçados pela falta de espaço e compartilham o mínimo possível, já outros desejam contribuir mais para o grupo e compartilham o máximo possível [Golle et al., 2001].

Esse problema é mais critico quando a aplicação distribuída precisa garantir a alta disponibilidade dos dados, tolerar faltas e possibilitar um acesso rápido ao dados. Estas aplicações precisam ainda mais de espaço em disco para: armazenar várias cópias da mesma informação, garantindo uma alta disponibilidade e tolerância a faltas; e para armazenar cópias locais em *cache*, diminuindo o tempo de acesso às informações. Consequentemente, as aplicações desse tipo acabam forçando o usuário a compartilhar ainda mais espaço, gerando mais resistência psicológica para a sua utilização.

4.3 A Proposta

A proposta deste trabalho é utilização dos arquivos transparentes apresentado no capítulo 3, construindo a partir deles um modelo de um sistema de arquivos distribuído com alta disponibilidade e tolerância a faltas. No ambiente distribuído proposto, diversos usuários/-computadores compartilham seu espaço livre em disco de forma transparente, sem prejuízo da aplicação local.

O sistema proposto é composto por vários usuários/computadores conectados entre si através de uma rede *peer-to-peer overlay*, onde todos podem usar o espaço livre compartilhado pelo grupo. Dessa forma, arquivos pertencentes a um determinado usuário poderão ser armazenados no disco de qualquer outro usuário/computador participante da rede *overlay*. O espaço disponível total para o grupo é grande e composto pela somatória do espaço compartilhado por cada usuário, entretanto a disponibilidade do espaço livre local não é afetada.

Adicionalmente, o sistema proposto deve garantir a alta disponibilidade dos arquivos nele armazenados e a tolerância a faltas dos seus membros. A tolerância a faltas é importante porque pode ocorrer um particionamento na rede, que deixaria alguns *peers* desconectados, e também porque os membros do sistema podem sair ou entrar na rede a qualquer momento, sem a necessidade de um anúncio prévio; mas o sistema deve continuar funcionando adequadamente. Já a alta disponibilidade dos arquivos é essencial, pois, como os arquivos podem estar espalhados em diversos *peers* do sistema, é essencial que esses dados sejam acessíveis independentemente do estado faltoso ou não de algum de seus membros. Portanto, o sistema proposto deve aplicar técnicas de replicação de arquivos para garantir a alta disponibilidade e técnicas de monitoramento e comunicação em grupo para garantir a tolerância a faltas.

A lista abaixo elenca os objetivos específicos que se deseja alcançar neste trabalho:

- Criar um modelo de um sistema de arquivos distribuído que suporte adequadamente o compromisso entre persistência e espaço disponível existente no TFS;
- Criar uma arquitetura distribuída para o sistema proposto baseada no paradigma de redes *peer-to-peer* e totalmente descentralizada, para que não exista a necessidade de um controle central;

- Avaliar o impacto do uso de arquivos transparentes para criação de sistemas de arquivos distribuído cooperativos.

4.4 Arquitetura do Sistema Proposto

Uma arquitetura para o sistema de arquivos distribuído que utiliza os benefícios trazidos pelo TFS e supera os problemas de remoção imprevisível dos seus arquivos transparentes é proposta nesta seção. O espaço de disco livre em cada nó é usado para armazenar réplicas dos arquivos transparentes. Cada nó participante da rede *peer-to-peer* é responsável por gerenciar as réplicas armazenadas localmente, interagir com os outros nós a fim de manter um número constante de réplicas disponíveis, e interagir com as aplicações oferecendo uma abstração de arquivos. Através do uso de réplicas, os arquivos ficarão disponíveis mesmo na presença de nós faltosos, particionamento da rede *overlay* e remoções de arquivos transparentes.

Além da arquitetura geral, o núcleo do modelo é a tolerância da destruição imprevisível de arquivos transparentes, desta forma é proposto o conceito de *invalidação de réplica*. Uma *invalidação de réplica* indica que uma réplica de um arquivo transparente foi removida ou sobrescrita em algum nó da rede. Quando uma *invalidação de réplica* ocorre, o nó (no qual a invalidação ocorreu) informa aos outros nós que uma réplica do arquivo foi removida. Desta forma, o sistema tem a oportunidade de redistribuir as réplicas nos nós e de assegurar que o mesmo número de réplicas continue disponível no sistema.

4.4.1 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 transparente 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 4.1: *Aplicação Distribuída*, *Sistema de Arquivos Distribuído*, *Gestor de Armazenamento*, *Gestor de Replicação*, *Gestor de Transferência*, *Transparent File System Local* e *Substrato P2P*.

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*. Múltiplos tipos de aplicações podem ser construídas 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

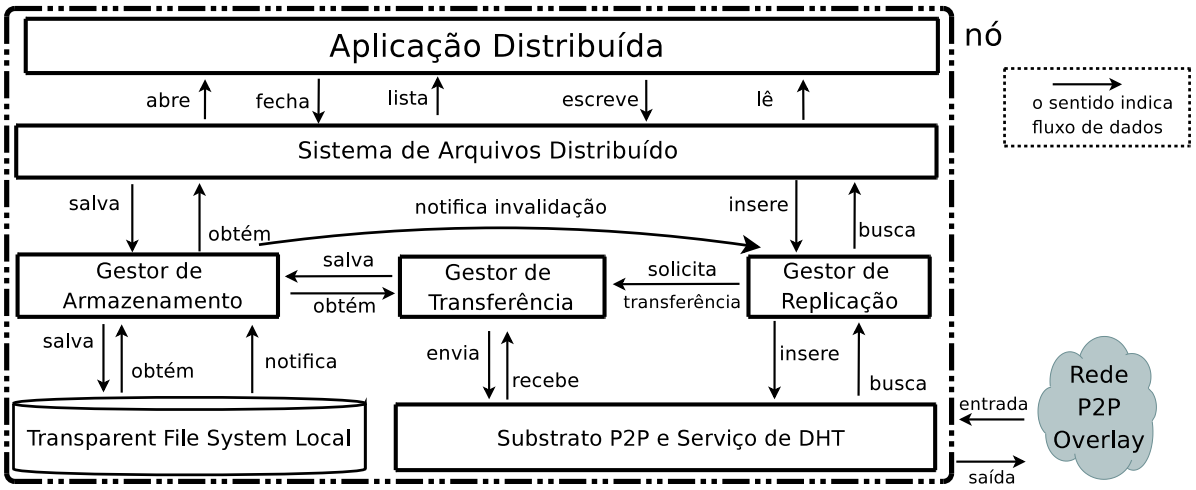

Figura 4.1: Modelo arquitetural do sistema proposto

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 4.4.4.

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.

4.4.2 Propriedades e Definições do Substrato *Peer-to-Peer*

Substratos *p2p*, como Chord [Stoica et al., 2001], Pastry [Rowstron and Druschel, 2001a] 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.

4.4.3 Gerenciando o 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. p_i envia a mensagem $repl\_req(f)$ para p_j
5. p_j envia a mensagem $file\_req(f)$ para p_i
6. p_i 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 [1] que possua uma réplica do arquivo ($r_k(f)$). Então, o *Gestor de Transferência* solicita a transferência de uma réplica do arquivo f de p_k ($file\_req(f)$)

[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

e a armazena localmente, como um arquivo transparente. Após o término da transferência, a solicitação da aplicação é atendida.

Procedimento 2 Aplicação no nó p_i solicita um arquivo f

1. **if** $\nexists r_i(f)$ **then**
2. Busca $metadata(f)$ da DHT
3. Busca $p_k \in \mathbb{R}(f)$ da DHT
4. p_i envia mensagem $file\_req(f)$ para p_k
5. p_k transfere $r_k(f)$ para p_i
6. **end if**
7. Atende a solicitação da aplicação com a $r_i(f)$ local

4.4.4 Gerenciando a Invalidação 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 iniciado quando uma réplica é invalidada em um nó.

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 $repl\_inv(f)$ para o *Gestor de Replicação*. O *Gestor de Replicação* então indaga o *Gestor de Armazenamento* sobre a quantidade de espaço livre no disco local $fspace(p_i)$. Se existe espaço suficiente para restaurar a réplica, 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 $repl\_inv(f)$

1. **if** $fspace(p_i) \geq size(f)$ **then**
2. Busca $p_j \in \mathbb{R}(f)$ da DHT
3. Envia mensagem $file\_req(f)$ para to 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 o nó 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 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 *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** $\nexists r_k(f)$ **then**
2. **if** $fspace(p_k) \geq size(f)$ **then**
3. Busca $p_m \in \mathbb{R}(f)$ da DHT
4. Envia mensagem *file_req*(f) para p_m
5. **else**
6. Busca $p_m = first(\mathbb{L}(p_k) \backslash \mathbb{R}(f))$ da DHT
7. Encaminha mensagem *repl_req*(f) para p_m
8. **end if**
9. **else**
10. Busca $p_m = first(\mathbb{L}(p_k) \backslash \mathbb{R}(f))$ da DHT
11. Encaminha a mensagem *repl_req*(f) para o nó p_m
12. **end if**

4.4.5 Escopo e Limitações

O sistema proposto tem como objetivo avaliar a usabilidade de um sistema de arquivos local que emprega o conceito de arquivos transparentes para a criação de um sistema de arquivos distribuído tolerante a faltas, sendo que o foco principal da proposta é um gerenciamento adequado da falta de persistência dos arquivos transparentes em virtude do grande valor de espaço em disco livre que pode ser compartilhado pelos vários usuários. O TFS faz com que o compartilhamento de espaço livre em disco não seja mais um problema para os usuários porque reduz muito o impacto psicológico no usuário e não aplica nenhuma penalidade no desempenho da máquina local.

Para a criação de um sistema de arquivos distribuído é necessária a modelagem de todas as operações locais de um arquivo para o sistema proposto, o que não é o escopo deste trabalho. Desta forma, o sistema proposto foca em adicionar uma maneira de detectar que arquivos ou réplica de arquivos transparentes foram eliminadas em alguns nós, e restabelecê-las em outros nós. Assim, uma alta disponibilidade dos dados é mantida.

Contudo, o sistema aqui proposto possui algumas limitações que poderão ser suprimidas no futuro. A primeira delas é a tolerância somente a faltas de parada e faltas maliciosas.

Faltas maliciosas poderiam ser toleradas no sistema com a inserção de algoritmos de consenso bizantino durante o processo de replicação e restauração de arquivos, e com o uso de chaves criptográficas para cifrar meta-dados e conteúdos. A segunda é um estudo do comportamento dos algoritmos propostos em situações limítrofes e durante a entrada e saída de nós na rede. Outra limitação, é a falta de mecanismos para garantir a alta disponibilidade em situações extremas, nas quais somente uma réplica de um determinado arquivo existe no sistema. Nesse caso seria possível implementar um valor limítrofe para o número de réplica de um determinado arquivo no sistema. Assim, quando esse valor é atingido, pode-se tomar alguma atitude para que o arquivo não seja removido completamente.

4.5 Casos de Uso para o Sistema Proposto

Várias aplicações foram desenvolvidas ao longo dos anos a partir de sistemas de arquivos distribuídos. Essas aplicações usam as propriedades inerentes aos diversos sistemas de arquivos distribuídos para oferecer algum tipo de serviço ao seu usuário, como por exemplo um sistema de biblioteca virtual distribuída. Uma biblioteca virtual distribuída consiste em usar o espaço de armazenamento de várias máquinas para guardar o conteúdo de diversos artigos, e arquivos de mídia, que podem ser posteriormente buscados e lidos por outros usuários.

No entanto, quando as propriedades do sistema de arquivos distribuídos mudam, a forma como as aplicações são projetadas e desenvolvidas também deve mudar. Três aplicações (ou casos de uso) foram selecionadas para ilustrar o possível comportamento do sistema proposto. Quatro características inerentes ao sistema proposto são analisadas à luz de cada aplicação: semântica de consistência dos dados, segurança, operações sobre os arquivos e o espaço de nomes.

4.5.1 Biblioteca Virtual Distribuída

O primeiro caso de uso foi citado anteriormente, uma biblioteca virtual distribuída. Com a evolução das tecnologias de redes de computadores, cujos computadores estão interconectados através de uma rede de alta velocidade, múltiplos usuários possuem informações que gostariam de compartilhar com outros em um formato de biblioteca. Estes usuários desejam que suas informações, que podem ser em formato de artigos científicos, artigos de opinião, diário de atividades, arquivos de mídia e outros, sejam armazenadas por um período indeterminado e que possam ser lidas por diversas pessoas ao longo do tempo.

Espaço de Nomes: Para uma biblioteca, o espaço de nomes deve ser único, transparente e independente para todo o sistema. Como em uma biblioteca real, onde os livros são únicos e organizados em categorias (área, assunto, etc) bem definidas, os arquivos de uma biblioteca virtual devem possuir uma identificação única e serem organizados em categorias bem definidas. Além disso, o espaço de nomes deve ser transparente e independente, porque o usuário, não importando quem seja e onde esteja, deve ver sempre a mesma organização hierárquica dos artigos e categorias.

Semântica de Consistência dos Dados: Os arquivos (ou artigos) pertencentes ao sistema devem ser vistos e pesquisados por qualquer usuário, mas o seu conteúdo poderá ser modificado

somente pelo seu autor. Desta forma, como somente o dono do arquivo possui permissão para escrita, a consistência dos dados pode ser garantida através de um mecanismo de *lock* em escrita. Isto é, o dono do arquivo solicita um *lock* ao sistema, assim o arquivo fica indisponível para escrita enquanto o *lock* estiver ativo. Após o término da modificação ou inserção de um arquivo, o dono libera o arquivo para replicação e atualização das réplicas.

Segurança: Como os arquivos (ou artigos) são públicos, os seus nomes e categorias devem ser visíveis para todos os usuários. No entanto, o seu conteúdo deve ser assinado pelo autor através de uma chave criptográfica com o intuito de garantir a integridade, autoria e validade do artigo.

Operações: As operações de leitura (*read*), escrita (*write*), busca (*lookup*), abertura (*open*) e fechamento (*close*) de arquivos devem estar disponíveis para um sistema de biblioteca virtual. Contudo a operação de exclusão (*delete*) não é necessária, já que os artigos disponíveis no passado foram publicados e podem continuar a sendo acessados a qualquer momento.

4.5.2 *Backup* Distribuído

O segundo caso de uso é um sistema de *backup* distribuído, em que diversos usuários gostariam de guardar cópias de segurança de seus dados pessoais de forma segura e confiável em outras máquinas. Além disso, os usuários podem restaurar as cópias de segurança e modificá-las a qualquer momento. Vários usuários pertencem ao sistema de *backup* distribuído, mas são independentes. Isto é, os dados de um determinado usuário **A** podem estar armazenados como cópia de segurança nos discos dos usuários **B** e **C**. Mas, estes não sabem quais e de quem são os dados armazenados em seus discos.

Espaço de Nomes: Em um sistema de backup distribuído com múltiplos usuários independentes, cada usuário possui o seu espaço de nomes que é transparente mas não independente. Em outras palavras, os espaços de nomes de cada usuário é visível somente para ele, não existe nenhum tipo de compartilhamento de arquivos, somente de espaço livre em disco. Desta forma, a estrutura de diretórios e arquivos que o usuário possui é transparente para ele, ou seja, é sempre a mesma, independentemente da máquina em que se encontram os arquivos e diretórios. Esta estrutura não é independente, ou seja, pode ser armazenada em qualquer local do disco, desde de que mantenha sua organização interna intacta.

Semântica de Consistência dos Dados: Neste sistema a consistência é garantida através de mecanismos similares ao do AFS (descrito em detalhes na seção 2.3), todas as alterações são executadas inicialmente na máquina do usuário e posteriormente propagadas aos demais servidores de espaço livre em disco do sistema. Como o acesso de leitura e escrita é permitido somente a um usuário, este sistema simples é suficiente para garantir a consistência dos dados.

Segurança: Como o sistema é utilizado por diversos usuários que armazenam dados de outros usuários em seus discos, a segurança é um fator crítico para o correto funcionamento do sistema como um todo. Assim, tanto os nomes dos arquivos e seus meta-dados como o conteúdo dos arquivos são cifrados através de chaves de criptografia. Desta forma, existe uma proteção

para os usuários que cedem espaço para o sistema, já que eles não conhecem o conteúdo dos arquivos, que podem conter informações privilegiadas ou ilegais; e para os usuários que usam o sistema, porque ninguém poderá ler o conteúdo de suas cópias de segurança.

Operações: Além das operações comuns de um sistema de arquivos, operações de leitura (*read*), escrita (*write*), busca (*lookup*), abertura (*open*), fechamento (*close*) e exclusão (*delete*), existe uma operação de restauração (*restore*). Tal operação faz com que cópias de segurança substituam as originais. Assim, o usuário pode restaurar seu *backup* a qualquer momento.

4.5.3 Cache Web Distribuído

O último caso de uso é um sistema de cache web distribuído, onde as páginas mais visitadas são armazenadas em um sistema de cache composto pelo espaço em disco de vários computadores. Estes computadores trabalham de forma cooperativa, fazendo que as páginas mais visitas sejam acessadas mais rapidamente pelo usuários do sistema de cache.

Além dos computadores que armazenam os objetos mais visitados, o sistema possui ainda um *proxy*. Este *proxy* é responsável por buscar o objeto, que não está no sistema de cache local, requisitado por algum cliente na Internet, e adicioná-lo ao cache local. Portanto, quando um cliente deseja buscar um objeto na Internet, primeiramente ele consulta o cache local distribuído. Em caso de insucesso, o cliente solicita ao *proxy* que o busque. Assim, o *proxy* busca o objeto, repassa-o ao cliente requisitante e o adiciona ao cache local distribuído.

Espaço de Nomes: Como os objetos são armazenados através do seu caminho completo e único na Internet (URL), o espaço de nomes de um sistema de cache distribuído é único. Todos os usuário procuram por arquivos em um mesmo repositório ou uma mesma estrutura.

Semântica de Consistência dos Dados: Como neste sistema somente o *proxy* é capaz de adicionar novos arquivos ou atualizar arquivos já existentes, uma semântica *close-to-open* pode ser utilizada, na qual o arquivo atualizado ou recém inserido ficará disponível para leitura somente após o *proxy* terminar a escrita e fechar o arquivo.

Segurança: A segurança neste sistema está focada em garantir que o conteúdo do arquivo armazenado não seja corrompido ou modificado de maneira inapropriada, portanto a aplicação de operações de *hash* no conteúdo do arquivo é suficiente.

Operações: Somente as operações de leitura (*read*), busca (*lookup*), abertura (*open*), fechamento (*close*) estão disponíveis para o clientes do sistema de cache. Já para o *proxy*, as operações de escrita (*write*), abertura (*open*), fechamento (*close*) e exclusão (*delete*) estão disponíveis.

4.6 Conclusão do Capítulo

Foi apresentada neste capítulo a proposta de um sistema de arquivos distribuído tolerante a faltas que usa como base arquivos transparentes providos pelo TFS, quais aplicações

distribuídas poderiam se beneficiar dos arquivos transparentes do TFS. A proposta em si descreve a arquitetura do sistema, e discorre sobre os algoritmos propostos a fim de que o uso de arquivos transparentes torne-se viável na construção do sistema.

No próximo capítulo é apresentado um protótipo que implementa a proposta aqui apresentada como uma prova de conceito, bem como os resultados obtidos na avaliação de seu funcionamento.

Capítulo 5

Implementação e Avaliação de um Protótipo

5.1 Introdução

Com o intuito de avaliar a estratégia proposta foi implementado um protótipo do sistema proposto. Esse protótipo serve como uma prova de conceito, e possui um conjunto mínimo de funcionalidades que permitem avaliar o sistema proposto em condições de operação normal, ou seja, onde somente faltas de persistência dos arquivos transparentes ocorrem. Posteriormente, este protótipo pode ser estendido para criar umas das aplicações descritas na seção 4.5.

Neste capítulo, primeiramente, é apresentada uma descrição mais elaborada das duas principais tecnologias utilizadas no desenvolvimento do protótipo, que servem como base para o seu funcionamento juntamente com o TFS. Em seguida, é descrito como o protótipo foi implementado e o seu funcionamento. Por fim, os experimentos e seus resultados são mostrados e discutidos.

5.2 Tecnologias Utilizadas

Além do TFS, outras duas tecnologias possibilitaram a implementação de um protótipo do sistema: PAST/Pastry e INotify. O PAST é um arcabouço para o desenvolvimento de sistemas de armazenamento de dados em redes *peer-to-peer* que oferece escalabilidade, alta disponibilidade, persistência e segurança [Druschel and Rowstron, 2001b], já o Pastry [Rowstron and Druschel, 2001a] é um substrato *peer-to-peer* que oferece funcionalidades de uma DHT e é a base do PAST. O INotify é um sistema de notificação de eventos para o sistema de arquivos. Através dele, quaisquer modificações no sistema de arquivos local poderão ser notificadas para uma ou mais aplicações.

5.2.1 PAST

O PAST é um sistema *peer-to-peer* puro e auto organizável, que possibilita que clientes compartilhem espaço de armazenamento em disco entre todos os integrantes da rede. Como na maioria das redes *peer-to-peer*, os nós participantes são não-confiáveis e podem entrar e sair

da rede a qualquer momento, condições que o PAST também trata em sua arquitetura. Os três aspectos mais relevantes do PAST são:

1. O uso do *Pastry*, descrito em [Rowstron and Druschel, 2001a], como infra estrutura de localização e roteamento de mensagens e para a administração de uma rede *peer-to-peer*;

2. O uso da diversidade geográfica durante o processo de distribuição de réplicas entre os vários nós da rede, para distribuição equilibrada das réplicas entre os nós;

3. O uso opcional de *smartcards* para controlar a relação entre oferta e demanda de espaço de armazenamento dos nós da rede e implementar a segurança do sistema.

Arquitetura

A arquitetura do PAST está apoiada em quatro alicerces. O primeiro é o uso do *Pastry* como infra estrutura para redes *peer-to-peer* e esquema de roteamento de mensagens. O *Pastry* é capaz de estabelecer uma rede *overlay peer-to-peer* tolerante a faltas e auto-organizável, e rotear mensagens em menos que $\log_{16} N$ passos em média, onde N é o número de nós pertencentes à rede. O segundo é o uso da probabilidade estocástica para garantir uma distribuição equilibrada das réplicas dos dados entre os nós da rede, sem a necessidade de um controle centralizado ou de protocolos de consenso distribuído. O terceiro é um controle descentralizado de armazenamento com suporte a *cache*, que faz o balanço entre o espaço de armazenamento utilizado nos nós e no sistema como um todo. Desta forma, o sistema é capaz de se equilibrar, manter os nós sem uma carga contínua de trabalho e utilizar todo o armazenamento provido pelo sistema. O quarto alicerce é o uso opcional de *smartcards*, que implementam um controle de quotas para equilibrar a razão entre necessidade de espaço e oferta de espaço.

O PAST é um sistema composto por nós conectados entre si através de uma rede *overlay peer-to-peer*. Cada um desses nós pode atuar como cliente, usando espaço disponível no sistema, ou como servidor, fornecendo espaço de armazenamento para o sistema; ou ainda como ambos. Ao se juntar à rede, cada nó recebe um identificador único de 128 bits (*nodeId*) que servirá para a identificação do nó na rede e é criado a partir de uma operação de *hash* em sua chave pública.

Sendo um arcabouço para sistemas de armazenamento de arquivos, o PAST possui uma semântica um pouco diferente de sistemas de arquivos convencionais para operações de inserção, busca e remoção, conforme comentado em [Rowstron and Druschel, 2001b].

Para cada arquivo inserido no sistema, um identificador de 160 bits *quasi* único (*fileId*) é calculado para o arquivo. Assim, os arquivos inseridos na rede são imutáveis, ou seja, não podem ser inseridos mais de uma vez sem que o identificador do arquivo mude. O *fileId* é computado através de operações de *hash* contendo o nome do arquivo, a chave pública do dono do arquivo e uma semente aleatória. Além disso, um certificado é emitido para cada arquivo (*file certificate*) contendo o identificador do arquivo, um fator de replicação do arquivo, uma semente aleatória, a data da inserção e o resultado de uma operação de *hash* no conteúdo do arquivo. O certificado é assinado pelo dono do arquivo através da sua chave privada. Após a geração do certificado, o arquivo é adicionado ao sistema e cópias são enviadas para outros nós na rede, de acordo com o fator de replicação contido no certificado.

Durante a busca de um determinado arquivo, uma mensagem de *lookup* com o fileId é propagada na rede pelo *Pastry*, que garante que esta mensagem chegará ao nó mais próximo que

possui uma das cópias do arquivo em questão. Uma mensagem de resposta com informações de como buscar o conteúdo do arquivo é enviado ao solicitante, possibilitando que o cliente pegue o arquivo em questão.

A remoção de arquivos do sistema não é garantida pelo PAST. Ao invés da remoção completa, o PAST oferece uma operação de solicitação de remoção (*reclaim*) que não garante que o arquivo seja removido completamente, mas remove o espaço utilizado (a quota) no sistema pelo arquivo. O cliente envia uma solicitação de remoção que contém um certificado de *reclaim* para a rede. Os nós que contêm as réplicas do arquivo a ser removido verificam a legitimidade da solicitação e emitem um recibo de remoção que é usado para redução do espaço utilizado pelo arquivo no sistema. Este método de remoção diferenciado permite que a operação de remoção seja implementada sem a necessidade de implementação de procolos distribuídos de consenso que geralmente são custosos ao desempenho do sistema.

Pastry

O *Pastry* é um sistema capaz de criar redes sobrepostas *peer-to-peer* e rotear mensagens de forma eficiente, escalável, tolerante a faltas e auto-organizável. Esta seção descreve sucintamente o funcionamento do *Pastry*, que é descrito em detalhes em [Rowstron and Druschel, 2001a].

Cada nó pertencente à rede do *Pastry* possui um identificador único (o *nodeId*). Este identificador é representado por um número de 128 bits, que é gerado a partir do endereço IP ou da chave pública do nó quando este entra na rede. Além disso, as informações são armazenadas na rede sempre em forma de um par (chave, valor). Assim, o *Pastry* implementa uma *Distributed Hash Table* (DHT), onde os pares (chave, valor) são colocados em nós com o identificador mais próximo numericamente ao valor da chave. Para o PAST, cada par (chave, valor) na rede significa um arquivo no sistema.

Para armazenar e localizar estes pares, cada nó do *Pastry* utiliza uma tabela de roteamento. Nessa tabela de roteamento existem ponteiros para os outros nós da rede, mas não todos. A tabela de roteamento existente em cada nós armazena ponteiros para os nós vizinhos e para nós folha (*leaf nodes*). Os nós vizinhos são os nós adjacentes ao nó em questão e são utilizados para controlar a entrada e saída dos nós na rede; já os nós folhas são usados com a intenção de roteamento de mensagens para nós mais distantes. A figura 5.1 representa um exemplo da tabela de roteamento, onde podem ser vistas três áreas: uma com os nós folha, outra com os nós vizinho e uma tabela de roteamento. Cada número na figura representa um *nodeid* de um nó na rede *peer-to-peer*.

Segurança

O modelo de segurança adotado pelo PAST é baseado em *smartcards*, que não podem ter seu comportamento alterado por ataques maliciosos [Rowstron and Druschel, 2001b]. Cada nó e cada usuário do sistema possui um *smartcard*, ao qual está associado um par de chaves (pública, privada) assinada pelo detentor do *smartcard*. A sua principal função é a geração e verificação dos certificados digitais usados nas operações do PAST: geração de *nodeids*, geração de certificados de arquivos e recibos de armazenamento e geração e verificação de certificados para solicitações de remoção de arquivos;

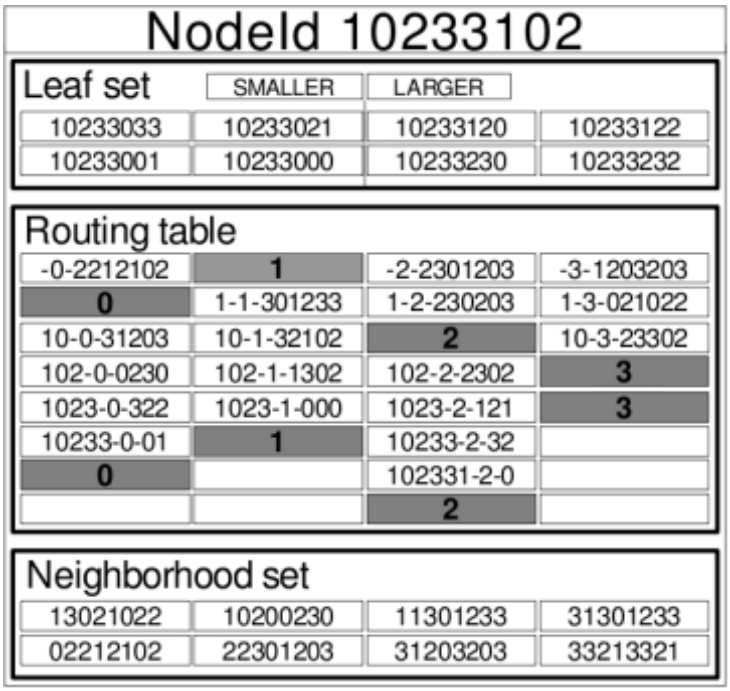

Figura 5.1: Exemplo de tabela de roteamento de um nó no *Pastry* [Rowstron and Druschel, 2001a]

5.2.2 INotify

O INotify é um sistema de notificação de eventos para o sistema de arquivos do Linux. Ele permite que modificações ocorridas no sistema de arquivos, por exemplo se um dado arquivo foi removido, sejam notificadas de forma assíncrona para aplicações sendo executadas no espaço de usuário (*user space*). Foi desenvolvido por Robert Love e John McCutchan para o kernel 2.6.13, e implementado como um módulo do *kernel*, conforme descrito em [Love, 2005].

O INotify não é o primeiro sistema com essa funcionalidade a ser criado, este tipo de sistema existe em diversas plataformas computacionais, como o Linux, o Unix e o Windows. O primeiro sistema desse tipo, chamado de *File Alteration Monitor (FAM)* foi criado pela *Silicon Graphics* para a plataforma Unix. Através do uso de notificações e eventos, aplicações que interagem com o sistema de arquivos da plataforma podem atualizar o seu estado, informar o usuário de algum evento e executar certa ações, sem um alto custo de processamento e consumo de memória.

Arquitetura

Um dos fatores que deram origem à criação do INotify foram as deficiências criticas do seu antecessor, o DNotify. Essas deficiências são:

1. observa e notifica somente alterações em diretórios;
2. usa sinais (*signals*) para comunicação entre *kernel* e aplicações;
3. requer a abertura de um descritor de arquivo (*file descriptor*) para cada diretório a ser observado.

Chamada de Sistema	Descrição
`inotify_init`	Cria e inicializa uma instancia do INotify e retorna o descritor do arquivo onde todos os eventos gerados serão armazenados.
`inotify_add_watch`	Adiciona um *watch* para um determinado arquivo/diretório, e retorma um identificador único.
`inotify_rm_watch`	Remove o watch especificado pelo identificador.

Tabela 5.1: Chamadas de Sistema providas pelo INotify

O INotify é implementado como um módulo do *kernel* que interage diretamente com a camada de abstração de sistema de arquivos, o *Virtual File System (VFS)*. O VFS é uma camada de abstração para tratamento de sistemas de arquivos que exporta uma interface padrão para as aplicações independentemente do sistema de arquivos em uso no meio físico (por exemplo, FAT32, Ext2, ...). Um dos objetos constituintes do VFS é os *inode*, que é responsável por armazenar todas as informações necessárias (metadados) para que o sistema de arquivos possa manipular diretórios e arquivos. Desta forma, o INotify é capaz de monitorar diretamente os arquivos e diretórios através dos seus *inodes*, originando assim o nome: **I**node **Notify**.

Graças à sua interação com o VFS, no INotify somente um *file descriptor* é aberto para cada aplicação reduzindo assim o uso de recursos do *kernel* e a complexidade para a aplicação. Além disso, essa abordagem permite que os eventos gerados pelo sistema sejam direcionados corretamente às aplicações em um cenário de multiplexação de entrada/saída, ou seja, quando diversas aplicações desejam monitorar os eventos no sistema de arquivos.

Implementação

Um conjunto específico de chamadas de sistemas (*system calls*) é apresentado às aplicações pelo módulo do INotify. Assim, as aplicações podem obter eventos do que acontece no sistemas de arquivos de forma simples e fácil. Estas *system calls* instruem o módulo do INotify para executar algumas operações: inicialização, adição de *watches* e remoção de *watches*. *Watches* podem ser entendidos como uma observação de eventos em determinados arquivos ou diretórios e são a parte principal do sistema. As aplicações podem, através da chamada de sistema para adicionar *watches*, solicitar que determinados eventos que acontecam no sistema de arquivos seja informados para ela. A Tabela 5.1 enumera as chamadas de sistema exportadas pelo módulo do INotify.

Adicionalmente, o sistema é capaz de notificar o acontecimento de diversos eventos. Estes eventos podem ser identificados no momento da adição ou remoção de um *watch*. Em outras palavras, a aplicação pode selecionar quais eventos ela gostaria de receber uma notificação pelo INotify. Isso torna o sistema de notificação flexível e adaptável para qualquer tipo de monitoramento que se deseja. A Tabela 5.2 elenca os eventos disponibilizados pelo sistema e que podem ser solicitados para os *watches*.

5.3 Implementação

O protótipo foi implementado usando quatro projetos *open source*: FreePastry [Druschel and Rowstron, 2001a], INotify [Love, 2005], JNotify [Yadan, 2005] e o TFS

Evento	Descrição
IN_ACCESS	Arquivo foi lido
IN_MODIFY	Arquivo foi modificado
IN_ATTRIB	Atributos do arquivo (metadados) foram modificados
IN_CLOSE_WRITE	Arquivo foi fechado, mas estava aberto para escrita
IN_CLOSE_NOWRITE	Arquivo foi fechado, mas não estava aberto para escrita
IN_CLOSE	Arquivo foi fechado, não importa tipo de acesso de leitura
IN_OPEN	Arquivo foi aberto
IN_MOVED_FROM	Arquivo foi movido para fora do *watch*
IN_MOVED_TO	Arquivo foi movido para dentro do *watch*
IN_MOVE	Arquivo foi movido para dentro ou fora do *watch*
IN_DELETE	Arquivo deletado
IN_DELETE_SELF	*Watch* deletado
IN_ALL_EVENTS	Solicita uma notificação para todos os eventos

Tabela 5.2: Eventos providos pelo INotify

[Cipar et al., 2007]. Eles foram adaptados e ligados através de 2.500 linhas de código Java. FreePastry é uma implementação livre, escrita em linguagem Java, do protocolo de roteamento e do substrato *peer-to-peer* do Pastry. Além disso, ele também contém uma implementação do PAST e de um serviço de DHT que é baseado no Pastry. O INotify é um módulo do kernel do linux que provê mecanismos para receber notificações do sistema de arquivos local caso algo seja modificado, e foi descrito em detalhes na seção 5.2.2. 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 e pelo sistema de arquivos local.

O conjunto de classes implementadas pelo código em Java é responsável por interligar todos esses componentes de forma que todos trabalhem conjuntamente, e oferecer a base para o desenvolvimento de uma aplicação que utilize arquivos transparentes em um modelo de sistemas de arquivos distribuído. Esse conjunto de classes é composto por quatro módulos e segue a arquitetura apresentada no capítulo 4: *Sistema de Arquivos Distribuido Simples*, *Gestor de Armazenamento*, *Gestor de Replicação*, e *Gestor de Transferência*.

O *Sistema de Arquivos Distribuido Simples* se baseia nos serviços oferecidos pelo PAST para implementar um sistema de arquivos simples e *flat*.O *Gestor de Armazenamento* é responsável por armazenar réplicas dos arquivos no sistema de arquivos local transparente (TFS), já o *Gestor de Replicação* é responsável por manter um número constante de réplicas disponíveis de cada arquivo. Finalmente, o *Gestor de Transferência* tem a função de transferir réplicas de um nó para outro.

O protótipo implementa as três funções principais da proposta: a inserção de um arquivo no sistema, a busca de arquivos na rede e o tratamento das remoções dos arquivos transparentes existentes no sistema. Desta forma, os quatro procedimentos apresentados no capítulo 4 foram implementados.

Quando um arquivo f é inserido no sistema, a primeira ação é calcular o seu *fileid*. Logo após as aplicações obtêm todas as informações do arquivo e monta um meta-dado desse arquivo, que será disponibilizado na DHT, com o seu tamanho, data de criação e usuário. Então, o objeto contendo todos os meta-dados de arquivo f é inserido no PAST/DHT e um fator de

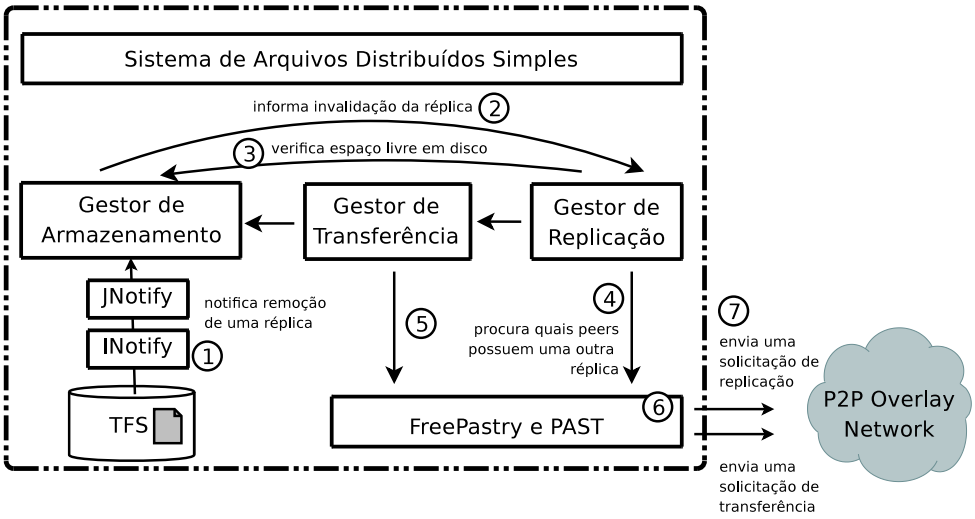

Figura 5.2: Diagrama de funcionamento do protótipo perante a remoção de um arquivo transparente

replicação k é atribuído. O objeto contendo os meta-dados é a seguir propagado para todos os nós que serão responsáveis por armazenar uma réplica do arquivo f. Assim, cada nó solicitará a transferência dos dados a fim que ele possa armazenar a réplica como um arquivo transparente no seu sistema de arquivos local.

Os passos a seguir, ilustrados na Figura 5.2, são executados quando um arquivo transparente f é removido/sobrescrito do sistema:

1. O módulo do INotify detecta que o arquivo transparente f foi deletado e informa ao JNotify;
2. O *Gestor de Armazenamento* recebe esta notificação do JNotify e repassa a informação ao *Gestor de Replicação* através de uma mensagem interna que contém o nome do arquivo e o seu *fileid*;
3. O *Gestor de Replicação* verifica se existe ainda espaço no disco para restaurar a réplica perdida;
4. Em caso positivo (existe espaço livre suficiente) o *Gestor de Replicação* faz uma busca na DHT, usando o *fileid* como chave. Essa busca retorna uma resposta contendo o objeto com os meta-dados do arquivo e um nó que possui uma réplica do arquivo. Então, o *Gestor de Transferência* é instruído a executar a transferência da réplica;
5. De posse do *fileid* e do *nodeid* do nó que possui uma cópia do arquivo f, o *Gestor de Transferência* então inicia o processo de transferência dos dados;
6. Em caso negativo (não existe espaço disponível no disco), o *Gestor de Replicação* busca na rede o próximo nó que poderá armazenar a réplica do arquivo;
7. Finalmente, o *Gestor de Replicação* então envia a solicitação de réplica para esse nó que poderá armazenar a réplica do arquivo f.

5.4 Avaliação

A avaliação do protótipo verifica o comportamento do sistema de arquivos distribuído proposto somente na presença das faltas de persistência geradas pelas remoções de seus arquivos transparentes. Outros tipos de faltas, como faltas de parada e faltas maliciosas, não fazem parte do escopo dessa avaliação de prova de conceito. Espera-se que o sistema proposto seja capaz de recuperar as réplicas removidas ou sobrescritas, de forma que a alta disponibilidade dos dados não seja afetada.

Além disso, o custo do algoritmo de recuperação de réplicas desenvolvido para tolerar adequadamente esse tipo de falta é também avaliado. Normalmente, qualquer operação em um sistema de arquivos local ou distribuído está fortemente relacionada com o tamanho do arquivo. No sistema aqui proposto esperasse que exista essa mesma correlação, onde o tempo gasto para recuperar uma ou mais réplicas removidas/sobrescritas seja proporcional ao tamanho da réplica.

5.4.1 Ambiente de Avaliação

Todos os testes foram realizados em uma única máquina, um computador HP AMD Turion 64 X2 com *clock* de 2.1 Ghz, 3GB de memória RAM e 250 GB de disco rígido. Nesse computador foi instalado o sistema operacional Linux Debian Sarge 3.1 mínimo 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 *peer-to-peer* são executados nessa única 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 espera-se imitar 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.

5.4.2 Metodologia

Três testes foram executados para avaliar o comportamento e o desempenho do sistema proposto. Para esse testes foi executado o mesmo procedimento de preparação e finalização do ambiente, a fim de que os resultados fossem os mais confiáveis possível. Além disso, o valor mostrado foi calculado com base na média dos valores obtidos através de três execuções de cada experimento, e o desvio padrão ficou abaixo de 5%. O sistema distribuído é posto em ação, e 20 nós independentes são inicializados e conectados através de uma rede *peer-to-peer overlay* estabelecida pelo *Pastry*.

A seguir, com todos os nós rodando adequadamente, o sistema de arquivos distribuído é populado com 25 arquivos de exemplo, onde o tamanho desses arquivo varia 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 teremos 3 réplicas disponíveis e espalhadas no sistema além do arquivo original. Adicionalmente, a interface de rede virtual de cada nó conectado à *bridge virtual* possui um sistema de controle de largura de banda, possibilitando assim a simulação de outras condições de banda disponível existentes atualmente. Após a execução de cada teste, e a coleta das medições, todo o sistema é reiniciado. Isto é, os nós são eliminados, a máquina Java

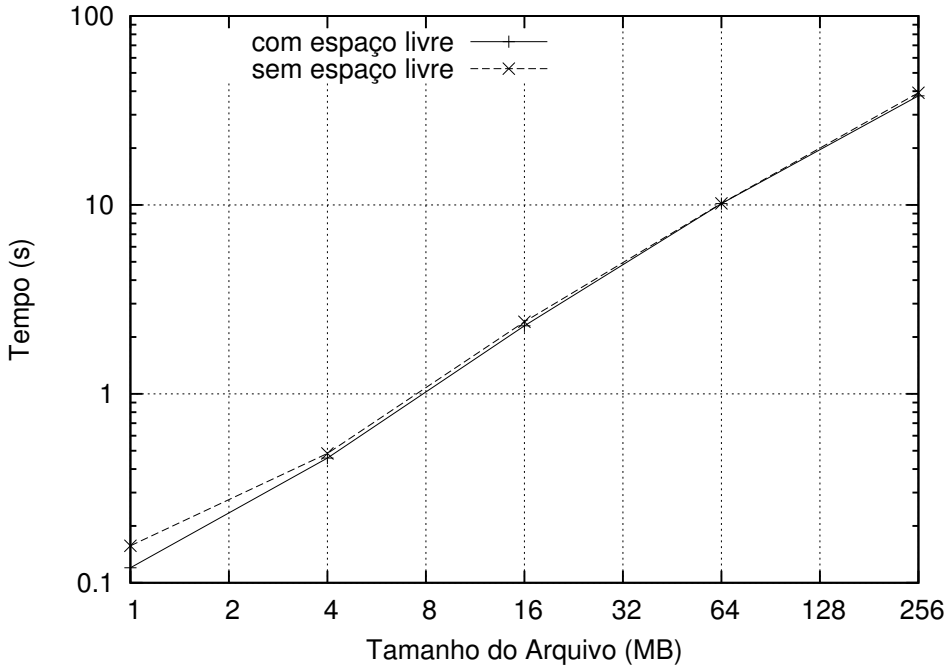

Figura 5.3: Tempo de recuperação de uma réplica invalidada em um nó com espaço livre em disco e em um nó sem espaço livre em disco.

virtual é removida, os arquivos transparentes são apagados, a *bridge virtual*, e suas conexões são removidas.

Além disso, com o intuito de criar um ambiente controlado para a deleção e sobrescrita de arquivos transparentes, as réplicas são deliberadamente removidas do TFS. Essa deleção intencional gerará exatamente a mesma cadeia de eventos no caso de um arquivo transparente for sobrescrito pelo TFS.

O primeiro experimento foi executado com o intuito de medir o custo adicional causado pelos Procedimentos 3 e 4 de recuperação de réplicas descrito no capítulo 4. Nesse experimento foi medido o tempo gasto para que o sistema recupere uma (1) réplica removida. Essa medição foi feita em duas situações distintas:

1. Quando o nó onde a réplica foi removida 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 removida não possui mais espaço livre em disco, e o sistema precisa achar outro nó que possa armazenar a réplica.

A Figura 5.3 mostra o resultado desse experimento. Pode-se observar que o custo adicional para a execução dos procedimentos de recuperação de réplicas é baixo se comparado com o tempo para transferir a réplica de um nó para outro. Verifica-se um custo proporcional e significativo somente quando o arquivo tem tamanho abaixo de 4 MB.

O segundo experimento realizado tem por objetivo verificar o correto comportamento do sistema caso várias deleções de réplica de um mesmo arquivo ocorram simultaneamente (1,2 ou 3 deleções simultâneas de réplicas de um mesmo arquivo). Como a manutenção da

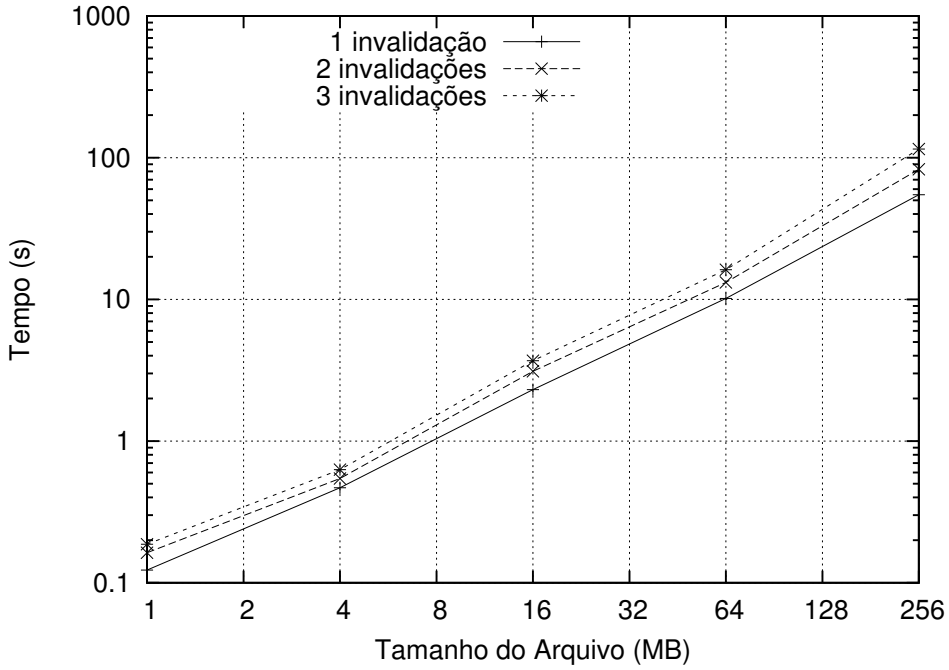

Figura 5.4: Tempo de recuperação de 1,2 e 3 réplicas removidas simultaneamente

alta disponibilidade dos dados é uma premissa do sistema proposto, espera-se que após algum tempo todas as réplicas removidas voltem a ficar disponíveis. Além disso, para exercitar o procedimento completo de recuperação de réplicas, esse experimento é medido somente no caso em que o nó onde a réplica foi removida não possui mais espaço livre em disco. Desta forma, é necessária uma atuação do mecanismo de auto-organização do sistema para identificar os melhores nós onde serão restauradas as réplicas.

A Figura 5.4 apresenta o resultado desse experimento. Pode-se observar que após um certo tempo, todas as réplicas removidas voltam a estar disponíveis no sistema. Nesse experimento também foi medido o tempo gasto para recuperar todas as réplicas. Outra observação interessante é que o tempo gasto está proporcional ao tamanho do arquivo, o que era previamente esperado.

O terceiro experimento teve como meta verificar o desempenho do sistema em situações onde diferentes larguras de banda estão disponíveis para conexão com outros computadores. Três valores de largura de banda foram escolhidos para simular uma conexão de computadores em rede local e na internet: 100Mbit/s, 3 Mbit/s e 1 Mbit/s. A Figura 5.5 ilustra o resultado desse experimento. Pode-se verificar que a réplica removida foi restaurada nas três condição de banda, mas que o seu tempo variou de acordo com o tamanho do arquivo e a largura de banda disponível.

5.4.3 Discussão dos Resultados

Os testes realizados mostram que o sistema proposto é capaz de tolerar a remoção imprevisíveis de arquivos transparentes através dos algoritmos desenvolvidos para a recuperação de réplicas. Desta forma, foi feita uma prova de conceito, onde o compromisso introduzido pelo TFS de sacrificar a persistência pelo maior espaço em disco livre compartilhado e por um

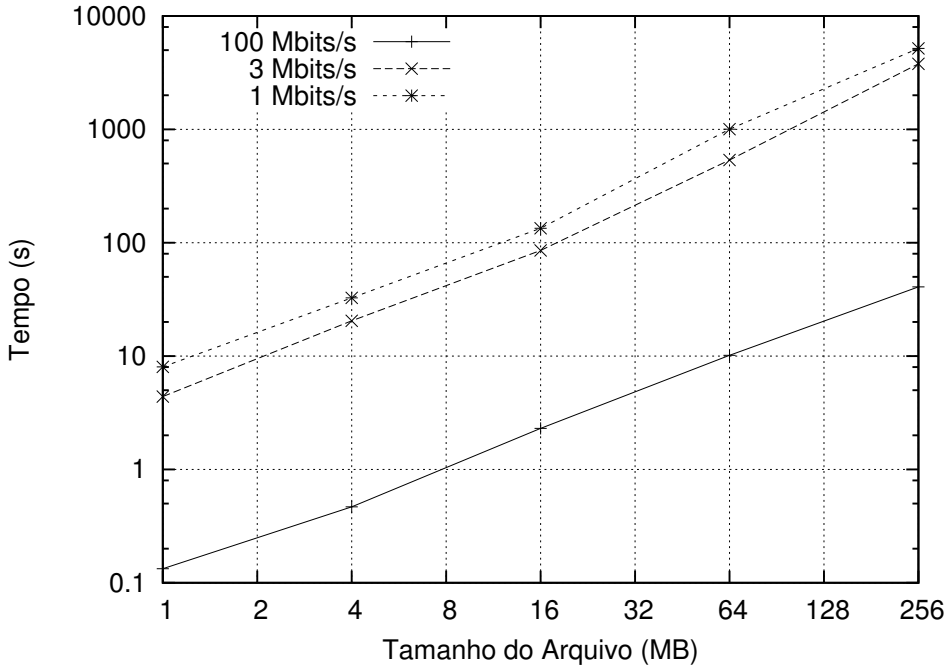

Figura 5.5: Tempo de recuperação de uma (1) réplica com diferentes larguras de banda

menor efeito psicológico é aceitável e que a tolerência a faltas de persistência não adiciona um impacto significativo no desempenho do sistema. Todavia, os testes foram executados em condições bem controladas e estáticas. Para complementar a avaliação dos algoritmos é necessário a realização de experimentos em condições dinâmicas (*churn*).

Além disso, é possível perceber que o custo do desempenho dos algoritmos propostos é baixo, onde a maior parte do tempo é gasta na transmissão dos arquivos. Esse tempo de recuperação é proporcional ao tamanho do arquivo e à largura de banda disponível.

5.5 Conclusão do Capítulo

Neste capítulo foram apresentadas as tecnologias utilizadas para a criação do protótipo do sistema proposto, foi descrito como foi implementado o protótipo e explicado o seu funcionamento; por fim foram mostrados os resultados dos experimentos realizados para comprovar o funcionamento do sistema e o tempo gasto na recuperação das réplicas sobrescritas pelo TFS.

Como resultado final dos experimentos pode-se dizer que o sistema proposto atende às suas premissas de desenvolvimento, sendo capaz de manter uma alta disponibilidade dos dados de um sistema de arquivos distribuído mesmo na presença de faltas de persistência.

Capítulo 6

Conclusão

Este trabalho apresentou um modelo de sistema de arquivos distribuído tolerante a faltas como prova-de-conceito de que o uso de arquivos transparentes é viável para a construção de sistemas similares. O sistema é baseado no modelo *peer-to-peer*, é composto por diversos nós, formando uma rede *overlay*, e os seus arquivos são armazenados no disco como arquivos transparentes. Com o uso de arquivos transparentes para armazenar os dados pertencentes ao sistema distribuído, a persistência desses dados não é assegurada pelo sistema de arquivos local. Desta forma, este trabalho propôs um método para prevenir que o sistema entre em um estado errôneo quando uma falta de persistência ocorre. Quando a exclusão de uma réplica ocorre, o sistema a detecta e executa uma série de passos, a fim de que o número de réplica disponíveis no sistema permaneça o mesmo. Desta forma, a alta disponibilidade dos dados é assegurada.

Um protótipo do sistema proposto foi implementado e testado. Os experimentos realizados mostraram a viabilidade de tolerar a não-garantia de persistência dos dados existente no TFS, e que o custo dessa tolerância é proporcional ao tamanho dos arquivos inseridos no sistema. Isto é, os procedimentos e algoritmos propostos não impõem uma severa penalidade no desempenho do sistema, mas quanto maior o arquivo a ser replicado, maior será o tempo gasto para recuperar uma réplica removida. Como as exclusões dos arquivos transparentes não são controladas, e dependem da atividade de cada nó participante do sistema, a ocorrência de exclusões simultâneas também é endereçada pelos algoritmos propostos. Essas exclusões também não impõem um custo adicional significativo durante o processo de restabelecimento das réplicas. Experimentos mais detalhados em um ambiente real e com situações mais dinâmicas (*churn*) deverão ainda ser realizados.

O conceito de arquivos transparentes foi introduzido recentemente pelo TFS [Cipar et al., 2007], assim não foi encontrado nenhum outro projeto similar. No entanto, o compartilhamento de recursos computacionais de forma transparente aos usuário vem sendo utilizado há algum tempo em sistemas computacionais. O sistema proposto, juntamente com o TFS, é capaz de compartilhar espaço disponível em disco para armazenamento de dados de forma transparente para o usuário, da mesmo forma com que os ciclos de CPU são compartilhados para a resolução de cálculos científicos complexos (conforme descrito na seção 3.4).

Como este trabalho foi o primeiro a aplicar arquivos transparentes em um sistema de arquivos distribuído, alguns pontos podem ser melhorados e aperfeiçoados em trabalhos futuros. O modelo aqui proposto não trata situações limítrofes, como, por exemplo, 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 dis-

poníveis 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. Além disso, 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 remoção apenas dos blocos sobrescritos. Nessa situação, somente os blocos excluídos poderiam ser restaurados, desta forma o tempo gasto para a restauração de réplicas diminuiria consideravelmente.

Referências Bibliográficas

[Adya et al., 2002] 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 et al., 2002] 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.

[Anderson et al., 1995] Anderson, T., Dahlin, M., Neefe, J., Pat-terson, D., Roselli, D., and Wang, R. (1995). Serverless network file systems. In *15th ACM Symposium on Operating System Principles*, pages 109–126, Copper Mountain Resort, Colorado.

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

[Barak and La'adan, 1998] Barak, A. and La'adan, O. (1998). The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372.

[Card et al., 1994] Card, R., Ts'o, T., and Tweedie, S. (1994). Design and implementation of the second extended filesystem. In *First Dutch International Symposium on Linux*.

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

[Cipar et al., 2007] 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 (FAST '07)*.

[Dabek et al., 2001] 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 (SOSP '01)*.

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

[Druschel and Rowstron, 2001b] 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*, pages 75–80.

[Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., and Leung, S. (2003). The Google file system. In *ACM Symposium on Operating Systems Principles*.

[Golle et al., 2001] 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*.

[Gorn, 1965] Gorn, S. (1965). Transparent-mode control procedures for data communication, using the american standard code for information interchange - a tutorial. *ACM Communications*, 8(4):203–206.

[Hasan et al., 2005] 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*.

[Howard, 1988] Howard, J. H. (1988). An Overview of the Andrew File System. In *USENIX Winter Technical Conference*.

[Howard et al., 1988] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. (1988). Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81.

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

[Karger et al., 1997] 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 *Proceedings of the twenty-ninth annual ACM Symposium on Theory of computing*, pages 654–663.

[Kubiatowicz et al., 2000] 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)*.

[Larson et al., 2002] Larson, S., Snow, C., Shirts, M., and Pande, V. (2002). Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 1.

[Leonard et al., 2002] 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 CUCS-014-02, Department of Computer Science, Columbia University.

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

[Mauthe and Hutchison, 2003] 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*.

[Mazières et al., 1999] Mazières, D., Kaminsky, M., Kaashoek, M. F., and Witchel, E. (1999). Separating key management from file system security. In *17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 124–139.

[Microsystems, 1994] Microsystems, S. (1994). NFS: Network file system version 3 protocol specification. Technical report, Sun Microsystems Incorporated.

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

[Parnas and Siewiorek, 1975] Parnas, D. L. and Siewiorek, D. P. (1975). Use of the concept of transparency in the design of hierarchically structured systems. *ACM Communications*, 18(7):401–408.

[Patterson et al., 1988] Patterson, D. A., Gibson, G., and Katz, R. (1988). A case for redundant arrays of inexpensive disks (raid). In *1988 ACM SIGMOD Conference on Management of Data*, pages 109–116.

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

[Pawlowski et al., 1994] Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., and Hitz, D. (1994). NFS version 3: Design and Implementation. In *Summer USENIX Conference*.

[Pawlowski et al., 2000] Pawlowski, B., Shepler, S., Beame, C., Callaghan, B., Eisler, M., Noveck, D., Robinson, D., and Thurlow, R. (2000). The NFS version 4 protocol. In *2nd Intl System Administration and Networking Conference*.

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

[Rowstron and Druschel, 2001b] Rowstron, A. and Druschel, P. (2001b). Storage Management and Caching in PAST, A large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*.

[Sandberg et al., 1985] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. (1985). Design and implementation of the Sun Network Filesystem. In *Summer 1985 USENIX Conference*, pages 119–130.

[Satyanarayanan, 1989] Satyanarayanan, M. (1989). A survey of distributed file systems. Technical Report CMU-CS-89-116, Carnegie Mellon University.

[Stein et al., 2002] Stein, C. A., Tucker, M. J., and Seltzer, M. I. (2002). Building a reliable mutable file system on peer-to-peer storage. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*.

[Stoica et al., 2001] 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 Conference*.

[Tanenbaum, 1994] Tanenbaum, A. S. (1994). *Distributed Operating Systems*. Prentice Hall.

[wan Ngan et al., 2003] 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, 2005] Yadan, O. (2005). JNotify. `http://jnotify.sourceforge.com`.

[Zhao et al., 2004] 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.