Tolerância a falhas no armazenamento distribuído de dados em grades oportunistas

Pablo Francisco Laura Huaman

Dissertação
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Raphael Yokoingawa de Camargo

São Paulo, março de 2010

Tolerância a falhas no armazenamento distribuído de dados em grades oportunistas

Esta versão definitiva da dissertação
contém as correções e alterações sugeridas pela
Comissão Julgadora durante a defesa realizada
por Pablo Francisco Laura Huaman em 2/6/2010.

Comissão Julgadora:

- Prof. Dr. Raphael Yokoingawa de Camargo (orientador) - UFABC
- Prof. Dr. Marco Dimas Gubitoso - IME-USP
- Prof. Dr. Jo Ueyama - ICMC-USP

Agradecimentos

No decorrer do mestrado, recebi a ajuda direta de diversas pessoas. Gostaria de agradecer à minha familia, aos meus pais Francisco e Agustina, à minhas irmas Edith, Angelica e Gisela. Seu apoio foi e será fundamental para mim sempre.

Gostaria de agradecer ao meu orientador, o professor Raphael Yokoingawa de Camargo, por compartilhar sua experiência ao trabalhar comigo durante este mestrado. Aprendi muito com ele, agradeço toda a sua ajuda, apoio e compreensão.

Finalmente agradeço aos meus amigos os chichos, gostaria especialmente de agradecer a Crhistian por sua ajuda durante o mestrado e aos meus amigos, Ivan, Jorge, Pablo, Marco, Juan Carlos, Yoryi, Alberto, Roberto, Carlos, Geiser, Wilbert, Mario, Vinicius, com os quais compartilhei a experiencia de fazer juntos o mestrado. Agradeço também à família Fernandes por compartilhar seus bons momentos comigo assim como a sua amizade, o mesmo para Jerônimo e Roseli, muito obrigado.

Resumo

Grades computacionais oportunistas permitem o compartilhamento de recursos computacionais ociosos, como processador, memória e espaço em disco, para a execução de aplicações que necessitam de grandes quantidades de poder computacional e para o armazenamento de dados de usuários e aplicações. OppStore é um sistema de middleware que realiza o gerenciamento das máquinas de uma grade computacional oportunista, permitindo o armazenamento de dados utilizando o espaço em disco livre durante seu período de ociosidade. Para permitir uma maior disponibilidade dos dados armazenados, OppStore codifica os arquivos em fragmentos redundantes, que são distribuídos em diferentes máquinas. Mas as máquinas da grade podem falhar, ficar inacessíveis ou passar de ociosas para ocupadas inesperadamente, impedindo o acesso aos fragmentos nelas armazenados. Um mecanismo de tolerância a falhas que permita manter a disponibilidade destes fragmentos é um quesito importante para este sistema.

Neste trabalho, definimos, analisamos, implementamos e avaliamos dois mecanismos de tolerância a falhas que permitem a recuperação de fragmentos perdidos devido à falhas ou indisponibilidades nas máquinas da grade. O primeiro mecanismo realiza a reconstrução do arquivo original, que é utilizado para gerar novamente os fragmentos perdidos. O segundo mecanismo mantém uma cópia adicional de cada fragmento, que é utilizada para recuperar os fragmentos perdidos sem a necessidade de reconstruir o arquivo original. Por meio de simulações, avaliamos o custo de cada mecanismo, como o número de mensagens geradas e a quantidade de tráfego na rede, e a capacidade de cada mecanismo de manter a disponibilidade dos arquivos armazenados na presença de falhas.

Palavras-chave: armazenamento distribuído, tolerância a falhas, grades computacionais, redes peer-to-peer.

Abstract

Opportunistic computational grids enable the sharing of idle computational resources, such as processor, memory, and hard disk space, for the execution of applications that need large amounts of computational power and for the storage of user and application data. OppStore is a middleware system that performs the management of machines from an opportunistic grid, enabling the the storage of data using the free disk space of these machines during their idle periods. To increase the availability of stored data, OppStore encodes the files in redundant fragments, that are distributed in several machines. But grid machines may fail, become inaccessible or change from idle to occupied unexpectedly, denying the access to stored fragments. A fault-tolerance mechanism that maintains the availability of these fragments is an important requisite for such a system.

In this work, we defined, analysed, implemented, and evaluated two fault-tolerance mechanisms that recover fragments lost due to failures or unavailabilities of the grid machines. The first mechanism performs the reconstruction of the original file, which is used to regenerate the lost fragments. The second mechanism maintains an additional copy of each fragment, that is used to recover the lost fragment without the need to reconstruct the original file. Using simulations, we evaluated the cost of each mechanism, such as the number of generated messages and the extra network traffic, and the efficiency in maintaining the availability of stored files in the presence of failures.

Palavras-chave: distributed data storage, fault-tolerance, computational grids, peer-to-peer networks.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Considerações Preliminares

Grades computacionais [VBR06] são sistemas de software que permitem o compartilhamento de recursos computacionais geograficamente distribuídos de modo transparente a seus usuários para a execução de aplicações. Estes sistemas devem considerar estratégias necessárias para permitir que os serviços que eles provêem continuem sua operação mesmo na presença de algum tipo de indisponibilidade nos membros da grade.

As aplicações para grades tipicamente requerem ou produzem grandes quantidades de dados. Dados de saída de uma aplicação podem ser utilizados por diversas outras aplicações criando a necessidade de um espaço de armazenamento compartilhado. Este espaço deve ser gerenciado para melhorar o armazenamento.

Os atuais sistemas de armazenamento de dados em grades computacionais utilizam sistemas gerenciadores de réplicas. Estas infra-estruturas são baseadas em máquinas dedicadas diferentemente do que acontece com grades oportunistas.

Grades oportunistas [VBR06] estão compostas por máquinas compartilhadas, que tipicamente possuem quantidades significativas de espaço livre em disco que são cedidas voluntariamente por seus usuários. Ao combinar este espaço livre em disco de algumas dezenas ou centenas de máquinas, podemos facilmente obter Petabytes de espaço de armazenamento.

As máquinas que são cedidas pelos usuários podem ser muitas vezes desligadas, por exemplo, em períodos noturnos ou fins de semana. Um sistema de armazenamento distribuído que utilize estas máquinas deve garantir a disponibilidade dos dados neste ambiente altamente dinâmico, utilizando estratégias como, por exemplo, replicação e/ou codificação dos dados armazenados. Estes sistemas devem ser auto-organizáveis e altamente escaláveis, devido a esse ambiente altamente dinâmico.

Para aproveitar os recursos de armazenamento em grades oportunistas foi projetado o middleware OppStore [dC08], que possibilita usar o espaço não dedicado e disponível das máquinas de grades oportunistas, mas este middleware não realiza o tratamento para a perda de dados como consequência da saída das máquinas que os armazenam devido às falhas apresentadas.

Levando em consideração o middleware OppStore [dC08], implementamos mecanismos para o tratamento de falhas apresentadas em nós da grade que comprometem o armazenamento dos dados, para assim permitir que os arquivos armazenados continuem disponíveis.

Existem vários trabalhos relacionados na área de armazenamento distribuído de dados baseados em sistemas peer-to-peer que procuram prover um serviço de armazenamento. Sistemas peer-to-peer [ATS04, LCP+05] lidam com populações variáveis de membros e se adaptam bem às falhas. Estes sistemas dão uma visão geral dos abordagens adotados para o tratamento de falhas. CFS [DKK+01], PAST [DR01] e OceanStore [KBC+00], por exemplo, são sistemas para armazenamento de dados distribuídos que estão baseados em sistemas peer-to-peer e que implementam estratégias de replicação dos dados armazenados para prover disponibilidade dos dados frente a falhas que podem acontecer com os nós do sistema.

1.2 Motivação

A questão do gerenciamento de arquivos armazenados é um ponto de grande importância quando é considerado um sistema que provê operações para o armazenamento dos mesmos. Neste caso, a medida que os nós deixam o sistema, parte dos fragmentos de arquivos armazenados são perdidos e precisam ser substituídos para garantir a disponibilidade dos arquivos. Ante essa situação, é preciso projetar um mecanismo para a manutenção dos arquivos que seja escalável, assim como definir quando esse mecanismo deve ser executado.

Considerando a variação do número nós disponíveis, os dados que o sistema armazena deverão ficar disponíveis mesmo com a presença de falhas nos diferentes nós da grade. Para isso, o objetivo é desenvolver um mecanismo que permita a recuperação automática de fragmentos de arquivos perdidos durante a variação da quantidade de nós disponíveis no sistema e assim manter um nível de disponibilidade dos fragmentos que permitam tolerar as falhas no armazenamento distribuído dos arquivos.

1.3 Objetivos

O OppStore implementa operações para armazenamento e recuperação de dados utilizando os recursos computacionais disponibilizados em uma grade oportunista. Arquivos armazenados no OppStore são codificados em fragmentos redundantes, que são distribuídos em diferentes máquinas, e podem ser reconstruídos utilizando apenas um subconjunto destes fragmentos. Mas as máquinas componentes da grade podem falhar, ficar inacessíveis ou passar de ociosas para ocupadas inesperadamente, impedindo o acesso aos fragmentos nelas armazenados.

O OppStore não implementa uma estratégia para o tratamento de falhas que comprometem a disponibilidade dos fragmentos quando as máquinas saem da grade. Quando falhas acontecem nas máquinas da grade, os fragmentos nelas armazenados são perdidos e deverão ser substituídos para evitar uma possível perda dos arquivos armazenados.

O principal objetivo neste projeto é desenvolver um mecanismo que permita a recuperação automática de fragmentos de arquivos armazenados que ficaram perdidos, devido à saída imprevista de nós que compõem a grade sobre a qual funciona o middleware OppStore. O mecanismo proposto permite manter uma quantidade adequada de fragmentos que possibilita manter disponíveis os arquivos armazenados minimizando a carga extra gerada pela execução do mecanismo.

Avaliamos o comportamento do mecanismo no sistema de armazenamento de dados levando em consideração o mecanismo proposto e comparando com uma segunda abordagem estabelecida para o tratamento das falhas.

1.4 Organização do Trabalho

No Capítulo 2, apresentamos conceitos relacionados a redes peer-to-peer de compartilhamento de arquivos que são importantes ao mostrarmos como implementam suas funcionalidades de acordo com o tipo de estrutura na qual se organizam. O Capítulo 3 apresenta sistemas de armazenamento distribuído de dados que utilizam redundância de dados para incrementar sua disponibilidade. No Capítulo 4, apresentamos a estrutura e os componentes principais do OppStore assim como seu funcionamento durante o armazenamento e recuperação dos arquivos nele armazenados.

No Capítulo 5 foram apresentados os conceitos relacionados à tolerância a falhas, além de apresentar as propostas que foram desenvolvidas para o tratamento dos dados na presença de falhas. No Capítulo 6, apresentamos a implementação do mecanismos propostos. No Capítulo 7, apresentamos as considerações e experimentos que foram realizados para a avaliação dos mecanismos propostos. No Capítulo 8, apresentamos alguns trabalhos relacionados com o nosso trabalho. Finalmente, no Capítulo 9 discutimos algumas conclusões obtidas neste trabalho.

Capítulo 2

Conceitos Gerais

Sistemas peer-to-peer [ATS04, LCP+05] são sistemas compostos por um conjunto de máquinas interconectadas constituindo uma rede sobreposta com o objetivo de compartilhar recursos computacionais. Estes sistemas oferecem autonomia aos seus participantes, possibilitando que entrem e saiam da rede de acordo com seu interesse e disponibilidade, mantendo um nível de desempenho aceitável nos serviços que provê, e assim, os usuários não percebam a perda dos serviços frente as possíveis falhas no sistema.

A seguir apresentamos conceitos relacionados com redes peer-to-peer de compartilhamento de arquivos, importantes ao mostrarmos como implementam suas funcionalidades e o tratamento de falhas de acordo com o tipo de estrutura na qual se organizam. Apresentamos também o Pastry [RD01], infra-estrutura que provê substrato para a construção de aplicações peer-to-peer de compartilhamento de arquivos, usada para o roteamento das mensagens gerenciadas pelo OppStore [dC08] apresentado no capítulo 4.

2.1 Redes peer-to-peer

Um sistema rede peer-to-peer [ATS04, LCP+05] é uma rede composta por um conjunto de máquinas interconectadas capazes de se auto-organizar constituindo uma rede sobreposta com o propósito de compartilhar recursos computacionais (conteúdo, recursos de armazenamento, largura de banda).

"Os Sistemas Peer-to-Peer são sistemas distribuídos consistindo de nós interconectados capazes de se auto organizar em topologias constituindo um rede sobreposta com o propósito de compartilhar recursos tais como conteúdo, ciclos de CPU, armazenamento e largura de banda, capazes de se adaptarem a falhas e acomodar populações variáveis de nós enquanto mantém conectividade aceitável e desempenho, sem precisar da intermediação ou apoio de uma entidade central" [ATS04].

Sistemas peer-to-peer lidam bem com grupos pequenos quanto com grupos grandes de participantes. Esta característica possibilita que ao incrementar os membros da rede, seja também possível incrementar a disponibilidade dos conteúdos.

Uma aplicação importante dos sistemas peer-to-peer é a distribuição de conteúdo. Nestas aplicações podemos identificar duas fases completamente distintas, a localização dos arquivos e a transferência dos mesmos. Como a função principal dos mesmos é a transferência de arquivos, esta função é realizada direitamente entre o origem e o destino. Quanto maior número de usuários, maior a capacidade do sistema para armazenar e transferir arquivos.

Considerando as funcionalidades dos membros da rede, um sistema peer-to-peer é puro quando todos seus membros têm a mesma funcionalidade, tanto de cliente quanto de servidor. Assim também, os sistemas podem ser híbridos quando consideram algum tipo de agrupamento para a realização das suas tarefas.

Sistemas peer-to-peer podem apresentar arquitetura centralizada ou descentralizada. Em sistemas peer-to-peer de arquitetura centralizada existe uma base centralizada com a localização dos recursos do sistema. Uma vantagem disso é que temos a garantia de que os dados que estão disponíveis em algum nó da rede serão encontrados, além de que as buscas são mais facilmente implementadas. A principal desvantagem é que existe um único ponto central de falha.

Sistemas peer-to-peer de arquitetura descentralizada não incluem um servidor com informações centralizadas. Cada membro num sistema peer-to-peer de arquitetura descentralizada tem funcionalidades tanto de cliente quanto de servidor. Dado que as funcionalidades do sistema estão distribuídas de maneira descentralizada, estes sistemas têm a capacidade de se adaptar às falhas e à variabilidade dos membros que a constituem, mantendo um desempenho aceitável dos serviços prestados. Como não existe uma coordenação central na distribuição de conteúdo, precisa-se a participação efetiva dos membros para tarefas tais como busca por outros nós, localização de conteúdo ou roteamento de mensagens.

Quando falamos de sistemas peer-to-peer de arquitetura descentralizada, podemos citar as redes estruturadas e as não-estruturadas [ATS04, LCP+05]. As redes peer-to-peer estruturadas seguem uma estrutura definida para a organização dos nós. A localização do conteúdo é dependente da topologia e há uma relação direta entre um conteúdo e o nó que o armazena. No caso das redes peer-to-peer não-estruturadas, a organização dos nós não têm uma estrutura definida. A localização dos nós não depende da topologia, e requer a implementação de sistemas mais eficientes de busca que os usados no caso das redes estruturadas.

2.1.1 Redes Peer-to-Peer Não-estruturadas

Uma rede peer-to-peer não estruturada [ATS04, LCP+05] é uma rede sobreposta onde os nós se organizam de maneira aleatória e não tem uma estrutura predefinida. A medida que os nós ingressam e saem da rede sobreposta se estabelecem conexões com outros nós arbitrários.

Para realizar as buscas o sistema faz uso de mecanismos tais como inundação (*flooding*) [JJ05], caminhos aleatórios ou expansão de busca controlada por TTL (*Time To Live*) que limita a profundidade da busca.

As buscas por inundação são um mecanismo onde cada nó que recebe uma mensagem durante uma requisição e avalia a consulta (*query*) localmente sobre seu próprio conteúdo. Se não encontra o recurso procurado então o nó reenvia a mensagem para todos seus nós vizinhos. Este tipo de busca por inundação é ineficiente, dado que consultas por conteúdos que não estão amplamente replicados devem ser enviados para uma grande quantidade de nós no sistema. Como os nós na busca são escolhidos de maneira aleatória, tem desvantagem pela dificuldade em encontrar os dados desejados.

Outro mecanismo de busca é o uso de caminhos aleatórios, onde os nós que recebem a mensagem de requisição não reenviam a mensagem de busca para todos seus nós vizinhos. As mensagens são enviadas para um nó eleito de forma aleatória. Neste tipo de busca o alcance máximo da mensagem está limitado por um mecanismo análogo ao de TTL (*Time To Live*), que restringe o número de saltos que a mensagem pode realizar entre os diferentes nós. Dado que a mensagem não é reenviada para todos os vizinhos de um nó na rede, este mecanismo não satura a rede como o mecanismo de inundação, mas a probabilidade de encontrar o recurso é menor que o mecanismo de inundação.

As vantagem das redes não-estruturadas são que estas não possuem um ponto central de falha e acomodam facilmente uma população de nós altamente transiente. As desvantagens são que não oferecem garantia quanto a acessibilidade dos dados, além de oferecer dificuldade em encontrar o arquivo desejado.

Entre os diferentes sistemas baseados em redes não-estruturadas estão o Napster [dpN], Gnutella [dpG], freeNet [CSWH01], Kazza [dpK].

Napster [dpN] foi o primeiro sistema pensado para o compartilhamento de arquivos numa rede peer-to-peer entre vários usuários direitamente entre a fonte e o destino. Para fazer a descoberta e localização dos arquivos utiliza um servidor central.

Quando um usuário deseja se conectar ao sistema, ele deve se conectar ao servidor central e fornecer a lista dos arquivos que vai compartilhar. Algum usuário, que procura por algum arquivo, iniciará uma busca que será respondida pelo servidor central. Após obter a resposta sobre a lo-

calização dos arquivos, a transferência dos arquivos é realizada entre os nós origem e destino. A centralização das buscas por arquivos procurados no servidor central gera um ponto único de falha no sistema.

Gnutella [dpG] se caracteriza por seguir uma arquitetura totalmente descentralizada. A diferença do Napster não possui um diretório centralizado onde fazer as buscas, para isso requer de mecanismos de busca tais como a busca por inundação com a participação dos nós componentes do sistema.

Gnutella é considerado um sistema peer-to-peer puro, os nós componentes do sistema têm a mesma funcionalidade e podem atuar tanto como cliente quanto como servidor. Para responder as buscas, os nós possuem uma interface servidora e para realizar suas próprias buscas, os nós possuem uma interface cliente. Como resultado da descentralização do sistema, Gnutella é tolerante a falhas porque as operações da rede não serão interrompidas com a saída de alguns nós do sistema.

Para localizar um arquivo dado, uma busca é feita usando tipicamente o mecanismo baseado em inundação controlado por TTL. Este controle consiste em fixar o número de saltos (hops) que a mensagem vai se propagar dentro dos nós no sistema durante uma requisição. Apesar de ser altamente tolerante a falhas, o mecanismo de busca baseado em inundação do Gnutella pode sobrecarregar os nós, limitando a escalabilidade do sistema.

FreeNet [CSWH01] é uma implementação de redes peer-to-peer na qual são feitas consultas para o armazenamento e a recuperação de arquivos de dados. Os dados são identificados por chaves independentes da localização dos mesmos. Uma das principais características de FreeNet é a possibilidade de armazenamento anônimo de informações. Esta característica consiste em não permitir a identificação do autor ou aquele que publica o objeto, a identidade do nó que o armazena, assim como as informações dos objetos e os detalhes das requisições para a recuperação dos mesmos.

FreeNet utiliza palavras chave e texto descritivo para identificar objetos. Cada nó contém informação dos seus vizinhos próximos para oferecer principalmente privacidade no armazenamento. As requisições possuem um identificador e sua propagação é controlada por um mecanismo de análogo ao mecanismo TTL, que permite limitar o número de nós a visitar. Na chegada de uma nova requisição cada nó decide para onde encaminhá-la utilizando para isso a informação local do nó. O processo continua até alcançar o objetivo ou exceder o limite de nós a visitar.

2.1.2 Redes Peer-to-Peer Estruturadas

Uma rede peer-to-peer estruturada [ATS04, LCP+05] é uma rede sobreposta sobre nós onde a estrutura de organização dos nós está bem definida. A conexão entre os nós é feita usando como

base o conceito de identificador de nó, obtido aplicando uma função de espalhamento segura sobre alguma informação única do nó.

As redes Peer-to-Peer estruturadas utilizam tabelas de espalhamento distribuídas (DHT) como substrato. Tabelas de espalhamento distribuídas (*Distributed Hash Table*) [RD01, SMK+01] associam tuplas (chave, dado). O valor da chave é usado para identificar algum dado armazenado, logo que chaves e dados foram distribuídos entre os nós do sistema para seu armazenamento.

Dado que a distribuição dos nós está predefinida e o os locais de armazenamento dos dados não são determinados de maneira aleatória, a localização dos dados é feita de maneira determinística, realizando consultas mais eficientes usando mapeamento entre a chave e os locais de armazenamento associados à chave procurada.

Entre as vantagens das redes peer-to-peer estruturadas, estão a garantia de acessibilidade aos dados e o balanceamento de carga automático. Durante a saída ou ingresso de um nó se realiza uma auto-organização dos nós que compõem o sistema. Entre as desvantagens, vemos que é difícil manter a estrutura requerida, além disso, o nó que faz a requisição precisa conhecer a chave do objeto procurado e a manutenção em populações altamente transientes é difícil.

Entre algumas das implementações que fazem possível a criação das redes peer-to-peer estruturadas podemos citar: O Pastry [RD01], Tapestry [ZKJ01], chord [SMK+01], CAN [RFH+01], Kademlia [MM02], Viceroy [MNR02].

Chord [SMK+01], assim como o Pastry, disponibiliza uma base para a construção de aplicações peer-to-peer oferecendo a funcionalidade de uma tabela de espalhamento distribuída (DHT). Cada nó recebe um identificador aleatório de um espaço de identificadores de m bits e os organiza em um anel lógico. O número m de bits é normalmente de 120 ou 160 dependendo da função de espalhamento usada.

Uma entidade com uma chave k no espaço de identificadores, cai sobre cuidado do nó que tenha o menor identificador $id \geq k$. Cada nó Chord mantém uma apontador para os n nós sucessores imediatos e uma tabela de derivação (**finger table**) com até m entradas que contém apontadores para outros nós(logaritmicamente espalhados no anel) para tornar as buscas mais eficientes. A busca inicia em um nó e percorre o anel lógico até encontrar o nó cuja lista de sucessores contém o nó imediatamente sucessor ao identificador procurado.

Cada nó Chord só precisa informação de roteamento de um pequeno número de nós. Dado que esta informação é distribuída, um nó resolve a função de espalhamento (Hash) comunicando-se com outros poucos nós.

Numa rede com N nós, cada nó precisa salvar informação de $O(logN)$ outros nós e o lookup também requer $O(log_2 N)$ mensagens; a atualização da informação quando um nó deixa o entra na rede requer $O(log^2 N)$ mensagens.

CAN [RFH+01] (Content-Addressable Network) usa um sistema baseado em coordenadas geométricas em que cada nó é responsável por um hipercubo.

O espaço de coordenadas é inteiramente lógico e serve para implementar a identificação de nós e sua localização via tabelas de roteamento distribuído. Cada nó é responsável por uma zona do espaço, que é dinamicamente determinada, e mantém uma tabela de roteamento com o endereço IP e as coordenadas de cada um de seus vizinhos no espaço.

O protocolo de busca emprega pares (chave, objeto) para mapear um ponto P no espaço de coordenadas usando uma função de espalhamento uniforme e coloca estas coordenadas nas mensagens. Uma mensagem é roteada para o nó mais próximo das coordenadas. Dado que múltiplos nós são responsáveis por um objeto, a falha de algum nó não causa uma falha global, e quando há falha se executa uma retentativa. A divisão geométrica dos nós é vantajosa pois, quando um nó entra na rede somente um número fixo de nós é afetado, diferentemente de outros algoritmos tais como Pastry, Tapestry e Chord, onde um número proporcional à população da rede é afetado.

A seguir ampliamos a descrição do Pastry porque ele é usado como base para roteamento das mensagens para o armazenamento e recuperação de dados no middleware OppStore que é parte de nosso estudo.

2.2 Pastry

Pastry [RD01], desenvolvido por Antony Rowstron da Microsoft Research Ltd. e Peter Druschel da Rice University, é uma infra-estrutura que provê substrato para a construção de aplicações peer-to-peer incluindo armazenagem e compartilhamento de arquivos, comunicação em grupo e serviços de nomes.

A cada nó no Pastry é atribuído um único identificador de nó num espaço de identificadores de 128 bits. O identificador de nó é usado para indicar a posição do nó num espaço circular de identificadores com valores entre 0 e $2^{128} - 1$. Estes identificadores de mensagens e de nós são representadas como uma seqüência de dígitos com base 2^b, onde b é um parâmetro de configuração (valendo normalmente 4). Os identificadores podem ser gerados a partir de uma função de espalhamento da chave pública ou do endereço IP do nó e são atribuídos de maneira aleatória quando um

nó ingressa ao sistema.

Cada nó Pastry mantém uma tabela de roteamento, um conjunto de nós vizinhos (nós fisicamente próximos) e um conjunto de folhas. A tabela de roteamento contém $\lceil log_{2^b} N \rceil$ filas com $2^b - 1$ entradas cada uma (de 0 a $(2^b - 1)$ colunas). Cada entrada na tabela de roteamento contém o endereço IP de um dos muitos potenciais nós que contém identificador com um prefixo adequado. O nó escolhido é aquele que se encontra a menor proximidade.

O conjunto de vizinhos $|M|$ contém os endereços IP dos M nós mais próximos ao nó local e é usado para manter propriedades de localização. O conjunto de folhas $|L|$ é conjunto de nós composto pelos $|L|/2$ nós mais próximos maiores e os $|L|/2$ mas próximos menores do que o no presente. O conjunto de folhas é usado durante o roteamento das mensagens.

2.2.1 Roteamento de mensagens

Pastry realiza o roteamento de uma mensagem até o nó que possui o identificador mais próximo à chave dada. Assim, em cada passo um nó encaminha a mensagem para o nó que compartilha com a chave um prefixo que é pelo menos um dígito ou b bits a mais que o prefixo que a chave compartilha com o nó presente.

Assumindo N nós, Pastry pode rotear uma mensagem até o nó com identificador mais próximo à chave dada em menos do que $\lceil log_{2^b} N \rceil$ passos numa operação normal, onde b é um parâmetro de configuração normalmente seu valor é 4. Isto porque a cada passo, o prefixo em comum entre a chave da mensagem e o identificador do nó terá um dígito a mais.

A figura 2.1 mostra como é feito o roteamento das mensagens no Pastry. Dada uma mensagem com uma chave X e sendo A o identificador do nó que processa o algoritmo, o algoritmo para roteamento de uma mensagem é o seguinte:

- Se X é um identificador limitado pelo maior e menor elemento do conjunto de folhas, então envie a mensagem para o identificador de nó N pertencente ao conjunto de folhas tal que a distância numérica entre X e N seja mínima.
- Caso contrário, acesse a posição da tabela de roteamento formada pelos n primeiros dígitos compartilhados entre X e A mais o dígito $n+1$ de X, e envie a mensagem para este identificador de nó.
- Caso esta posição na tabela seja nula, envie a mensagem para o identificador de nó Z pertencente à união do conjunto de folhas, tabela de roteamento e o conjunto de vizinhos, tal que a

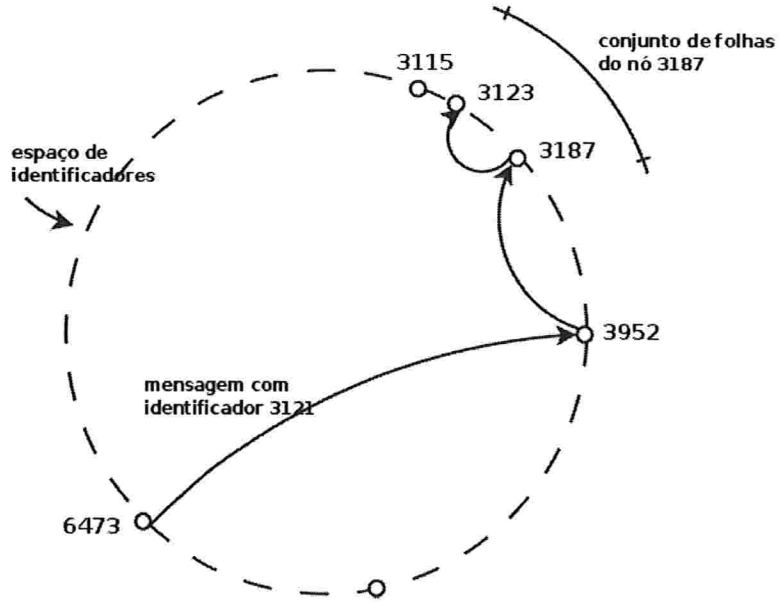

Figura 2.1: Roteamento de mensagens no Pastry [RD01]

distância entre A e X e o número de dígitos primeiros compartilhados entre Z e X seja maior ou igual aos de A e X.

2.2.2 Ingresso de um novo nó

Quando um nó X ingressa no sistema, X contacta algum nó mais próximo A e pede para esse nó A enviar uma mensagem especial de *join* com a chave X. Esta mensagem é encaminhada para o nó Z que tem o identificador numericamente mais próximo a X.

Os nós que receberam a mensagem de *join* (A,Z e todos os outros nós no caminho de A para Z), enviam suas tabelas de roteamento para o novo nó X.

O novo nó X analisa as tabelas recebidas e constrói sua própria tabela de roteamento, depois informa sua chegada aos que precisam ser avisados. O nó X obtém um conjunto de folhas do nó Z, o conjunto de vizinhos do nó A, e a i-éssima fila da tabela de roteamento do i-éssimo nó encontrado de A a Z.

2.2.3 Saída de um nó

Um nó está perdido (saiu fora da rede), quando não responde à tentativa de comunicação de algum outro nó.

- Se a perda é de detectada na tabela de roteamento num identificador de nó com fila l e

coluna d, então o nó pergunta ao nó ${R_l}^i$,$i \neq d$ sob a posição ${R_l}^d$. caso não seja retornado um identificador de nó para ser colocado na posição, então é feita a mesma pergunta para ${R_{l+1}}^i$,$i \neq d$, e assim por diante.

- Se uma perda é detectada no conjunto de folhas, o nó envia uma mensagem requisitando o conjunto de folhas para o índice extremo do lado do perdido. Recebendo-a, o nó decide qual identificador de nó será adicionado em seu próprio conjunto.
- Se a perda é detectada no conjunto de vizinhos, o nó pede a um outro nó do conjunto sua tabela de vizinhos e a analisa, de acordo com as distância, qual deve substituir ao nó que falhou.

2.3 Resumo

Neste capítulo vimos as redes peer-to-peer de compartilhamento de arquivos, importantes ao mostrarmos como implementam suas funcionalidades de acordo com o tipo de estrutura na qual se organizam.

Pastry fornece uma estrutura que implementa a funcionalidade de uma tabela de espalhamento distribuída (DHT) e que é importante para o roteamento das mensagens gerenciadas pelo OppStore apresentado no capítulo 4.

No próximo capítulo apresentamos sistemas de armazenamento distribuído de dados que fazem uso das diversas infra-estruturas baseadas em tabelas de espalhamento distribuídas (DHT).

Capítulo 3

Armazenamento Distribuído de Dados

Armazenamento distribuído de dados consiste na capacidade de um sistema permitir o armazenamento de informação em diferentes máquinas interligadas (nós) que estão localizadas geograficamente distribuídas.

Existem vários trabalhos relacionados na área de armazenamento distribuído de dados sobre sistemas peer-to-peer e grades computacionais que procuram prover um serviço de armazenamento.

CFS [DKK+01], PAST [DR01], são sistemas peer-to-peer para armazenamento de dados que estão construídos baseados em uma infra-estrutura DHT. Estes sistemas, implementam uma estratégia de replicação de dados armazenados para melhorar a disponibilidade dos mesmos.

OceanStore [KBC+00] provê armazenamento consistente sobre uma infra-estrutura não confiável, implementa uma estratégia de replicação de fragmentos de arquivos de dados. Codifica os arquivos de dados em fragmentos redundantes e os distribui entre várias máquinas que compõem o sistema.

A seguir apresentamos as principais características dos sistemas de armazenamento distribuído de dados acima mencionados com ênfase nas estratégias usadas para manter a disponibilidade dos dados.

3.1 CFS

O Cooperative File System (CFS) [DKK+01] é um sistema peer-to-peer para armazenamento de dados distribuídos que provê serviços de armazenamento e recuperação de arquivos, além de balanceamento de carga dos dados. Seu desenvolvimento está inspirado nos sistemas peer-to-peer descentralizados Napster [dpN], Gnutella [dpG] e FreeNet [CSWH01].

Um arquivo de sistema em CFS existe como um conjunto de blocos de dados distribuídos sobre o conjunto de servidores CFS disponíveis. O software cliente CFS interpreta o conjunto de blocos armazenados como um arquivo de sistema e meta-dados, e o apresenta como uma interface ordinária do arquivo de só leitura para as aplicações.

Um usuário pode inserir dados apenas em seu próprio sistema de arquivos, através da atualização de um bloco especial chamado bloco raiz. Este bloco contém o hash do conteúdo dos demais blocos, sejam eles blocos ou arquivos. Ele é assinado digitalmente com a chave privada do usuário para verificar integridade e autenticidade.

Os blocos são de tamanho fixo e recebem um identificador único. Este identificador é usado pelo sistema para determinar os servidores responsáveis pelo seu gerenciamento. O identificador de cada bloco é dado pelo hash do conteúdo do bloco com exceção do bloco raiz cujo identificador é a chave pública do usuário. Cada máquina recebe um identificador no mesmo espaço de identificadores.

A arquitetura do CFS do lado cliente, como mostra a figura 3.1, apresenta três camadas, a camada de arquivos do Sistemas (File System), a camada DHash e a camada Chord. A camada de arquivos de sistema utiliza a camada DHash para recuperar blocos de dados.

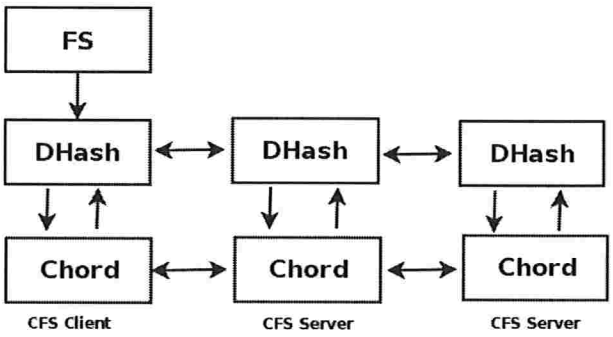

Figura 3.1: Estrutura de software do CFS [DKK+01]

A camada DHash representa blocos trazidos para o cliente, distribui os blocos entre os servidores, mantém cache e replicação de blocos de dados. A camada DHash utiliza a Chord para localizar o servidor que contém o bloco desejado além de prover balanceamento de carga para arquivos populares.

Chord [SMK+01] oferece a funcionalidade de uma tabela de espalhamento distribuída (DHT) e também uma base para a construção de aplicações peer-to-peer. Utiliza um espaço de identificadores de m bits para designar identificadores aleatórios aos nós e os organiza em um anel lógico.

Do lado servidor, apresenta duas camadas de software, a camada de armazenamento DHash e

a camada Chord. A camada DHash do lado servidor é responsável pelo armazenamento de blocos codificados mantendo níveis adequados de replicação sobre servidores quando ingressam e saem do sistema, além de manter *caching* de blocos populares.

Em cada servidor é armazenada uma tabela de roteamento contendo a ubicação de outros $O(logN)$ servidores, onde N é o número total de servidores; este também é o número de mensagens que um servidor troca com os outros servidores para realizar o armazenamento e recuperação de dados.

A camada Dhash em um servidor sucessor de bloco é responsável pela replicação desse bloco, garantindo que todas as k cópias dos seus servidores sucessores tenham uma cópia do bloco o tempo todo. Esta camada posiciona uma réplica do bloco de dados sobre os k servidores imediatos contínuos do sucessor do bloco seguindo a distribuição no anel Chord. Se o servidor cai, o novo sucessor assume a responsabilidade pelo bloco.

CFS não utiliza algoritmos para codificar os blocos de dados, dado que CFS assume que o espaço de armazenamento não é altamente restritivo. A replicação faz que um cliente possa selecionar provavelmente a réplica mais rápida disponível. O resultado da busca no anel Chord pelo identificador de um bloco é a identidade do servidor que precede ao identificador. O cliente pergunta a este predecessor por sua lista de sucessores. Esta lista incluirá a identidade dos servidores que contém as réplicas do identificador de bloco, além das latências entre o predecessor e esse servidor. O cliente então procura a réplica no servidor com a mais baixa latência reportada.

DHash evita a sobrecarga dos servidores que contêm os blocos dados mais populares deixando cópias cache dos blocos procurados nos servidores que visitou durante a busca por um bloco. Isto permite que se um novo nó precisa de um desses blocos populares, ele pegará provavelmente uma destas réplicas por estar estas localizadas nos nós com identificadores mais próximos no anel Chord.

Dado que as capacidades de armazenamento e rede dos servidores nem sempre são as mesmas, Chord usa a idéia de que um servidor real atue como múltiplos servidores virtuais. O protocolo CFS opera no nível de servidores virtuais. Um servidor virtual usa um identificador de bloco derivado do hashing do endereço IP do no servidor e o índice do servidor virtual dentro do servidor real. O número de servidores virtuais é controlado pelo administrador e dependerá das capacidades de armazenamento e de rede dos servidores.

Usar servidores virtuais potencialmente poderia incrementar o número de saltos na busca Chord. CFS evita isso permitindo que servidores virtuais sobre o mesmo servidor físico examinem outras tabelas de roteamento. CFS pode variar o número de servidores virtuais por cada servidor real. Na presença de uma alta carga, um servidor real pode apagar alguns dos seus servidores virtuais e em

baixa carga poderiam se criar servidores virtuais adicionais.

3.2 PAST

O PAST [DR01] é um sistema para armazenamento distribuído de arquivos baseado no Pastry [RD01], onde os arquivos são associados a um identificador(ID) gerado quando o arquivo é inserido na primeira vez. Os arquivos podem ser compartilhados pela distribuição dos identificadores. Diferentemente de CFS [DKK+01], PAST armazena arquivos por inteiro nos nós destino ao invés de blocos destes arquivos.

Durante a operação de inserção, PAST armazena o arquivo nos k nós cujos identificadores são numericamente mais próximos aos 128 bits do identificador do arquivo a ser armazenado. O valor de k dependerá da disponibilidade necessária do arquivo relacionado ao índice de falhas esperadas de nós individuais. Os arquivos mais populares podem precisar ser mantidos em vários nós para balancear as consultas para o arquivo e assim minimizar a latência da rede.

Para recuperar um arquivo, o cliente deve conhecer o identificador do arquivo e se é possível uma descrição do arquivo. PAST não provê facilidade para realizar a busca ou distribuição de chaves, utiliza o Pastry para fazer o roteamento de mensagens, o qual garante que as requisições dos clientes são encaminhadas até os nós adequados.

Dado que os arquivos são armazenados inteiros e são de tamanho variável, um dos k nós responsáveis por esse arquivo pode não ter espaço suficiente para armazenar uma réplica. Quando isso acontece o nó que não pode armazenar a réplica procura outro dentre os seus vizinhos na rede que ainda não estejam armazenando uma cópia do arquivo. Um apontador é criado para esse nó vizinho de forma que as requisições pelo arquivo sejam encaminhadas para ele. O objetivo é balancear o espaço ainda disponível entre um conjunto de nós vizinhos.

O balanceamento do espaço disponível é motivado pelas diferenças nos tamanhos dos arquivos gerenciados por cada nó, a heterogeneidade dos recursos de armazenamento e a variação dos identificadores gerenciados por cada nó. Caso não seja possível inserir as k réplicas mesmo com um desses apontadores, a operação é rejeitada e um novo conjunto de nós é sorteado. Assim, procura-se o espaço global ainda disponível.

PAST permite que um nó que não é um dos k nós numericamente mais próximos do identificador do arquivo armazene um arquivo se está dentro do conjunto de folhas de um desses k nós. Este procedimento é chamado diversificação de réplicas (*replica diversion*), e seu propósito é balancear o espaço disponível dos nós dentro de um conjunto de folhas. A diversificação de cópias é realizada

quando o conjunto de folhas de um nó atingiu sua capacidade total armazenamento.

No momento que um nó não pode armazenar uma cópia local do arquivo inicia o processo de diversificação de réplicas. O processo começa com um nó A que escolhe um nó B no seu conjunto de folhas que não seja um nó que possua um identificador que se encontre entre os k mais próximos do identificador do arquivo e não tenha já uma cópia armazenada do arquivo. O nó A pede ao nó B armazenar uma cópia, se é permitido, o nó A insere na sua tabela uma entrada apontando ao nó B e faz pública a mensagem de recepção.

Para o tratamento dos arquivos populares PAST faz cópias adicionais em uma cache. Devido a isso, existe temporariamente mais do que k cópias (cópias cache) para um mesmo arquivo. As cópias cache são criadas em nós que tinham sido contatados pelo Pastry durante a requisição por algum identificador de arquivo, seja na inserção ou na recuperação de arquivos. Cópias próximas fisicamente aos nós requisitantes podem ser criadas e mantidas nos nós próximos adjacentes, diminuindo as latências de acesso aos dados. Se um arquivo é popular entre um aglomerado de clientes, então é vantajoso criar uma cópia desse arquivo perto desse aglomerado. As cópias cache de arquivos que deixam de ser populares com o tempo são reemplazadas pelas de outros arquivos.

3.3 OceanStore

OceanStore [KBC+00] é uma infra-estrutura desenvolvida para abarcar e prover acesso contínuo a informação persistente sob uma infra-estrutura não confiável, onde os recursos de armazenamento (servidores de armazenamento) podem sair do fora do sistema por alguma falha.

A unidade fundamental para armazenamento no OceanStore é o objeto persistente. Cada objeto(arquivo) é identificado por um identificador global único(GUID). OceanStore replica e armazena os objetos sob múltiplos nós, esta replicação de arquivos nos seus servidores permite a disponibilidade dos dados na presença de possíveis falhas nos mesmos. As réplicas são independentes do nó onde reside em algum momento.

OceanStore segue uma política de *caching promiscuous*, que permite que sejam criadas cópias cache dos dados em qualquer servidor em qualquer momento.

A informação armazenada no OceanStore é modificada utilizando atualizações dos arquivos. Atualizações dos arquivos contêm informações sobre as mudanças que devem ser aplicadas aos objetos, onde toda atualização do objeto cria uma nova versão do mesmo.

Neste ponto existem objetos em estado ativo e arquivado. Um objeto em estado ativo é a última

versão dos dados junto com uma estrutura para atualização. Um objeto em estado arquivado representa uma versão permanente dos dados e só de leitura. No momento que uma aplicação deseja armazenar um arquivo de forma permanente, este arquivo deve ser codificado utilizando código de rasura e depois estes fragmentos serão distribuídos entre os nós servidores que formam parte do sistema OceanStore.

A localização de um objeto é feita através de dois mecanismos. O primeiro mais rápido é o algoritmo probabilístico usando uma versão modificada dos filtros de Blomm [Blo], que permitem realizar testes de inclusão de um membro em um conjunto, com a lista de membros armazenada de forma bastante compacta. Existe uma pequena possibilidade de retornar falsos positivos neste teste de inclusão. OceanStore mantém uma lista de filtros de Bloom contendo a lista de objetos armazenados no próprio nó e em nós vizinhos.

Caso o sistema não tenha éxito na busca usando os filtros de Bloom, então é usado uma segunda estrutura denominada *árvores de Plaxton* [PRR99]. Nesta estrutura cada nó servidor recebe um único identificador aleatório, estes identificadores são usados para construir múltiplas árvores com enlaces vizinhos, cada uma com diferente raiz.

Como resultado, os enlaces vizinhos podem ser usados para ir de um nó a qualquer outro resolvendo os endereços de cada nó por vez, primeiro no nível um da árvore, depois no nível dois e assim em diante. A partir destas árvores é possível localizar o nó responsável por um identificador em $(logN)$ passos, onde N é o número de nós servidores do sistema.

Um ponto de possíveis falhas nesta estrutura são as raízes dos objetos, estas falhas são tratadas espalhando os identificadores dos objetos com um número pequeno de diferentes valores de salto, o mapa resultante para vários diferentes nós raízes conseguem ganho de redundância.

Para incrementar a redundância a estrutura de localização do OceanStore complementa os enlaces básicos dos nós raízes com enlaces adicionais para seus vizinhos. Os enlaces aos vizinhos são monitorados e reparados quando as falhas acontecem.

3.4 Neutralizer

Neutralizer [YDL09] é um detector de falhas auto-configurável que foi projetado para minimizar os custos associados à manutenção de dados distribuídos na presença de falhas em nós do sistema.

O objetivo da auto-configuração é manter limiares de tempo que permitam manter um equilíbrio no número de réplicas disponíveis no sistema. O objetivo é que o excesso de réplicas criadas devido

a réplicas que o sistema considera perdidas, mas voltam depois ao sistema (falsos positivos), seja minimizada com o número de réplicas perdidas devido a réplicas que o sistema considera disponíveis, mas não estão (falsos negativos).

Para diferenciar as falhas transientes das falhas persistentes, Neutralizer utiliza detectores baseados em tempos. Os detectores de tempo de falsos positivos incrementam a quantidade de réplicas no sistema enquanto os detectores de tempo de falsos negativos as reduzem.

Neutralizer utiliza um detector de tempo com um valor agressivo T_1 e um detector de tempo com um valor conservativo T_2, onde $T_1 < T_2$. O valor de T_1 é usado para considerar réplicas que possivelmente estejam com problema de indisponibilidade e por isso são suspeitas de estar perdidas, mas sem excluí-las do sistema. O valor de T_2 é usado para determinar falhas permanentes e excluir as réplicas fora do sistema. Uma réplica é suspeita de ser perdida quando T_1 termina, mas poderá ser reintegrada no sistema antes do que T_2 termine.

Para manter o número de réplicas:

- O sistema primeiro estima o número objetivo de réplicas r_L que precisa para manter a disponibilidade desejada dos dados.
- O sistema mantém o nível de replicação, detectando o número de réplicas disponíveis (n réplicas vivas) para um dado particular.
- Quando n fica baixo de r_L, novas réplicas são geradas até que, ou r_L réplicas estão vivas e reconhecidas pelo detector ou todas as réplicas estão mortas e o objeto pode estar perdido.

Em um momento determinado, quando há r réplicas existentes, a probabilidade de que uma réplica viva seja considerada pelo detector é $1 - p_{FP}$, onde p_{FP} é a probabilidade da existência de falsos positivos. O número médio de réplicas extras $E[e]$ criadas quando o detector percebe que há menos réplicas do que r_L está dada por:

$$E[e] = \sum_{i=0}^{r_L-1} \binom{r}{i} (r_L - i)(1 - p_{FP})^i p_{FP}^{r-i} \quad (1)$$

Inicialmente $E[e] = r_L p_{FP}$ quando $r = r_L$. O índice de reparação cai rapidamente tanto como r se incrementa. A probabilidade que uma réplica considerada pelo detector esteja realmente viva é $1 - p_{NP}$, onde p_{NP} é a probabilidade da existência de falsos negativos. O número médio de réplicas perdidas $E[l]$ devido a falsos negativos está dado por:

$$E[l] = \sum_{i=0}^{r} i \binom{r}{i} p_{NP}^i (1 - p_{NP})^{r-i} = r_{p_{NP}} \quad (2)$$

Neste caso o índice de perdidas é incrementado com r. Assume-se que (1) é igual a (2) quando $r = r_0$. O valor significativo de réplicas que o sistema matém $E[r]$ em torno de r_0: r será diminuído

se $r > r_0$ para $E[e] > E[l]$, mas será incrementado se $r < r_0$ para $E[e] < E[l]$, além o valor de r varia em torno de r_0. O valor de r_0 é usado para estimar o valor de $E[r]$ nas seguintes situações:

1. $p_{FP} > p_{NP}$, neste caso r é incrementado inicialmente dado que o índice de recreação de réplicas extras é maior do que o índice de perdida de réplicas não reparadas($E[e] = r_L p_{FP} > E[l] = r_L p_{NP}$). Como r é incrementado, $E[e]$ começa a diminuir, enquanto $E[l]$ começa a aumentar, de forma que r finalmente se aproximaria de um equilíbrio médio $r_0 > r_L$.

2. $p_{FP} = p_{NP}$, neste caso o índice de re-criação de réplicas é igual ao índice de perdidas ($E[e] = E[l] = r p_{FP}$) resultando um equilíbrio médio. Os falsos positivos se cancelam com os falsos negativos.

3. $p_{FP} < p_{NP}$, neste caso r diminuirá inicialmente dado que o índice de re-criação é menor do que o índice de perdidas de réplicas ($E[e] = r_L p_{FP} < E[l] = r_L p_{NP}$). Como no primeiro caso, o sistema pára de perder réplicas como r diminui, e finalmente mantém um equilíbrio médio $r_0 < r_L$.

Neutralizer considera uma otimização para tentar minimizar a quantidade de réplicas que devem ser criadas quando as falhas acontecem.

3.5 Estratégias de Replicação

Dado que os sistemas peer-to-peer são formados por membros com presença altamente variável, para garantir a disponibilidade dos dados armazenados nos membros, o sistema requer a introdução de redundância dos dados armazenados. A forma mais simples de introduzir redundância é usar a replicação dos dados.

Em Napster e Gnutella o mecanismo de replicação é implícito, não possui uma estratégia de replicação. A replicação acontece quando um usuário descarrega um arquivo. Dado que o mecanismo de replicação nestes sistemas não está definido explicitamente, arquivos impopulares podem ficar indisponíveis.

CFS e PAST utilizam um sistema de replicação explícita onde os mecanismos de replicação estão definidos para ocultar as falhas que podem acontecer em nós do sistema e manter disponíveis os arquivos armazenados.

A replicação também nestes casos pode considerar a granularidade dos dados, nesse sentido temos granularidade no nível de blocos e no nível de arquivo completo. CFS utiliza granularidade de arquivos em blocos onde os arquivos são particionados em blocos de dados que são distribuídos para ser armazenados nos nós do sistema. Gnutella e PAST entre outros utilizam granularidade

no nível de arquivo completo armazenando os arquivos de tamanho completo nos diversos nós do sistema.

O custo de armazenar arquivos por inteiro pode ser custoso termos de espaço e tempo. O armazenamento de um arquivo como uma sequência de blocos tem a vantagem de que diferentes blocos de dados podem ser baixados de diferentes nós mais rapidamente. Como os blocos replicados são pequenos e de tamanho fixo, o custo de replicação pode ser menor do que usando arquivos inteiros.

Outros sistemas de armazenamento massivo como o OceanStore utilizam codificação de rasura (*erasure codes*) para melhorar a disponibilidade da replicação por blocos e por arquivos inteiros reduzindo assim os custos de armazenamento. No OppStore [dC08] parte de nosso trabalho é usado também codificação de rasura, neste caso o algoritmo de dispersão de informação (IDA).

Algoritmo de dispersão de informação (IDA)

O algoritmo de dispersão de informação [Rab89a], permite codificar um arquivo F de tamanho n em $m+k$ fragmentos, onde $m+k > n$. Existe um índice de codificação r tal que $r = n/m < 1$. O índice de codificação r incrementa o custo de armazenamento por um fator de $1/r$. O arquivo original pode ser reconstruído a partir de quaisquer m fragmentos.

O algoritmo requer o uso de operações matemáticas sobre um campo $GF(q)$, um campo finito de elementos, onde q é um número potências de p^x sendo p um número primo.

Escolhe-se um valor apropriado inteiro m tal que $n = m+k$ satisfaz $n/m \leq 1+\epsilon$, para um $\epsilon > 0$ especificado.

São gerados $m+k$ vetores linearmente independentes $a_i = (i, b_i, ..., b_i^{n-1})$ de tamanho m, onde $0 \leq i < n$. Os vetores são organizados numa matriz G definida como:

$$G = [a_0^T, a_1^T, ..., a_{m+k}^T]$$

O arquivo F é particionado em sequências F_i de tamanho m e são geradas n/m sequências codificadas S_i de tamanho $m+k$, onde:

$$S_i = F_i \times G$$

Os $m+k$ vetores codificados V_i, onde $0 \leq i < m+k$ estão definidos por:

$$V_i = (S_0[i], S_0[i], ..., S_{n/m}[i])$$

Para recuperar as sequências de informação originais F_i, tem se que recuperar m do total de fragmentos $m+k$. São construídas uma matriz G' que contém elementos relativos aos fragmentos

recuperados, uma seqüência S'_j equivalente às seqüências codificadas S_i que contêm os componentes dos m fragmentos recuperados. As seqüências de informação originais F_i, são encontradas multiplicando S'_j com a inversa de G':

$$F_i = S'_j \times G'^{-1},$$

Depois de ver as estratégias de replicação, pode se dizer que a replicação de arquivos inteiros provê maior disponibilidade que replicação de arquivos por bloco. Isto porque só precisamos de um arquivo disponível de todos os replicados para ter o arquivo completo e no caso de replicação por blocos de dados precisamos de um conjunto dos blocos replicados, mas a probabilidade de ter todos disponíveis no momento reduze sua disponibilidade.

Para arquivos maiores, uma análise quantitativa entre a codificação de rasura (*erasure codes*) e as técnicas de replicação [WK02] mostra que a codificação de rasura provê maior garantia de disponibilidade comparada com a replicação de arquivos inteiros. Isto devido ao fato que não é preciso recuperar o total de fragmentos e sim um subconjunto conjunto suficiente de fragmentos para reconstruir o arquivo.

3.5.1 Grade de Dados

Grades de dados [VBR06] são adotadas por muitas comunidades científicas para o compartilhamento, acesso, transporte, processo e administração de um conjunto de dados distribuídos. Combina computação de alto desempenho e técnicas de administração de armazenamento de dados.

Uma grade de dados provê os serviços que ajudam na descoberta, transferência e manipulação de um conjunto de dados armazenados em um conjunto de repositórios, além de criar e administrar as réplicas desse conjunto de dados. Também provê uma plataforma através da qual os usuários pedem recursos computacionais e de armazenamento para a execução de suas aplicações sobre dados remotos.

Uma grade de dados deve prover as funcionalidades básicas de alto desempenho relacionado com o mecanismo de transferência de dados e um mecanismo escalável de descoberta e gerenciamento de réplicas [VBR06]. Uma grade de dados pode manter o compartilhamento de dados entre os diferentes domínios como diversas organizações, universidades ou institutos de pesquisa.

Os recursos em uma grade são heterogêneos em termos de ambientes de operação, capacidade e disponibilidade e estão sobre controle de seus próprios domínios locais administrativos. Os grades de dados são concebidos para o compartilhamento de recursos, autenticação e autorização de entidade, e gerenciamento de recursos além de escalonamento eficiente e efetivo para o uso de recursos

disponíveis.

Dentro das características principais das grades de dados podem-se citar:

- Um massivo conjunto de dados, aplicações intensivas de dados são caracterizados pela presença de uma grande quantidade de dados em termos de Petabytes ou superiores. Gerenciamento dentro das grades de dados inclui a criação de réplicas através de estratégias de administração de recursos de armazenamento.
- Compartilhamento de coleções de dados, os recursos de compartilhamento dentro de grade de dados incluem entre outros o compartilhamento distribuído de coleções de dados.
- Espaço de nomes unificado, os dados na grade de dados compartilham os mesmos espaços lógicos de nomes no qual todos os elementos tem um único nome de arquivo lógico. Este nome lógico depois é mapeado para um ou mais arquivos de nomes físicos sobre vários recursos na grade.
- Restrições de acesso, usuários podem desejar assegurar a confidencialidade dos seus dados ou restringir sua distribuição para colaboradores próximos.

A replicação de dados garante a escalabilidade da colaboração dos membros do sistema, disponibilidade de acesso a dados preservando a largura da banda. Esta replicação está limitada pelo tamanho disponível de armazenamento presente dentro da grade de dados assim como pela largura de banda entre os locais de armazenamento. Um gerenciador de réplicas garante acesso a dados requisitados enquanto gerencia o armazenamento subjacente.

3.6 Resumo

Neste capítulo foram apresentados sistemas de armazenamento distribuído de dados que utilizam redundância de dados para incrementar a disponibilidade.

Past utiliza a técnica de replicação para garantir a durabilidade dos dados e incrementar sua disponibilidade. A replicação tem o efeito de balancear a carga das requisições pelo arquivo e reduzir as latências de acesso devido à forma como Pastry faz o roteamento das mensagens. CFS lida com as falhas dos nós replicando os blocos de dados sobre os nós que sucedem o nó que é responsável. O objetivo de replicar os blocos é permitir a escolha do nó que permita descarga mais rápida na hora de transmitir o bloco, além de incrementar a disponibilidade ante a presença de falhas. Já no caso do OceanStore o uso da codificação de rasura incrementa a disponibilidade dos fragmentos codificados. A tabela 3.1 mostra o resumo das características dos sistemas de armazenamento apresentados.

	PAST	CFS	OceanStore
Ambiente de execução	P2P	P2P	P2P
Infra estrutura P2P	Pastry	Chord	não estruturada
Organização de nós	P2P descentralizado	P2P descentralizado	P2P descentralizado
Recursos	Máquinas com baixa conectividade	Máquinas com baixa conectividade	Alta conectividade entre servidores
Tipo de dados	Imutáveis	Imutáveis	Mutáveis
Tipo de armazenamento	Permanente	Temporário	Permanente
Dados armazenados	Arquivo completo	Bloco de dados	Fragmentos codificados
Mecanismo de redundância	Replicação	Replicação	Codificação de rasura (Read-Solomon codes and Tornado codes)

Tabela 3.1: Resumo comparativo dos sistemas de armazenamento apresentados

Apresentamos também as estratégias de replicação que podem ser usadas em cada um desses sistemas, assim como o armazenamento de dados em grades computacionais. Finalmente, apresentamos o Neutralizer, detector de falhas auto-configurável que procura estabelecer um equilíbrio na quantidade de réplicas que devem ser mantidas quando ocorre a saída de nós do sistema.

Diferentemente dos sistemas acima mencionados, também apresentamos os conceitos de grades de dados, que permitem armazenamento distribuído de dados, porém precisam de sistemas gerenciadores de réplicas para manter a disponibilidade dos dados por ele gerenciados.

No capítulo seguinte veremos especificamente o armazenamento distribuído de dados no middleware OppStore, que provê o armazenamento distribuído de dados em grades oportunistas, parte do objetivo de nosso trabalho.

Capítulo 4

Armazenamento distribuído de dados no OppStore

Grades oportunistas como o InteGrade [GKG+] utilizam tempos ociosos de computadores em laboratórios, universidades para a execução de aplicações, mas não só o poder de processamento destas grades oportunistas pode ser aproveitado.

Para aproveitar os recursos de armazenamento em grades oportunistas, utilizamos o OppStore [dC08], um middleware que permite utilizar o espaço não utilizado de computadores pertencentes a grades oportunistas, tais como o InteGrade, para armazenar dados de usuários e aplicações da grade.

Para prover operações de armazenamento, os arquivos devem ser codificados em fragmentos redundantes e distribuídos entre as máquinas da grade para seu armazenamento. Contudo, as máquinas da grade podem falhar e os fragmentos nelas armazenados ficam perdidos. Atualmente OppStore não implementa um tratamento para a perda de fragmentos, essa falta compromete a disponibilidade dos arquivos chegando a provocar a perda dos mesmos.

Nosso interesse ao estudar o OppStore é contribuir com seu funcionamento, ao implementar um mecanismo que permita o tratamento dessa perda de fragmentos e assim, contribuir com a disponibilidade dos arquivos armazenados ante a saída imprevista de nós do sistema causadas por falhas nas máquinas da grade.

4.1 Oppstore

OppStore [dC08] é um middleware para armazenamento distribuído de dados que utiliza as máquinas não dedicadas e disponibilizadas de grades oportunistas para armazenar os dados dos

usuários e aplicações da grade.

As máquinas são organizadas como uma federação de aglomerados (máquinas fisicamente próximas), que correspondem com a organização de diversos sistemas de grades oportunistas. Estes aglomerados estão conectados em uma rede peer-to-peer estruturada usando Pastry [RD01] como substrato para o roteamento das mensagens para armazenamento e requisição de dados.

OppStore faz uso dos recursos de armazenamento das máquinas quando elas estão ociosas, mas também os recursos de espaço para armazenamento podem ser disponibilizadas pelos usuários. Dado que muitas das máquinas que pertencem aos aglomerados da grade podem ficar indisponíveis, os dados armazenados nessas máquinas ficarão indisponíveis permanentemente ou até que esses nós voltem ao sistema. Também podemos ter a situação em que os donos das máquinas solicitem o retorno do espaço de armazenamento cedido, em ambas situações é importante manter uma quantidade adequada de fragmentos armazenados que permitam manter a disponibilidade dos arquivos.

Para permitir as funcionalidades de armazenamento e recuperação de arquivos, cada aglomerado no OppStore deve disponibilizar uma máquina que instancia o módulo responsável pelo gerenciamento das máquinas desse aglomerado denominado gerenciador de repositório de dados (CDRM). As demais máquinas instanciam o módulo chamado repositório autônomo de dados (ADR) e funcionam como repositórios de dados ao ser usados para o armazenamento dos fragmentos. A figura 4.1 mostra a arquitetura do OppStore. Nessa figura, podemos observar como os CDRMs se comunicam formando uma rede peer-to-peer para realizar a troca de mensagens durante as operações armazenamento e recuperação de dados.

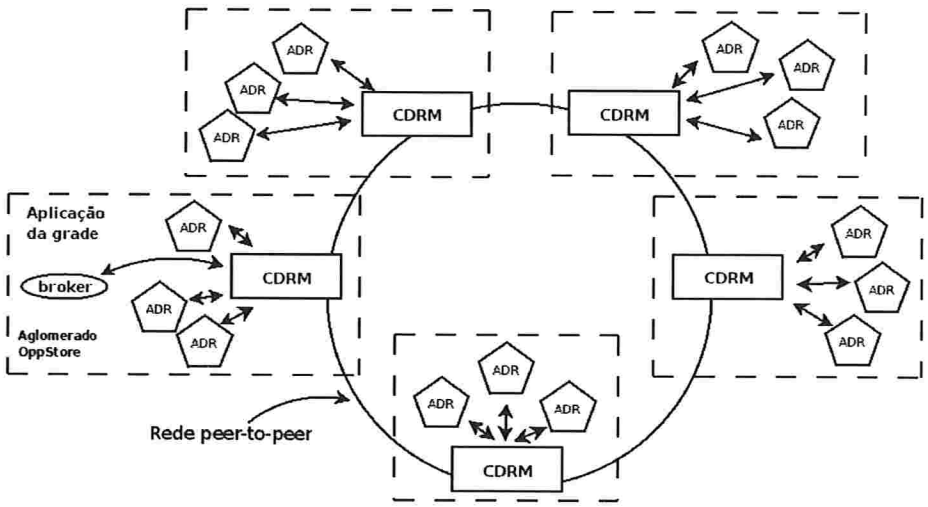

Figura 4.1: Arquitetura do OppStore [dC08]

Os ADRs funcionam como repositórios para o armazenamento dos fragmentos, eles recebem re-

quisições para armazenamento e recuperação de fragmentos em uma máquina na qual foi instanciada.

Para o armazenamento de dados, OppStore codifica os arquivos em fragmentos redundantes fazendo uso da codificação de rasura (*erasure code*), para isso utiliza uma versão otimizada do algoritmo de dispersão de informação (IDA) [Rab89b]. Após a codificação, distribui os fragmentos entre os diferentes aglomerados da grade e assim, finalizar com seu armazenamento.

Quando é feita uma requisição para a recuperação de um arquivo, é necessário recuperar apenas um número suficiente do total dos fragmentos armazenados para reconstruir o arquivo original. Durante a recuperação dos fragmentos, são escolhidos os aglomerados mais próximos para assim melhorar o desempenho do sistema.

4.2 Componentes Principais do OppStore

OppStore está composto por três componentes principais: O repositório autônomo de dados (ADR), o gerenciador de repositórios de dados do aglomerado (CDRM), e o intermediador de acesso (*access broker*).

4.2.1 Repositório Autônomo de dados (ADR)

Os repositórios autônomos de dados (ADRs) são responsáveis pelo armazenamento dos fragmentos resultantes da codificação dos arquivos após usar a codificação de rasura.

Quando os ADRs ingressam ao aglomerado, eles se registram com o CDRM responsável pelo aglomerado ao qual estão ingressando. Para isso, enviam para o CDRM responsável do aglomerado informações, como o endereço de rede e a capacidade de espaço disponível. O endereço de rede do ADR é fornecido pelo CDRM ao *access broker* para que estes possam estabelecer comunicação direta com o ADR. Essa comunicação é realizada quando é realizada a transferência de fragmentos, feita direitamente entre ADRs e o *access broker* nas operações de armazenamento e recuperação de arquivos.

O CDRM no inicio atribui ao ADR o valor médio da disponibilidade dos demais ADRs do aglomerado. Após coletar informações sobre a disponibilidade média da máquina por um tempo determinado, o valor de disponibilidade do ADR que acaba de ingressar é atualizada pelo CDRM.

4.2.2 Gerenciador de repositórios de dados do aglomerado(CDRM)

Os CDRMs são os responsáveis pelo gerenciamento dos ADRs presentes em seu aglomerado. Cada CDRM mantém informações sobre todos os ADRs que formam parte de seus aglomerados.

Estas informações incluem a capacidade, endereço de rede, estado, espaço livre em disco e a lista de fragmentos armazenados.

Quando o CDRM recebe uma requisição para o armazenamento de um fragmento de um arquivo, ele escolhe um ADR para armazenar o arquivo e devolve o endereço de rede. As restrições são que o ADR tenha espaço suficiente para armazenar o fragmento e que esteja disponível no momento.

CDRMs também são responsáveis pelo armazenamento dos FFI. Os FFI (*File Fragment Index*), são estruturas que contêm as informações relacionadas com a localização dos fragmentos e são criadas após terminado o armazenamento dos fragmentos e são encaminhados até o CDRM responsável pelo identificador do FFI para seu posterior armazenamento.

O FFI contém informações como: (1) O identificador de cada fragmento, (2) o endereço do ADR que armazena cada fragmento e (3) uma cadeia de caracteres que identifica o arquivo. O identificador do fragmento é usado durante o processo de reconstrução do arquivo para verificar a integridade do conteúdo do fragmento.

Para prover tolerância a falhas, as informações contidas no CDRM, são replicadas em outros k CDRMs vizinhos, onde k é um parâmetro configurável. Quando um CDRM falha, ele é reiniciado em outro nó do aglomerado com as informações replicadas nos CDRMs vizinhos.

4.2.3 Intermediador de acesso (Access Broker)

É uma biblioteca que permite que aplicações e usuários da grade acessem os serviços de armazenamento de dados do OppStore. Provê uma API em C contendo funções para realizar o armazenamento e recuperação de dados.

Devido à aplicação do IDA que permite a codificação e decodificação incremental dos fragmentos, pode-se transferir os fragmentos paralelos com sua codificação. No caso de recuperação de fragmentos, o arquivo pode ser reconstruído incrementalmente, à medida que os fragmentos são recuperados. A recuperação dos fragmentos de dados pode ser feita também de forma paralela usando os fragmentos mais próximos ao CDRM que recebeu a requisição para recuperar o arquivo inteiro.

4.3 Armazenamento e Recuperação de Dados

O OppStore permite que as aplicações clientes escolham entre duas formas de armazenamento de arquivos, (1) O armazenamento perene, e (2) O armazenamento efêmero.

4.3.1 Armazenamento Perene

O armazenamento perene é usado para armazenar dados por um período longo de tempo. Neste tipo de armazenamento, os arquivos são codificados em fragmentos redundantes e distribuídos entre os diferentes aglomerados da grade para seu armazenamento.

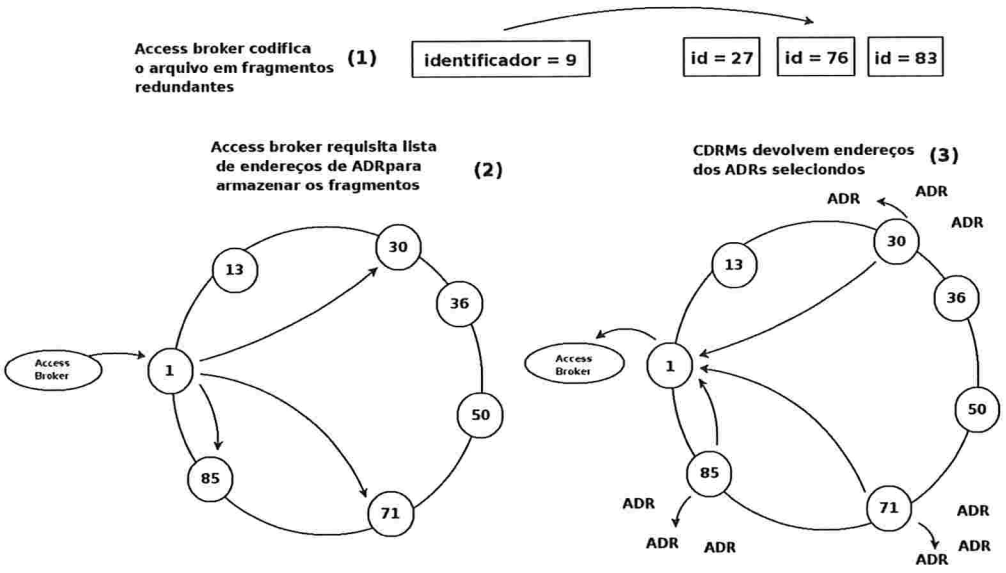

Figura 4.2: Armazenamento de dados no OppStore [dC08]

O processo de armazenamento perene de arquivos é como segue:

1. O *access broker* codifica o arquivo que deve ser armazenado em m fragmentos redundantes usando algoritmo de dispersão de informação (IDA) [Rab89b] e atribui um identificador para cada fragmento gerado. (figura 4.2)

2. O *access broker* envia a lista de fragmentos codificados ao CDRM do seu aglomerado. Este CDRM realiza o roteamento de uma mensagem para cada identificador na rede peer-to-peer até encontrar o CDRM responsável pelo aglomerado que será responsável por armazenar cada fragmento. (figura 4.2)

3. Cada CDRM responsável por um fragmento, seleciona um ADR do seu aglomerado e devolve o endereço de rede do ADR selecionado ao CDRM original. O CDRM original repassa esta lista de endereços de ADRs para o *access broker*. (figura 4.3)

4. O *access broker* transfere o conteúdo dos fragmentos direitamente para os ADRs. (figura 4.3)

5. Após o término da transferência, o *access broker* notifica ao CDRM do seu aglomerado, então o CDRM cria uma estrutura chamada FFI(**file fragment index**) que contém a localização, assim como o identificador de cada fragmento. O CDRM do aglomerado realiza o roteamento

de uma mensagem contendo o FFI que deverá ser armazenado remotamente no CDRM responsável pelo identificador do FFI. (figura 4.3)

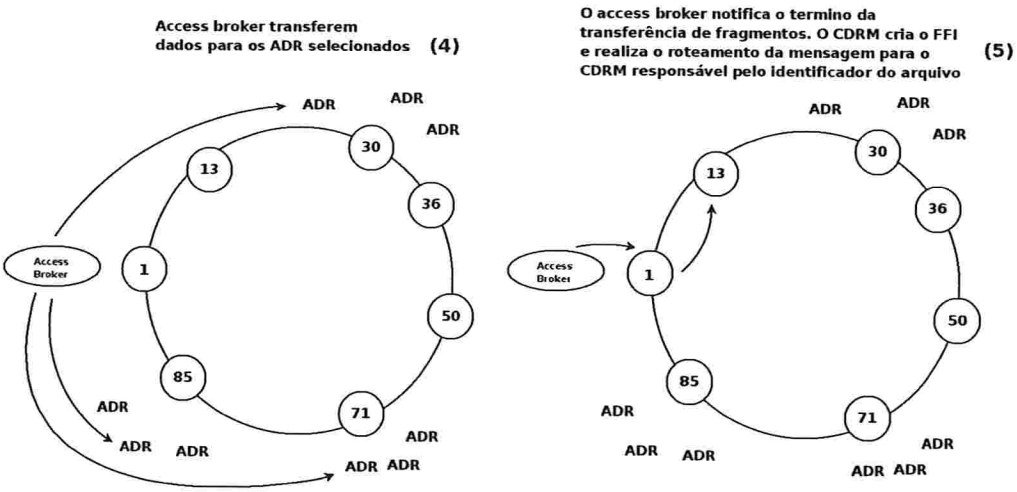

Figura 4.3: Armazenamento de dados no OppStore [dC08]

4.3.2 Armazenamento Efêmero

No caso do armazenamento efêmero, duas ou mais réplicas do arquivo são armazenadas nas máquinas do aglomerado. Este modo de armazenamento é definido para dados que serão usados em período curtos de tempo, como poucas horas.

Para realizar o armazenamento efêmero, o *access broker* solicita ao CDRM de seu aglomerado uma lista de ADRs daquele aglomerado, onde as réplicas do arquivo serão armazenadas. O *access broker* transfere as réplicas para estes ADRs e notifica ao CDRM de seu aglomerado a finalização do armazenamento do arquivo. Finalizado o armazenamento do arquivo, o CDRM cria o FFI contendo os locais onde cada réplica foi armazenada, assim como o valor SHA-1 das réplicas. O CDRM atribui como identificador do FFI o valor do SHA-1 das réplicas e realiza o armazenamento remoto do FFI no CDRM responsável pelo identificador do FFI.

4.3.3 Processo de Recuperação de dados

Toda vez que é solicitado a recuperação de um arquivo armazenado pelo OppStore é iniciado o processo de recuperação do arquivo. Para isso deve-se criar e rotear mensagens pedindo os fragmentos para assim iniciar o processo de reconstrução do arquivo original. O processo de recuperação de arquivos é o seguinte:

1. O *access broker* requisita ao CDRM do seu aglomerado o FFI associado ao arquivo desejado. Este FFI contém as informações associadas a cada arquivo armazenado no OppStore, nele estão registradas informações sobre a ubicação e o nome do arquivo.

2. Após receber o FFI, o *access broker* verifica sua integridade aplicando uma função SHA-1 sobre seu conteúdo. Após comprovada a integridade do FFI, o *access broker* realiza a recuperação dos fragmentos e escolhe para isso os repositórios mais próximos e com maior velocidade de conexão.

3. Caso consiga recuperar um número suficiente de fragmentos o arquivo original é recuperado. Caso contrario, o arquivo fica indisponível.

Como foi mencionado, OppStore disponibiliza operações para o armazenamento e a recuperação de arquivos, embora para permitir essas operações ele deve ser implantado sobre grades computacionais oportunistas como o InteGrade [GKG+], OurGrid [ACGC05], Condor [TTL03], definindo uma interface para que assim, usuários e aplicações da grade possam interagir com os dados armazenados.

Ampliamos a descrição do InteGrade, dado que é possível implantar OppStore nesta grade oportunista e assim testar o funcionamento do mecanismo proposto para o tratamento de falhas.

4.4 InteGrade

InteGrade [GKG+] é um middleware para implantação de grades computacionais sobre recursos computacionais não dedicados. Foi concebido com o objetivo de destinar o uso dos recursos computacionais (ciclos de CPU) ociosos disponibilizados pelos usuários para execução aplicações paralelas ou aplicações sequênciais.

InteGrade foi desenvolvido baseado numa arquitetura de objetos distribuídos usando CORBA [VI97] que permite o desenvolvimento de aplicações para ambientes heterogêneos e facilita a integração de módulos escritos em diversas linguagens e são executadas sobre diferentes plataformas. CORBA que fornece os serviços de transações, persistência, nomes e *trading* aproveitados no InteGrade.

Uma característica importante com a qual foi concebido o InteGrade é que os usuários que compartilham os recursos das suas máquinas não deveriam perceber perda de desempenho das aplicações que eles executam.

InteGrade inclui a possibilidade de coletar e analisar informações sobre padrão de disponibilidade das máquinas da grade e assim pode determinar a probabilidade da disponibilidade dos recursos de

um nó para executar alguma tarefa. Esta característica é muito útil para melhorar as decisões de escalonamento de tarefas, assim, evitar relançar as tarefas em outras máquinas se a máquina que estiver realizando alguma tarefa ficasse indisponível antes de terminar a execução da sua tarefa.

4.4.1 Arquitetura

InteGrade é formado por um conjunto de aglomerados, onde cada aglomerado consiste de um conjunto de máquinas agrupadas por algum critério, por exemplo, máquinas que pertençam a um domínio administrativo. Tipicamente um aglomerado está conformado por máquinas fisicamente próximas em termos de conectividade.

A figura 4.4 apresenta os componentes típicos de um aglomerado do Integrade. As máquinas que compõem o aglomerado, chamadas também de nós, desempenham um papel específico na grade. De acordo com isso, encontramos os seguintes tipos de nós:

- O nó dedicado é uma máquina dedicada para a computação em grade, como em um aglomerado dedicado tradicional.
- O nó compartilhado é aquele pertencente a um usuário que disponibiliza seus recursos ociosos à grade.
- O nó de Usuário é aquele por meio do qual podemos submeter aplicações para ser executadas na grade.
- Finalmente, temos um nó gerenciador do aglomerado que é o nó onde são executados os módulos responsáveis pela coleta de informações e escalonamento entre outros.

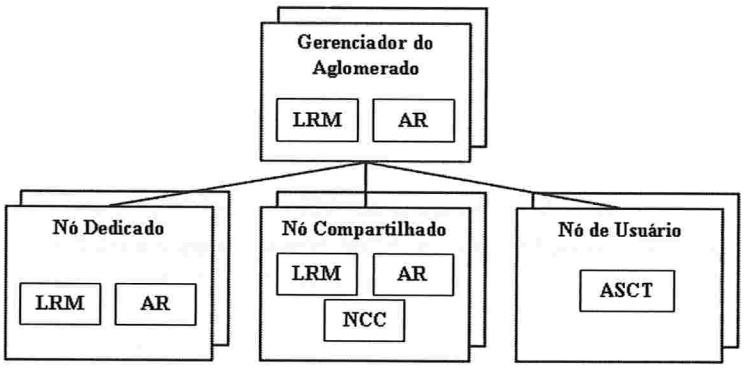

Figura 4.4: Arquitetura intra-aglomerado do InteGrade [GKG+]

Cabe ressaltar que um nó no InteGrade pode ser considerado como pertencente a duas categorias simultaneamente, assim por exemplo, um nó que compartilha seus recursos ociosos é também capaz

de submeter aplicações para serem executadas na grade.
Os módulos que são executados nas máquinas dos aglomerados são os seguintes:

- O **LRM** *(Local Resourse Manager)* é executado em todas as máquinas que compartilham seus recursos com a grade. É responsável pela coleta de informações referentes à disponibilidade dos recursos de cada nó, além de exportar os recursos desse nó à grade.
- O **GRM** *(Global Resourse Manager)* é executado no nó Gerenciador de Aglomerado, utiliza informações que dispõe para escalonar tarefas aos nós mais apropriados, estas informações provêem das informações coletadas pelos **LRMs**, e são atualizadas periodicamente.
- O **NCC** *(Node Control Center)* é executado em máquinas que compartilham seus recursos com a grade, permitindo ao proprietário da máquina definir os recursos para compartilhamento. Atuando em forma conjunta com o LRM, permite que um usuário defina os períodos em que os recursos podem ser ou não podem ser utilizados. O NCC ainda não foi implementado.
- O **ASCT** *(Aplication Submission and Control Tool)* permite que um usuário submeta uma aplicação para ser executada na grade, além disso permite ao usuário configurar requisitos como plataformas de hardware ou software, e preferências como quantidade de memória para executar o melhor da aplicação. Também permite o monitorar o andamento da execução da aplicação.
- O **LUPA** *(Local Usage Pattern Analyzer)* é responsável do análise e monitoramento dos padrões de uso dos recursos das máquinas da grade pelos seus usuários, o resultado da análise das informações coletadas pelos LRM são usadas para a toma de decisões sobre o escalonamento, fornecendo informações sobre probabilidades de maior disponibilidades de recursos em cada nó.
- O **AR** *(Application Repository)* armazena as aplicações que serão executadas na grade. As aplicações são registradas através do ASCT. Quando o LRM recebe um pedido de execução de uma aplicação, o LRM requisita essa aplicação ao repositório de aplicações.

4.4.2 Principais Protocolos

Os módulos dos aglomerados do InteGrade colaboram de maneira a desempenhar funções importantes no sistema, para isso utilizam os protocolos de disseminação de informações e o protocolo de execução de aplicações.

Protocolo de disseminação de informações

Este protocolo no InteGrade permite que o GRM mantenha informações relativas à disponibilidade dos recursos nas diversas máquinas do aglomerado. Informações importantes para o escalonamento de aplicações como: arquitetura da máquina, sistema operacional, memória, assim como

informações de porcentagem de CPU ocioso, quantidades disponíveis de memória e disco, entre outros.

A cada intervalo de tempo t_1 o LRM verifica a disponibilidade dos recursos do nó. Em caso da presença de mudanças significativas, o LRM envia a variação de essas informações ao GRM. Caso não existam mudanças significativas em um intervalo t_2 ($t_2 \geq t_1$) do mesmo jeito envia uma atualização das informações. Tal atualização serve para detectar quedas nos LRMs.

Protocolo de Execução de Aplicações

Permite que um usuário da grade submeta aplicações para sua execução sobre recursos compartilhados. Derivado do protocolo de execução utilizado no sistema 2K [KCD+00]. O funcionamento do protocolo é explicado a seguir:

Um usuário solicita a execução de uma aplicação contida no Repositório de Aplicações e submetida através do ASCT (1). Podem-se especificar requisitos para a execução da aplicação, por exemplo, a arquitetura sobre a qual aplicação foi compilada. Assim que a requisição de execução é enviada ao GRM este procura um nó para execução (2).

Caso consiga uma máquina disponível para executar a aplicação, o GRM envia a solicitação ao LRM da máquina candidata a executar a aplicação (3).

O LRM verifica se possui recursos disponíveis para executar a aplicação, caso não possua os recursos necessários, o LRM informa ao GRM e este retorna ao passo (2).

Caso o nó possua os recursos o LRM pede a aplicação ao Repositório de Aplicações (4), solicita os arquivos de entrada da aplicação ao ASCT solicitante (5) e lança a aplicação (6) notifica ao ASCT que sua requisição foi atendida (7).

No caso que a execução da aplicação precisar de vários nós no passo (2) o GRM procura até n máquinas para executar a aplicação, depois vai para o passo (3) onde o GRM realiza n chamadas para executar a aplicação.

4.4.3 Análise dos padrões de uso das máquinas da grade

O objetivo da análise dos padrões de uso é detectar quais máquinas dos aglomerados podem receber determinadas tarefas no momento certo para reduzir os tempos de execução.

A análise de padrões de uso se realiza no nível local através do modulo LUPA instalado em todas as máquinas dos aglomerados que formam parte da grade. O LUPA coleta informações referentes sobre como os usuários usam seus recursos computacionais como CPU, memória, espaço em disco, entre outras, através do tempo. Estas informações são coletadas como séries temporais e são usadas para predizer a probabilidade do grau de uso ou disponibilidade de alguma máquina em certo dia para fazer possível a alocação de tarefas.

A análise de séries de tempo se realiza em um ambiente de aprendizado de máquina, e o processo é desenvolvido em duas fases. A fase de aprendizado, durante a qual as séries de tempo dos recursos usados são coletadas por um período de tempo. Os dados são coletados 24 horas durante várias semanas em intervalos de 5 minutos . As séries de tempo, então são processadas para determinar os padrões de uso de recursos. Durante a fase operacional, uma decisão sobre a disponibilidade de uma máquina da grade deve ser realizada. As séries de tempo continuam sendo coletadas durante a fase operacional.

4.5 Resumo

Neste capítulo foi apresentada a estrutura e os componentes principais do OppStore, assim como o funcionamento do mesmo na execução de operações para armazenamento e recuperação dos arquivos nele armazenados. OppStore permite armazenamento perene para períodos longos de tempo, assim como o armazenamento efêmero para períodos curtos.

Apresentamos o InteGrade, grade oportunista onde o OppStore pode ser implantado para assim avaliar o desempenho das propostas do mecanismo para o tratamento das falhas apresentadas, que é parte de nosso trabalho.

No capítulo seguinte são apresentadas as abordagens adotadas para garantir a disponibilidade dos dados ante a presença de falhas nos nós do sistema.

Capítulo 5

Tolerância a falhas no OppStore

Um sistema de armazenamento de dados precisa manter a disponibilidade dos dados por ele gerenciados durante a presença de falhas no sistema. No caso do funcionamento do OppStore que opera sobre uma grade oportunista, o gerenciador de dados é o responsável por iniciar os protocolos de entrada e saída dos nós que formam parte do sistema, além de prover um nível de disponibilidade aceitável dos dados por ele gerenciados, mesmo durante a presença de falhas no sistema. Este nível aceitável implica manter a um número disponível de fragmentos tentando minimizar os custos no desempenho do sistema.

5.1 Tolerância a Falhas

O conceito de tolerância a falhas foi apresentado por Avizienis [Avi67] em 1967. Um sistema de armazenamento de dados exige um nível aceitável de confiabilidade e disponibilidade dos dados por ele gerenciado. Para prover essas exigências, o sistema deve ser construído considerando técnicas de tolerância a falhas. Essas técnicas vão garantir o correto funcionamento do sistema mesmo na ocorrência de falhas e são baseadas em níveis de redundância, exigindo para esse propósito a utilização de técnicas e algoritmos especiais.

Num sistema de armazenamento distribuído de dados que funciona sobre uma grade oportunista, onde a presença das máquinas é variável em termos de disponibilidade, a presença de falhas nos membros é esperada. A execução de um mecanismo de tolerância a falhas é fundamental, dado que falhas apresentadas em nós do sistema comprometem a disponibilidade dos dados armazenados.

No caso do OppStore, as falhas em nós do sistema que podem ser causadas pela saída de máquinas por períodos de tempo variável e indeterminado ou retiradas pelos usuários, implementamos estratégias que permitem manter a disponibilidade de uma quantidade de fragmentos de dados suficientes para a recuperação do arquivo original quando as saídas dos nós acontecem.

5.2 Mecanismo de Tolerância a Falhas no OppStore

OppStore codifica os arquivos em fragmentos redundantes e os distribui entre os diferentes aglomerados para seu armazenamento. Para realizar a codificação dos arquivos, utiliza o algoritmo de dispersão de informação (IDA) [Rab89a]. A distribuição dos dados incrementa a disponibilidade dos arquivos armazenados, dado que o arquivo original pode ser reconstruído a partir de uma quantidade suficiente de fragmentos quando as falhas acontecem no sistema.

Conforme as máquinas que compõem a grade apresentam falhas no seu funcionamento, deixam a grade por um período de tempo indeterminado ou por alguma razão perdem os fragmentos dos arquivos que armazenam, o número de fragmentos de dados armazenados correspondentes a um arquivo determinado diminui.

Se a saída de nós dos aglomerados da grade continua, poderia-se chegar à situação de que em algum momento determinado, o número de fragmentos necessários para reconstruir o arquivo original seja insuficiente, trazendo como consequência a perda do arquivo armazenado.

No nosso trabalho implementamos um mecanismo de tolerância a falhas no OppStore que está constituído por um protocolo responsável por manter um número adequado de fragmentos, toda vez que um nó deixa algum aglomerado que forma parte da grade e que a quantidade de fragmentos disponíveis para recuperar o arquivo original alcançe um limiar L estabelecido.

Este limiar L indica a quantidade de fragmentos que devem ficar disponíveis para permitir a recuperação do arquivo original antes de iniciar o mecanismo de substituição de fragmentos perdidos.

O mecanismo é iniciado quando é detectada uma falha permanente ou transiente em qualquer nó que forma parte dos aglomerados da grade. Definimos que um nó apresenta falha permanente ou apresenta falha transiente quando ele não se comunica com o CDRM do seu aglomerado por um tempo t determinado, portanto, ficará indisponível.

Dado que a execução do protocolo para a substituição dos fragmentos perdidos implica a reconstrução do arquivo original, o custo associado à quantidade de mensagens enviadas e ao tráfego de dados na rede como consequência da troca de mensagens é potencialmente alto, então introduzimos uma variação no processo de armazenamento de fragmentos que evita o inicio do protocolo de substituição de fragmentos perdidos.

5.2.1 Mecanismo de substituição de fragmentos perdidos

Quando acontece a saída de um nó de algum aglomerado devido a uma falha apresentada, os fragmentos armazenados nesse nó são considerados como perdidos, então devem-se atualizar as informações contidas nas estruturas que mantém informações sobre a ubicação dos fragmentos (FFIs).

Dado que cada CDRM é responsável por monitorar os ADRs do seu aglomerado, quando um ADR deixa o sistema por alguma falha, o CDRM verifica a lista de fragmentos que estavam armazenados no ADR perdido e os identificadores dos FFIs que contêm esses fragmentos.

As figuras 5.1 e 5.2 mostram simplificadamente como é feito o processo de substituição de fragmentos perdidos.

Processo de atualização de Id fragmentos disponíveis nos FFIs

```
se ADR deixa a grade por uma falha apresentada então
    Obter lista de identificadores de fragmentos armazenados no ADR.
    Criar uma mensagem para atualizar o FFI por cada Id de fragmento
    Rotear a mensagem até o CDRM responsável pelo FFI procurado
    se mensagem de atualização do FFI chega até CDRM reponsável então
        Atualizar o FFI, tirando os identificadores de fragmentos perdidos.
        se Nro fragmentos do FFI = LIMIAR adotado então
            Iniciar o processo de reconstrução de fragmentos perdidos.
        fim
    fim
fim
```

Figura 5.1: Processo de atualização de fragmentos disponíveis no FFI (I)

Processo de reconstrução de fragmentos perdidos

```
CDRM requisita baixar fragmentos a partir das chaves de fragmentos contidos no FFI.
se Nro fragmentos recuperados ≥ Nro fragmentos necessários então
    Reconstruir arquivo e o obter fragmentos perdidos
fim
para i = 1 até Nro fragmentos recuperados faça
    Procurar endereço válido para fragmento i
    se Todos endereços foram encontrados então
        Armazenar fragmentos
    fim
fim
se todos novos fragmentos foram armazenados então
    Criar e rotear mensagem de atualização do FFI
fim
```

Figura 5.2: Processo de reconstrução de fragmentos perdidos (II)

A seguir, apresentamos o detalhe de como é feito este processo de substituição de fragmentos perdidos.

1. O CDRM responsável pelo ADR que apresenta indisponibilidade verifica (1) quais fragmentos estavam armazenados no ADR perdido, (2) os identificadores dos FFIs que devem ser atualizados.

   Depois cria uma mensagem contendo como informações importantes (1) o identificador do FFI que deve ser atualizado, e (2) o identificador do fragmento que deve ser removido da lista de chaves armazenadas por cada fragmento no FFI responsável. Logo após criar a mensagem o CDRM inicia o roteamento desta mensagem até o CDRM responsável por armazenar o FFI que deve ser atualizado. Cada CDRM que recebe a mensagem atualiza as informações contidas nos FFIs respectivos (figura 5.3).

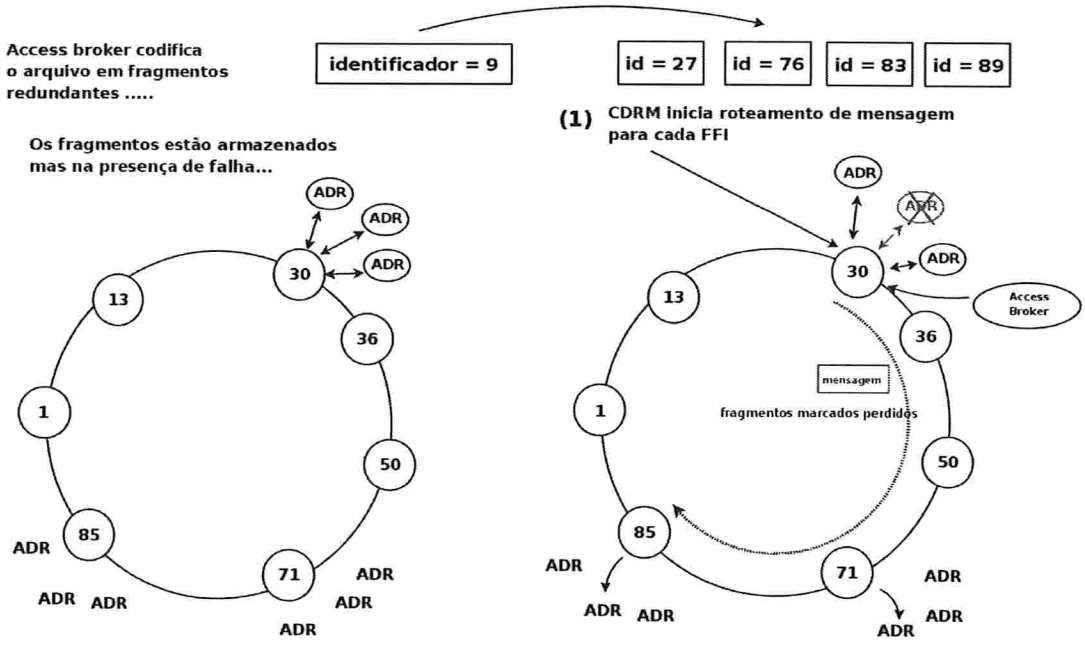

Figura 5.3: Mecanismo de substituição de fragmentos perdidos (1)

2. Quando a mensagem chega até o CDRM responsável por algum FFI que mantém armazenado, este CDRM realiza as seguintes operações:

   - Atualiza a lista de identificadores de fragmentos que estão armazenadas no FFI.
   - Logo após a atualização das informações verifica se o LIMIAR para esse arquivo foi alcançado.

3. Se o LIMIAR foi alcançado, o CDRM inicia o processo de substituição dos fragmentos perdidos (figura 5.4). Para substituir fragmentos perdidos é necessário:

- Reconstruir o arquivo original a partir dos fragmentos disponíveis.
- Obter os novos fragmentos codificando o arquivo original em fragmentos redundantes.

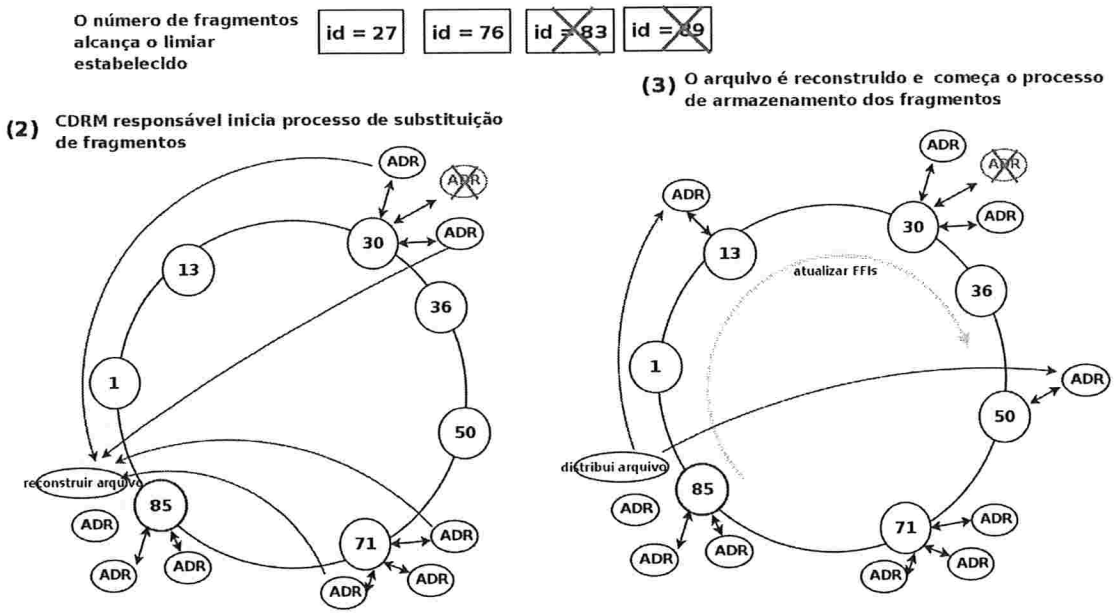

Figura 5.4: Mecanismo de substituição de fragmentos perdidos (2)

4. Após obter os novos fragmentos se realiza o armazenamento dos mesmos.

   Para cada fragmento recuperado é designado um identificador de fragmento que o CDRM responsável utiliza para criar uma mensagem que é roteada até o CDRM responsável pelos ADRs que armazenaram os novos fragmentos.

5. O CDRM responsável pelo identificador de cada fragmento recebe a mensagem de pedido de um endereço válido e este CDRM seleciona um ADR do seu aglomerado e atualiza a mensagem com o novo endereço, logo a mensagem é retornada ao CDRM que iniciou o roteamento da mensagem.

   Após obter os endereços começa a transferência dos fragmentos desde a fonte até o ADR responsável por armazenar dito fragmento. Finalmente, os FFI devem ser atualizados com a inclusão dos identificadores dos novos fragmentos e as novas localizações dos mesmos.

5.3 Evitando a execução do protocolo

Como o processo de substituição de fragmentos perdidos implica reconstruir o arquivo original, implementamos uma variação no processo de armazenamento dos fragmentos introduzindo redundância no nível de fragmentos. Assim, a substituição dos fragmentos é tratado localmente, onde o CDRM simplesmente utiliza a cópia do fragmento armazenado para substituir o fragmento perdido. Com esta abordagem evitamos a reconstrução do arquivo inteiro.

5.3.1 Cópia local de fragmento

O OppStore codifica o arquivo em fragmentos redundantes utilizando codificação de rasura, porém devido à ausência de replicação destes fragmentos, precisa-se reconstruir o arquivo por inteiro para substituir os fragmento perdidos quando as falhas apresentadas comprometem a disponibilidade dos arquivos armazenados.

A quantidade de mensagens trocadas entre os CDRMs para realizar o processo de substituição de fragmentos perdidos pode incrementar o tráfego de dados sobrecarregando a rede. Como alternativa para manter a disponibilidade dos fragmentos durante a presença de falhas em nós da grade, introduzimos uma estratégia de replicação no nível de fragmento no momento do armazenamento dos mesmos, embora sua aplicação introduza o incremento no espaço de armazenamento dos fragmentos.

Armazenar uma cópia do fragmento numa outra máquina do mesmo aglomerado permite a possibilidade de substituir algum fragmento perdido a partir da cópia do fragmento disponível. O CDRM do aglomerado é o responsável por tratar as falhas localmente quando o CDRM percebe a falha nos ADRs do seu aglomerado, então substitui uma cópia do fragmento que é perdido em outro ADR do aglomerado.

A figura 5.5 mostra como é feito o armazenamento de uma cópia de fragmento no mesmo aglomerado.

Processo de armazenamento

Codificar o arquivo em uma lista de fragmentos redundantes.
para $i = 1$ **até** *Nro fragmentos codificados* **faça**
  Procurar 2 endereços válidos para armazenar o fragmento i no mesmo aglomerado.
  se *Todos endereços foram encontrados* **então**
    Armazenar fragmentos.
    CDRM mantém uma estrutura contendo os endereços relacionados ao fragmentos
  fim
fim
se *todos novos fragmentos foram armazenados* **então**
  Criar e rotear mensagem pedindo o armazenamento do FFI criado
fim

Figura 5.5: Processo de armazenamento de fragmentos

A vantagem de fazer este procedimento é que evitamos iniciar o protocolo de substituição de fragmentos perdidos. Aplicá-lo implica utilizar o duplo de espaço de armazenamento requerido por arquivo codificado, porém evitamos a reconstrução do arquivo original.

A implementação desta abordagem como é mostrado na figura 5.6, implica armazenar uma cópia local do fragmento em outro nó do mesmo aglomerado. No momento do armazenamento de um fragmento incluímos a possibilidade de armazenar uma cópia desse fragmento em outro nó do mesmo aglomerado.

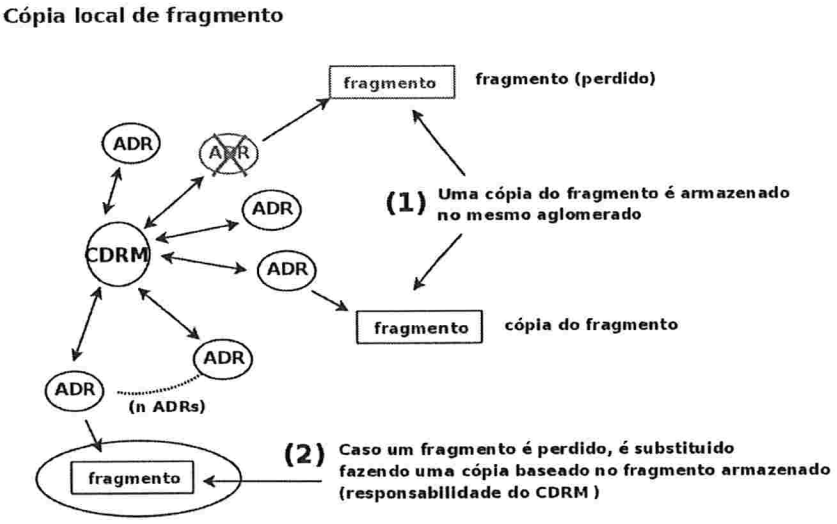

Figura 5.6: Tratamento da falha baseado na cópia local de fragmento

A figura 5.7 mostra como é realizado o processo de substituição de fragmentos perdidos baseado na cópia local de fragmento armazenada.

Processo de substituição baseado na cópia local armazenada
se *ADR deixa a grade por uma falha apresentada* **então**
  Obter lista de Id de fragmentos armazenados no ADR.
  para $i = 1$ **até** *Nro fragmentos perdidos* **faça**
    Procurar o segundo endereço onde está armazenada o cópia do fragmento i no mesmo aglomerado.
    se *segundo endereço é encontrado* **então**
      Obter cópia de fragmento.
      Encontrar um novo endereço disponível para armazenar fragmento no mesmo aglomerado.
      Armazenar fragmento.
      Atualizar a estrutura contendo os endereços relacionados ao fragmentos com o novo endereço do fragmento.
      Criar uma mensagem para atualizar o endereço do fragmento perdido i no FFI.
      Rotear a mensagem até o CDRM responsável pelo FFI procurado
      se *mensagem de atualização do FFI chega até CDRM responsável* **então**
        Atualizar o FFI, atualizando para cada Id de fragmento perdido o novo endereço disponível.
      fim
    fim
    se *segundo endereço não foi encontrado* **então**
      O CDRM contendo o FFI do arquivo deve ser notificado de que aquele fragmento foi perdido
    fim
  fim
fim

Figura 5.7: Processo de substituição de fragmentos perdidos baseado na cópia local

5.4 Resumo

Neste capítulo foram apresentados os conceitos relacionados à tolerância a falhas. Foram apresentadas as propostas dos mecanismos que foram desenvolvidas para melhorar a disponibilidade dos fragmentos na presença de falhas.

O primeiro mecanismo proposto permite substituir os fragmentos perdidos a partir da recuperação do arquivo original, precisando para este procedimento a criação de mensagens para realizar todo o processo. O segundo mecanismo, substitui os fragmentos perdidos a partir da cópia armazenada do fragmento que foi incluída no processo de inicial do armazenamento dos fragmentos.

No capítulo seguinte apresentamos a implementação dos mecanismos propostos, sendo que inicialmente apresentamos uma breve descrição da implementação dos módulos que compõem o OppStore para depois incluir as novas funcionalidades.

Capítulo 6

Implementação

Realizamos a implementação do mecanismo que será responsável por substituir os fragmentos perdidos e assim manter uma quantidade suficiente de fragmentos que garantam a futura recuperação dos arquivos armazenados. A implementação deste mecanismo implica incrementar algumas funcionalidades nos principais módulos que compõem o OppStore.

A implementação do mecanismo é feita usando java, assim como FreePastry [dpF], uma implementação de pastry de código aberto em java, que foi usada no desenvolvimento do OppStore.

6.1 Componentes do OppStore: Visão geral

Nesta seção apresentamos com detalhes os três principais módulos (CDRM, ADR, e Access Broker) que compõem o OppStore [dC08]. Estes componentes serão utilizados em nossa implementação dos mecanismos de substituição de fragmentos.

6.1.1 CDRM

O gerenciador de repositórios de dados do aglomerado **CDRM** é responsável pelo gerenciamento de ADRs dos algomerados, é responsável pelo armazenamento dos FFIs. A figura 6.1 mostra as classes que foram implementadas para cumprir estas responsabilidades. Através da classe `AdrManagerImpl` se realiza o gerenciamento dos ADRs. Para armazenamento e recuperação de dados utiliza a classe `CdrmRequestManagerImpl` que serve para chamar operações no *access broker*. O gerenciamento dos FFIs é feito usando a classe `FileFragmentIndexManager` e as mensagens são representadas como classes derivadas da classe `CdrmMessage` especialização da classe `VirtualMessage`.

Como podemos observar na figura 6.1, a classe `CdrmApp` implementa a interface `Application` definida no FreePastry. Esta implementação é necessária para o uso do protocolo Pastry no rotea-

mento de mensagens. A classe `CdrmApp` recebe e processa as mensagens recebidas pelo CDRM provindas de CDRMs de outros aglomerados através do protocolo Pastry. O CDRM utiliza dois métodos de comunicação, (1) para se comunicar como os CDRMs de outros aglomerados utiliza o protocolo de roteamento Pastry e (2) para se comunicar com ADRs e *access brokers* utiliza o CORBA. O CDRM utiliza JacORB como especificação CORBA para java, e uma versão do FreePastry estendida com identificadores virtuais.

As mensagens para gerenciamento dos dados estão representadas através da classe `CdrmMessage`, especialização da classe `VitualMessage` que foi definida para a comunicação entre CDRMs. Estas mensagens correspondem as operações de armazenamento e recuperação de arquivos no OppStore. A classe `StorefragmentMessage` representa mensagens enviadas por um CDRM para requisitar o endereço de um ADR para armazenamento de um fragmento. A classe `FileFragmentIndexMessage` representa uma solicitação para armazenamento ou recuperação de um FFI.

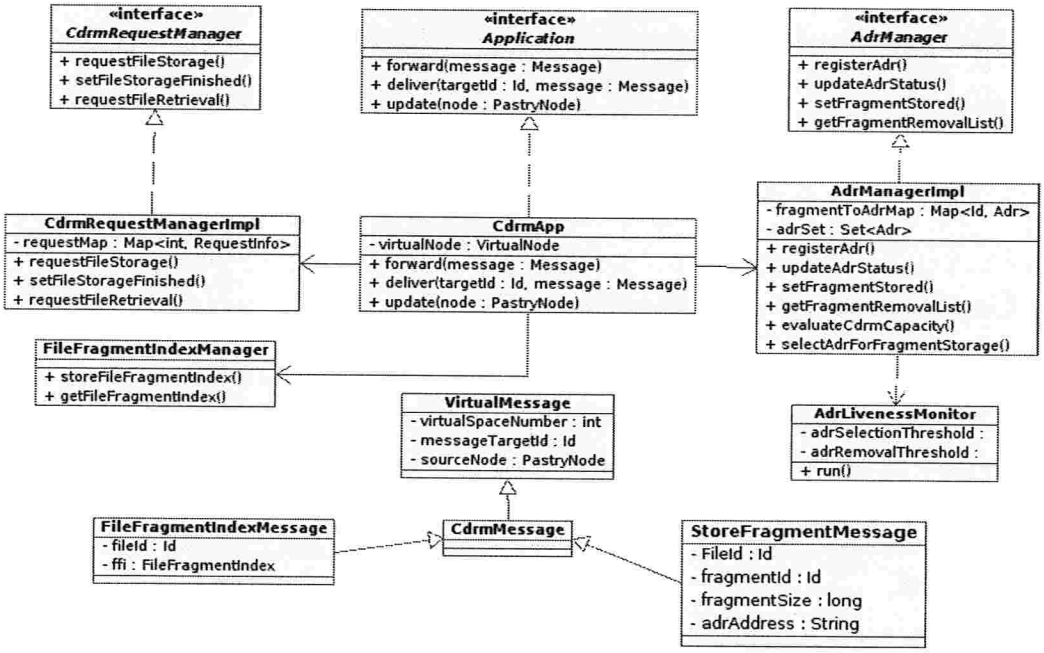

Figura 6.1: Diagrama de classes do OppStore

A classe `AdrManagerImpl` é responsável por gerenciar os ADRs do aglomerado. A classe `FileFragmentIndexManager` é a responsável pelo gerenciamento de FFIs armazenados nos CDRMs.

A interface `CdrmRequestManager` é utilizada pelo *access broker* para submeter requisições para o armazenamento e recuperação de arquivos armazenados remotamente. A interface `AdrManager`, contém operações usadas pelos ADRs para permitir seu registro no CDRM, mensagens que notifiquem o recebimento de algum fragmento de algum *access broker*.

A classe `CdrmRequestManagerImpl` implementa a interface CORBA `CdrmRequestManager` é responsável por processar as requisições de armazenamento e recuperação de arquivos. As requisições recebidas por esta classe são repassadas para a classe `VirtualNode` responsável pelo roteamento da mensagem a outros CDRMS. As respostas destas mensagens são recebidas pela classe `CdrmRequestManagerImpl` que envia os resultados obtidos ao *access broker* que realizou a requisição.

A classe `AdrManagerImpl`, implementa a interface CORBA `AdrManager` e mantém uma lista das características dos ADRs do seu aglomerado. Esta classe seleciona os ADRs que estejam disponíveis no momento e com suficiente espaço livre para armazenar o fragmento.

6.1.2 ADR

A implementação dos ADRs é feita em C++ por fazer menor uso de memória que as linguagens que usam máquinas virtuais. Quando um ADR é iniciado, ele se registra com o CDRM do seu aglomerado, enviando suas características como o estado atual e o espaço livre em disco. Uma *thread* periodicamente envia atualizações ao CDRM para informar que aquele ADR esta ativo. A comunicação com o CDRM é feita usando OiL, uma implementação de código aberto em Lua da especificação CORBA.

6.1.3 Access broker

O *access broker* está implementado em C++ como uma biblioteca compartilhada. Nesta biblioteca, estão definidas interfaces em C com funções para realizar armazenamento e recuperação de dados. A interface CORBA, `CdrmRequestManager`, definida no CDRM é utilizada pelo *access broker* para submeter requisições para o armazenamento e recuperação de dados armazenados remotamente. O CDRM processa as requisições de modo assíncrono, o *access broker* não espera pelas repostas do CDRM, as respostas são depois enviadas ao *access broker*. Para este processamento assíncrono o *access broker* instância um servidor CORBA que permite que o CDRM lhe envie respostas de requisições feitas.

Para codificação de arquivos é usada o algoritmo de dispersão de informação (IDA)[Rab89a] citado na seção 3.5, que reduz a complexidade computacional do processo de codificação e decodificação de arquivos. Para produzir os identificadores dos FFIs é utilizada uma função de espalhamento segura (SHA-1) da biblioteca openSSL sobre o conteúdo do FFI. Esta função de espalhamento é aplicada sobre o conteúdo do fragmento para assim obter o identificador do fragmento.

6.2 Mecanismo de substituição de fragmentos perdidos: Componentes

A implementação do mecanismo de substituição de fragmentos perdidos implica incrementar funcionalidades nos principais componentes do OppStore. Estas funcionalidades são implementadas como novos atributos, métodos e definição de novas classes que são usadas no mecanismo. Nesta seção apresentamos os componentes do OppStore onde foram incrementadas estas operações, descrevemos os atributos, métodos e classes que permitem o funcionamento do mecanismo.

6.2.1 CDRM

O CDRM como responsável pelo gerenciamento dos ADRs do seu aglomerado, deve obter as informações relacionadas à saída de um ADR. O CDRM deve obter os identificadores dos fragmentos que estavam armazenados no ADR perdido. Estas informações devem estar armazenadas em alguma estrutura para depois serem recuperadas. Assim, na classe `AdrManagerImpl` incrementamos como atributos (1)uma tabela `adrFragmentMap` que mantém uma lista dos identificadores de fragmentos armazenados em cada ADR, relacionando o identificador do ADR com a lista de fragmentos que estão armazenados neste ADR, e (2)uma tabela `fragmentFfiMap` que relaciona cada identificador de fragmento com um identificador de FFI associado.

Na figura 6.2, definimos as mensagens que são usadas para a atualização das estruturas que armazenam os fragmentos. Definimos as classes `UpdateFragmentListMessage` e `StoreNewFragmentMessage` derivadas da classe `CdrmMessage` para facilitar a identificação das mensagens. A classe `CdrmMessage`, definida como parte do OppStore, é uma especialização da classe `VitualMessage` que foi definida para a comunicação entre CDRMs. Estas mensagens correspondem as operações de atualização de informações no processo de substituição de fragmentos no OppStore.

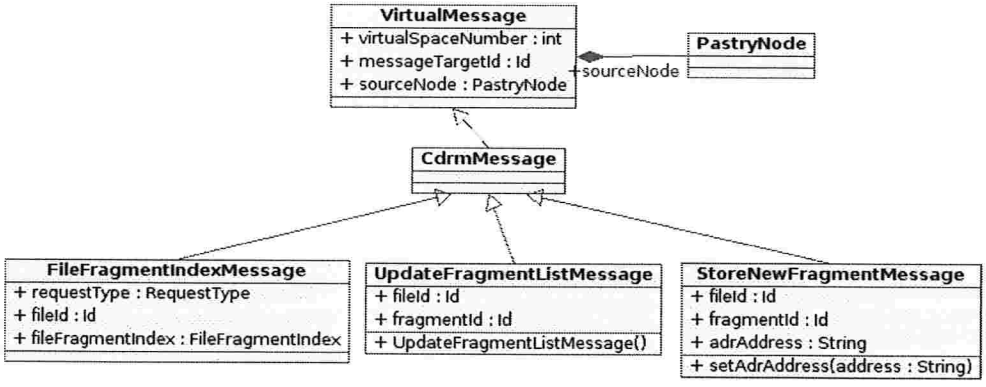

Figura 6.2: Diagrama de classes das mensagens criadas no processo de substituição

A seguir descrevemos as classes mostradas na figura 6.2, que representam as mensagens que foram acrescentadas no OppStore para o funcionamento de mecanismo de substituição de fragmentos perdidos.

- A classe `UpdateFragmentListMessage`, representa as mensagens que são criadas por cada CDRM que detecta a falha de um ADR no seu aglomerado, esta mensagem contém como informações importantes o identificador do FFI destino e o identificador do fragmento a atualizar.
- A classe `StoreNewFragmentMessage` contém o identificador do fragmento que precisa de um endereço válido para ser armazenado, também é usada para recuperar na mesma mensagem o endereço encontrado para depois ser processada pelo CDRM destino.
- A classe `FileFragmentIndexMessage` definida para solicitação para armazenamento ou recuperação de um FFI agora também será usada para atualizar os FFIs com as novas informações, tais como os novos identificadores de fragmentos, assim como os novos endereços. Incrementamos para tal fim um novo tipo de requisição *UPDATE* que é processada pelo `FileFragmentIndexManager`.

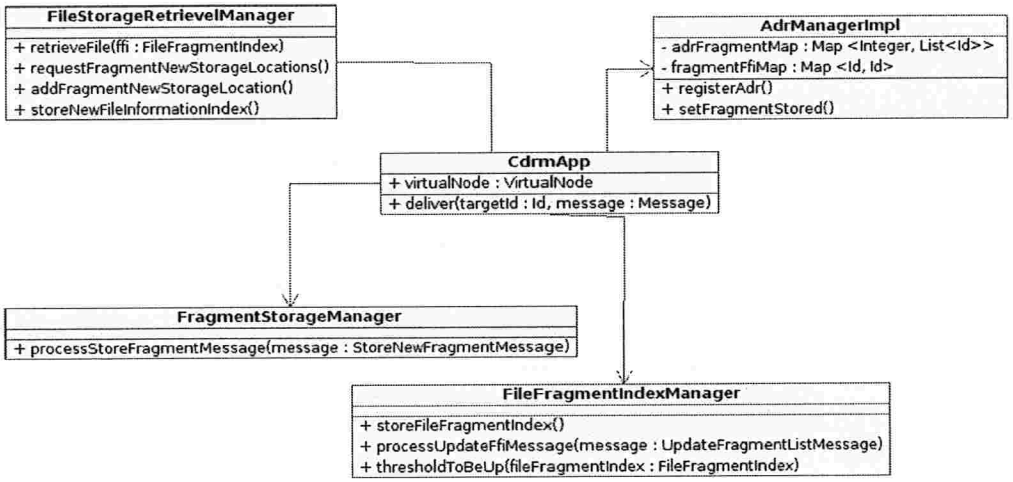

Figura 6.3: Diagrama de classes associadas ao processo de substituição de fragmentos

A figura 6.3 apresenta as classes que já estavam definidas no OppStore para operações de armazenamento e recuperação de arquivos. Nesta figura mostramos os atributos que foram incluídos para permitir o funcionamento do mecanismo de substituição de fragmentos. A seguir, descrevemos as classes apresentadas na figura 6.3.

A classe `CdrmApp`, recebe e processa as mensagens recebidas pelo CDRM provindas de CDRMs de outros aglomerados. As mensagens que processa podem ser: (1) mensagem de requisição de

atualização dos identificadores dos fragmentos contidos no FFI, esta mensagem é repassada para a classe `FileFragmentIndexManager` que é a responsável pelo gerenciamento dos FFIs do aglomerado. (2) mensagem de requisição de endereço de um ADR para armazenamento de um novo fragmento, que é repassada para a classe `AdrManagerImpl`, responsável por gerenciar os ADRs do aglomerado.

A interface `AccessBroker` define as operações `uploadFragments` e `downloadFragments` para armazenamento e descarga dos fragmentos, além de operações para verificação de tempo de validade dos fragmentos. A classe `FileStorageRetrievelManager` requisita à classe `RequestSourceManager`, que mantém uma lista de *access broker*, uma *access broker* passando como parâmetro o numero de requisição. Após obtê-lo requisita à classe `AccessBroker` a chamada do método `retrieveFragment` que recupera os novos fragmentos que devem ser armazenados depois de reconstruir o arquivo original.

Após obter os novos fragmentos, a classe `FileStorageRetrievelManager` faz chamado do método `requestFragmentNewStorageLocations` que permite criar mensagens da classe `StoreNewFragmentMessage` contendo como informação o identificador do novo fragmento e o tamanho do mesmo. Depois faz uso da classe `VirtualNode` para fazer o roteamento da mensagem até o CDRM que será responsável por armazenar os novos fragmentos.

6.2.2 Access Broker

O *access broker* precisa se comunicar com o CDRM do seu aglomerado para submeter requisições para o armazenamento e recuperação de arquivos. Assim, o *access broker* invoca operações da interface CORBA, `CdrmRequestManager`, definida no CDRM é utilizada pelo *access broker* para submeter requisições para o armazenamento e recuperação de dados armazenados remotamente. O CDRM processa as operações de modo assíncrono e as respostas são enviadas de volta ao *access broker*. Durante o processo de armazenamento, o CDRM processa requisições para obtenção de uma lista de endereços de ADRs para armazenar fragmentos invocando operações da classe `AdrManagerImpl` como `getAdrAdress` que retorna endereços de ADRs disponíveis para o armazenamento do fragmentos.

Como parte do mecanismo de substituição de fragmentos perdidos, o *access broker* é usado para recuperar os fragmentos perdidos logo após reconstruir o arquivo. Depois de obter os novos fragmentos, *access broker* se comunica com o CDRM para obter os novos endereços de ADRs para armazenar os novos fragmentos. Logo após conseguir todos os endereços de ADRs relativos aos novos fragmentos, o CDRM utiliza a classe `FileStorageRetrievelManager` para requisitar a *access broker* armazenar os novos fragmentos chamando o método `reloadFragments`. O método `reloadFragments` permite o armazenamento dos fragmentos nos ADRs dos aglomerados correspondentes com os endereços obtidos pelo *access broker*. Terminado o armazenamento dos fragmentos,

notifica ao CDRM a finalização do processo. O CDRM será o responsável por finalizar o processo substituição atualizando os FFIs com os novos identificadores de fragmentos assim como seus endereços.

6.3 Execução do mecanismo de substituição de fragmentos perdidos

Implementamos o mecanismo para a substituição de fragmentos perdidos definido na seção 5.2.1. Simplificando, na execução do mecanismo descrevemos os seguintes passos:

1. Atualizamos a lista de chaves dos fragmentos armazenado nos FFIs.
2. Verificamos se o Limiar foi alcançado.
3. Caso o Limiar foi alcançado, reconstruímos o arquivo original e a partir dele recuperamos os fragmentos que foram perdidos.
4. Após obter os fragmentos, iniciamos o processo de armazenamento dos mesmos em outros ADRs.
5. O processo finaliza após atualizar os FFIs com as novas informações.

A seguir detalhamos como é realizado o processo de substituição de fragmentos perdidos.

O processo de substituição de fragmentos perdidos inicia quando o CDRM verifica a falha de um ADR do seu aglomerado.

(1) Atualização da lista de chaves dos fragmentos armazenadas nos FFIs.

O Processo inicia quando o CDRM responsável pelo ADR que apresenta indisponibilidade verifica (1)quais fragmentos estavam armazenados no ADR perdido, (2)os identificadores dos FFIs que devem ser atualizados. Por cada fragmento que estava armazenado no ADR morto, o CDRM cria uma mensagem do tipo `UpdateFragmentListMessage` contendo como informações o identificador de fragmento e o identificador do FFI que deve ser atualizado. Este processo é mostrado na figura 6.4.

Como se mostra na figura 6.4, após criar a mensagem de atualização do FFI, o CDRM inicia o roteamento da mesma até encontrar o CDRM responsável pelo seu armazenamento. Quando a mensagem chega até CDRM responsável por armazená-lo, o CDRM recebe a mensagem e repassa à classe `FileFragmentIndexManager` que a processa chamando o método `processUpdateFfiMessage` onde se realiza a atualização das informações do FFI.

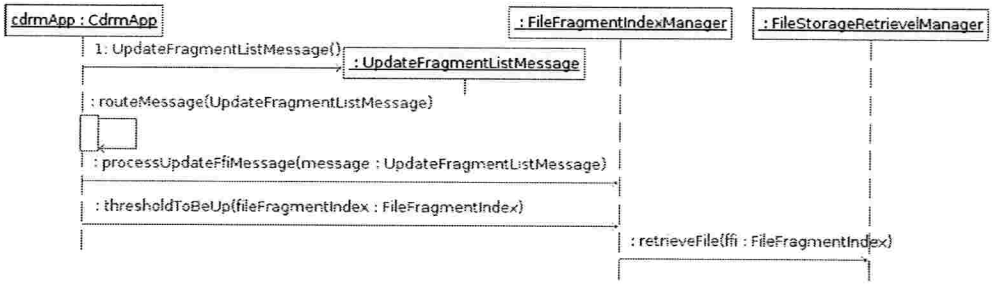

Figura 6.4: Diagrama de sequência do processo de substituição de fragmentos(1)

(2) Verificação do Limiar

Depois de atualizar as informações do FFI, o CDRM verifica se o Limiar foi alcançado chamando o método `tresholdToBeUp` da classe `FileFragmentIndex`. Se o Limiar foi alcançado, dá inicio o processo de substituição de fragmentos perdidos.

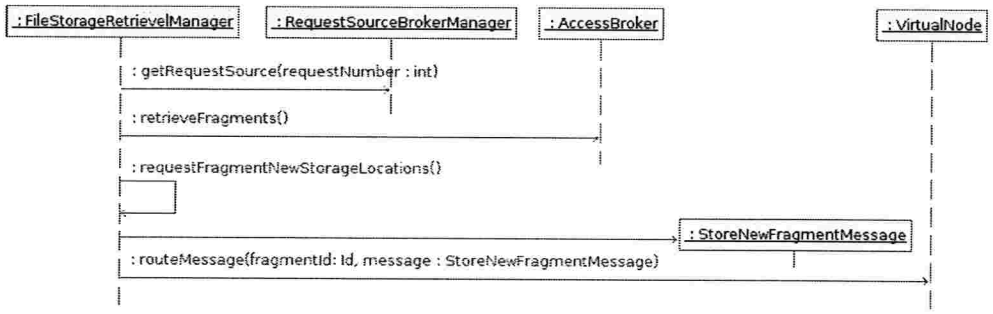

Figura 6.5: Diagrama de sequência do processo de substituição de fragmentos(2)

(3) Recuperação dos fragmentos perdidos a partir da recuperação do arquivo original

Como mostramos no diagrama da figura 6.5, o CDRM através da classe `FileStorageRetrieval-Manager` chama o método `retrieveFile` onde se realiza o processo de reconstrução do arquivo original quando a classe `AccessBroker` chama o método `retrieveFragment` que devolve os fragmentos perdidos.

(4) Processo de armazenamento dos novos fragmentos em outros ADRs

Depois de obter os novos fragmentos, a classe `FileStorageRetrievalManager` utiliza o método `requestFragmentNewLocations` onde é criado uma mensagem do tipo `StorageNewFragmentMessage` por cada novo fragmento recuperado. O processo é mostrado na figura 6.5.

Após criar a mensagem de tipo `StorageNewFragmentMessage`(figura 6.5), que é usada para procurar novos endereços para o armazenamento dos fragmentos, o CDRM roteia a mensagem até cada CDRM responsável pelo futuro armazenamento de cada fragmento. Quando a mensagem chega até o CDRM responsável pelo aglomerado onde se encontra o ADR, como podemos observar na figura 6.6, o CDRM repassa a mensagem para a classe `FragmentStorageManager` que a processada chamando o método `processStorageFragmentMessage` que recebe como parâmetro a mensagem, dentro do método `processStorageFragmentMessage` a mensagem é atualizada com um endereço de ADR válido para armazenamento.

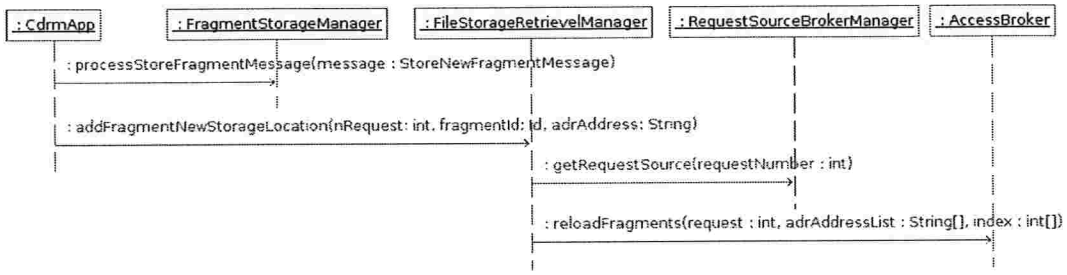

Figura 6.6: Diagrama de sequência do processo de substituição de fragmentos(3)

Depois de obter o endereço para o fragmento, a mensagem é retornada ao CDRM origem. O CDRM que recebe a mensagem de retorno repassa a mensagem à classe `FileStorageRetrievelManager` que a processa chamando o método `addFragmentNewStorageLocation`, onde os identificadores dos novos fragmentos são associados a seus endereços. Quando todos os endereços para os novos fragmentos são completados, a classe `AccessBroker` é requerida para realizar a transferência dos fragmentos até os ADRs responsáveis por armazená-los chamando o metodo `reloadFragments`.

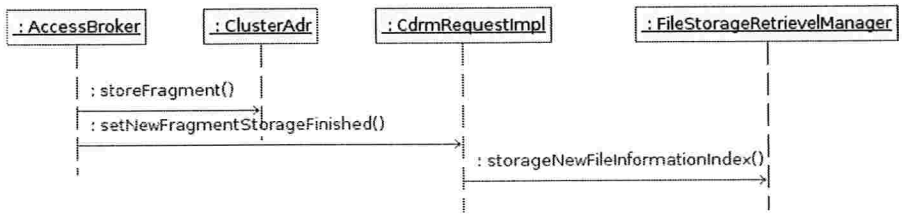

Figura 6.7: Diagrama de sequência do processo de substituição de fragmentos(4)

(5) Atualização dos FFIs com as novas informações

Concluída a transferência dos arquivos até os ADRs, como mostramos na figura 6.7, o CDRM utiliza a classe `CdrmRequesImpl` que chama o método `setNewFragmentStorageFinished` para concluir a atualização dos FFIs com as novas informações, onde a classe `FileStorageRetrievelManager` chama o método `storageNewFileInformationIndex` que cria uma mensagem da classe `FileFragmentIndexMessage` de tipo `Update`. Concluída a criação da mensagem de atualização do

FFI com os novos identificadores de fragmentos, o CDRM roteia a mensagem até o CDRM responsável pelo armazenamento do FFI correspondente.

Quando a mensagem de atualização do FFI chega até o CDRM responsável pelo seu armazenamento, o CDRM recebe a mensagem e a repassa para a classe `FileFragmentIndexManager` que a processa chamando o método `processFileIndexMessage` onde é realizado o processo de atualização do FFI com as novas informações. Concluída a atualização a mensagem é devolvida ao CDRM origem que finaliza o processo atualizando as réplicas dos FFIs.

6.3.1 Usando uma cópia local de fragmento

Como foi descrito na seção 5.3, incluímos uma outra abordagem para tratamento das falhas que acontecem nos ADR e comprometem a disponibilidade dos fragmentos armazenados. Este tratamento implica o armazenando de uma cópia adicional de cada fragmento em alguma outra máquina do mesmo aglomerado. Na ocorrência de alguma falha, o CDRM do aglomerado é responsável por substituir o fragmento perdido usando a cópia armazenada para armazenar uma nova cópia do fragmento em outra máquina do mesmo aglomerado.

Para realizar o processo de substituição de fragmentos baseado em uma cópia local, em primeiro lugar modificamos o processo de armazenamento de fragmentos para assim permitir o armazenamento de uma cópia adicional do fragmento em outra máquina do mesmo aglomerado.

A seguir descrevemos o processo de substituição de fragmentos perdidos baseado numa cópia local de fragmento armazenado numa outra máquina do mesmo aglomerado.

(1)Processo de armazenamento de fragmentos

O processo de armazenamento e recuperação de arquivos são realizados de modo assíncrono, com o *access broker* chamando operações da interface CORBA `CdrmRequestManager`, implementada pela classe `CdrmRequestManagerImpl`.

O método `requestFileStorage` permite que um *access broker* solicite o armazenamento de um arquivo no OppStore. O *access broker* envia como parâmetros os identificadores do arquivo e dos fragmentos. O tamanho do arquivo e os fragmentos permitem que o CDRM escolha o ADR que contenham espaço em disco suficiente para os fragmentos.

Ao receber uma requisição de armazenamento para obter os endereços de ADRs para o armazenamento de fragmentos, a classe `CdrmRequestImpl` cria uma mensagem de tipo `StoreFragmentMessage` para cada fragmento e repassa à classe `VirtualNode` para que realize o roteamento da

mesma até o CDRM responsável pelo identificador do fragmento.

Quando o CDRM recebe a mensagem `StoreFragmentMessage` repassa à classe `FragmentStorageManager` que utiliza o método `processStoreFragmentMessage` para encontrar dois endereços de ADRs para o fragmento no mesmo aglomerado. Após conseguir os dois endereços, a mensagem é atualizada com os endereços dos ADRs e enviada de volta ao CDRM. O CDRM que recebe a mensagem atualizada contendo os dois endereços repassa a mesma para a classe `FileStorageRetrievalManager` que a processa chamando o método `addFragmentListStorageLocation`.

Após receber as respostas para as mensagens solicitando os endereços dos ADRs para o armazenamento de todos os fragmentos, o CDRM chama a operação `uploadFragments` do *access broker* passando como parâmetro a lista de endereços de ADRs. O *access broker* realiza então a transferência dos fragmentos para os ADRs. Terminada a transferência, o *access broker* chama o método `setFragmentStorageFinished` do CDRM, enviando a lista dos fragmentos armazenados com sucesso. O CDRM cria o FFI do arquivo, então cria a mensagem do tipo `FileFragmentIndexMessage`, para ser roteada até o CDRM que será responsável pelo seu armazenamento. Após receber a confirmação de armazenamento, o CDRM chama a operação `setFileStorageRequestCompleted` da interface `AccessBroker` para informar que o armazenamento foi finalizado com sucesso.

(2) Substituição de fragmentos perdidos baseado na cópia local

O processo de substituição de fragmentos perdidos inicia quando o CDRM verifica a falha de um ADR do seu aglomerado. É considerado que um ADR apresenta falha quando não se comunica mais com o CDRM do aglomerado dentro um tempo estabelecido.

O CDRM responsável pelo ADR que apresenta indisponibilidade recupera a lista dos fragmentos que estavam armazenados no ADR perdido. O CDRM utiliza a classe `AdrManagerImpl` que chama o método `getSecondAdrAdress` que retorna o segundo endereço armazenado para o fragmento. Após conseguir o endereço do ADR onde se encontra a cópia do fragmento, e chamada a operação `replaceFragment` da classe `AdrManagerImpl` onde a partir do endereço é recuperado a cópia do fragmento. Após obter a cópia do fragmento e chamada a operação `getAdrAddress` que devolve um novo endereço válido para o armazenamento do fragmento.

Após o CDRM obter os endereços válidos para o armazenamento dos novos fragmentos, o novos fragmentos são armazenados. Terminado o armazenado dos fragmentos, é gerada uma mensagem do tipo `UpdateFragmentAddressMessage` que e roteada pela classe `VirtualNode` até o CDRM responsável pelo armazenamento do FFI para sua posterior atualização.

No tratamento local de falhas poderia-se tentar o armazenamento de dois endereços disponíveis

para cada fragmento contido no FFI, mas nesse caso teríamos que utilizar mais uma mensagem de atualização desses endereços toda vez que algum fragmento é perdido e nosso interesse é diminuir a quantidade de mensagens que trafegam pela rede toda vez que iniciamos o tratamento das falhas.

6.4 Resumo

Neste capítulo foram apresentados de maneira geral as implementações dos principais componentes do OppStore. Num primeiro caso apresentamos brevemente a implementação dos principais componentes, para depois descrever a implementação das novas classes, atributos e métodos incrementadas para a realização do tratamento das falhas.

A implementação do primeiro mecanismo de substituição de fragmentos perdidos, implica a criação e tratamento de novas mensagens para realizar o processo de substituição de fragmentos baseado na reconstrução do arquivo original. No caso do segundo mecanismo, foi modificado o processo de armazenamento para permitir o armazenamento de um segundo fragmento no mesmo aglomerado.

A seguir apresentamos os resultados obtidos após a avaliação dos mecanismos propostos.

Capítulo 7

Experimentos

Como foi apresentado no capítulo 5, propusemos 2 mecanismos para assim manter a disponibilidade dos fragmentos armazenados na presença de falhas nas máquinas da grade que os armazenam. O primeiro mecanismo reconstrói o arquivo original para depois obter e substituir os fragmentos que ficaram perdidos. O segundo mecanismo mantém uma cópia adicional de cada fragmento armazenada no mesmo aglomerado e permite a recuperação de um fragmento perdido a partir da cópia armazenada.

Realizamos a avaliação dos mecanismos propostos, para assim estabelecer o desempenho dos mesmos. O desempenho dos mecanismos são estabelecidos através de medições da quantidade de mensagens extras geradas (mensagens para atualização de FFIs, requisição de novos endereços de ADRs e para transferência de dados) durante o normal funcionamento do OppStore. Além das mensagens, consideramos o incremento do tráfego de dados da rede como consequência do armazenamento dos novos fragmentos.

Verificamos o impacto da variação dos níveis de redundância do IDA, usados para a geração de fragmentos, sobre a execução de ambos mecanismos. Finalmente verificamos também o impacto de estabelecer mudanças no LIMIAR no mecanismo de substituição de fragmentos perdidos baseado na reconstrução de fragmentos perdidos.

7.1 Simulações

O objetivo das simulações é avaliar o desempenho dos mecanismos em uma grade simulada de grande escala. A vantagem da simulação é que podemos estabelecer grades compostas por centenas de aglomerados e máquinas compartilhadas. Após a finalização do armazenamento dos arquivos podemos simular um grande conjunto de falhas nas máquinas compartilhadas que armazenam os fragmentos. Iniciamos descrevendo os parâmetros de simulação para depois apresentar os resultados obtidos.

7.1.1 Ambiente de simulação

Realizamos as operações de armazenamento e recuperação de arquivos para uma grade simulada com 30 aglomerados. Cada aglomerado é iniciado com um número fixo de ADRs, definindo inicialmente 50 ADRs por aglomerado. Fizemos a suposição que as máquinas inicialmente possuem a mesma quantidade de espaço livre.

Realizamos o armazenamento de 900 arquivos na grade simulada, terminado o armazenamento dos arquivos iniciamos o padrão de falhas definido que acontecerão nas máquinas dos aglomerados. O objetivo da simulação é avaliar entre outras coisas a quantidade de fragmentos que podem ser recuperados por cada arquivo armazenado após as falhas apresentadas no sistema.

Operações de codificação, transferência e armazenamento de arquivos são simuladas uma vez que nosso objetivo é avaliar os mecanismos de tolerância a falhas propostos em função da distribuição dos fragmentos e da sua disponibilidade frente às falhas apresentadas.

- Simulamos um período de um mês de operação do sistema utilizando os padrões de uso das máquinas descritos na tabela 7.1. Nessa simulação de períodos longos realizamos requisições para armazenamento e recuperação de arquivos e coletamos informações sobre a quantidade de requisições para a recuperação de arquivos que foram bem sucedidas.
- Definimos a saída de n máquinas em todos os aglomerados, a quantidade de máquinas que deixarão o sistema serão no máximo a metade do total de máquinas que compõem cada aglomerado. Estabelecemos padrões de saída das máquinas, inicialmente estabelecemos a saída progressiva de só uma máquina de um aglomerado por cada período de tempo t. Num segundo caso permitimos a saída de várias máquinas do mesmo aglomerado no mesmo tempo t para observar o comportamento dos mecanismos.
- Avaliamos a execução dos mecanismos em função do nível de redundância incluído para realizar a codificação dos arquivos.

	padrão1	padrão2	padrão3
média diurna de tempo ligado	90.0%	60.0%	90.0%
média noturna de tempo ligado	82.5%	45.0%	82.5%
média diurna de tempo ocioso	60.0%	25.0%	40.0%
média noturna de tempo ocioso	80.0%	40.0%	70.0%

Tabela 7.1: Padrões de uso de máquinas compartilhadas

A tabela 7.1 mostra os padrões de uso de máquinas que serão usadas para a execução dos experimentos, o tempo ligado significa que a máquina está sendo usada por algum usuário. As máquinas de tempo ocioso são as máquinas que estão ligadas mas seus recursos computacionais não estão sendo usadas no momento.

Dado que requisições para a recuperação de fragmentos são feitas exclusivamente para máquinas que estejam em tempos ociosos, pode se encontrar a situação de que alguma das máquinas não permita a recuperação do fragmento por ela armazenado. Essa situação dificulta a reconstrução do arquivo e neste caso pode ser responsável pela indisponibilidade temporária ou definitiva do arquivo, dado que o mecanismo não consegue uma quantidade de fragmentos suficientes para reconstruir o arquivo armazenado e obter os fragmentos perdidos.

Logo após iniciado o sistema, o período entre o início da simulação e cada requisição periódica de recuperação de arquivos, n máquinas apresentarão falhas ou ficarão indisponíveis. A quantidade de máquinas que ficarão indisponíveis é definida antes do início dos experimentos.

7.1.2 Coletando informações

Durante a simulação do funcionamento do OppStore realizamos a coleta de informações relacionadas com a saída das máquinas dos aglomerados.

Fazemos uso da classe StatisticCollector, que permite coletar e armazenar as informações seguintes:

- Coletamos informações sobre a quantidade de vezes que cada mecanismo é iniciado. O início do mecanismo é afetado pelo nível de redundância estabelecido para a codificação dos arquivos. A quantidade de vezes que o mecanismo de substituição de fragmentos perdidos é iniciado está relacionada com o estabelecimento do limiar de reconstrução. No caso do mecanismo de substituição de fragmentos baseado na reconstrução do arquivo original, um limiar muito agressivo pode implicar iniciar esse mecanismo um número maior de vezes que no caso de usar um limiar mais conservativo.

  Falar de um limiar muito conservativo é equivalente a falar do número de fragmentos estritamente necessários para recuperar o arquivo original, passado esse limiar não será possível recuperar o arquivo. Um limiar muito agressivo, é um valor maior do que o número de fragmentos estritamente necessários para recuperar o arquivo original.

- Informações sobre a quantidade de fragmentos gerados na execução dos mecanismos. A quantidade de fragmentos gerados no tempo de simulação dá uma idéia da carga extra que é gerada na execução dos mecanismos.

- Informações sobre a quantidade de mensagens geradas. A quantidade de mensagens geradas se traduz em sobrecarga de dados numa operação normal do sistema sem uso de nenhum mecanismo.

- Informações sobre a quantidade de requisições bem sucedidas. Verificaremos a quantidade de requisições que podem ser atendidas com sucesso. Nem sempre será possível recuperar os fragmentos suficientes para a reconstrução do arquivo original para substituir os fragmentos

que foram perdidos, isso levando em consideração a indisponibilidade das máquinas.

7.2 Experimentos e avaliação

Configuramos a implementação do *access broker* para permitirmos codificar os arquivos levando em consideração dois diferentes quantidades de fragmentos para codificação dos arquivos nos experimentos realizados. O objetivo de experimentar com diferentes quantidades de fragmentos é estabelecer uma comparação da relação existente entre o nível de redundância escolhido e sua implicação no comportamento de cada um dos mecanismos.

Num primeiro experimento iniciamos a simulação do sistema e realizamos as operações para o armazenamento dos arquivos. Após terminado o armazenamento, permitimos a saída de até 50% das máquinas de cada um dos aglomerados que conformam a grade. O objetivo deste experimento é avaliar o comportamento do OppStore sem o uso de algum mecanismo de tolerância a falhas.

O segundo experimento tem como objetivo a avaliação do primeiro mecanismo proposto na seção 5.2.1. Este experimento esta dividido em duas partes.

Na primeira parte do experimento codificamos os arquivos em 6 fragmentos redundantes sendo 3 fragmentos suficientes para recuperar o arquivo original. Iniciamos o armazenamento dos arquivos e logo após terminado o armazenamento simulamos a saída das máquinas. Neste experimento avaliamos o comportamento do mecanismo quando são considerados 3 limiares diferentes que permitem o início do mecanismo. O primeiro limiar usado é o um limiar com o valor limite (3) que permite recuperar o arquivo. O valor do segundo limiar usado é 4 e o terceiro o valor 5. Iniciamos o sistema, coletamos dados e avaliamos o comportamento do mecanismo.

Na segunda parte do experimento avaliamos o comportamento do mecanismo quando no sistema são armazenados os arquivos codificados em 12 fragmentos redundantes sendo 6 fragmentos necessários para recuperar o arquivo original. Avaliamos o desempenho do mecanismo estabelecendo 4 limiares diferentes que indicam o início do mecanismo. O primeiro valor do limiar usado é o valor limite (6), logo são realizados mais 3 experimentos usando limiares com valores 7, 8 e 9. Cada experimento é iniciado para usar um limiar diferente e de valor crescente ao experimento anterior.

O terceiro experimento tem como objetivo a avaliação do segundo mecanismo proposto na seção 5.3.1. Assim como no primeiro experimento, consideramos o experimento composto por duas partes. Na primeira parte realizamos o experimento para avaliar o comportamento do mecanismo quando os arquivos são codificados em 6 fragmentos redundantes onde 3 fragmentos são necessários para recuperar o arquivo original. Na segunda parte do experimento, codificamos os arquivos em 12

fragmentos redundantes onde 6 fragmentos são suficientes para recuperar o arquivo original. Em cada um dos experimentos fazemos uso de diferentes limiares assim como foi feito para a avaliação do primeiro mecanismo.

A avaliação dos mecanismos é apresentada nas subseções a seguir. Na subseção 7.2.1, apresentamos os resultados obtidos levando em consideração o índice de disponibilidade de fragmentos como resultados dos experimentos propostos. Na subseção 7.2.2 consideramos a quantidade de mensagens que trafegam como consequência da execução dos mecanismos.

7.2.1 Avaliando a disponibilidade dos fragmentos armazenados

Para avaliar o índice de disponibilidade que provêem os mecanismos, iniciamos a simulação do sistema composto com 30 aglomerados onde cada aglomerado está composto por 50 máquinas (ADRs) destinadas para o armazenamento dos arquivos.

Numa primeira avaliação simulamos o armazenamento de 900 arquivos codificados em 6 fragmentos redundantes dos quais 3 são suficientes para recuperar o arquivo original. Após terminado o armazenamento, simulamos as falhas em todos os aglomerados que compõem o sistema. Estas falhas implicam a saída do 50% das máquinas destinadas ao armazenamento dos fragmentos em cada aglomerado. O número total de falhas de ADRs simuladas é de 750.

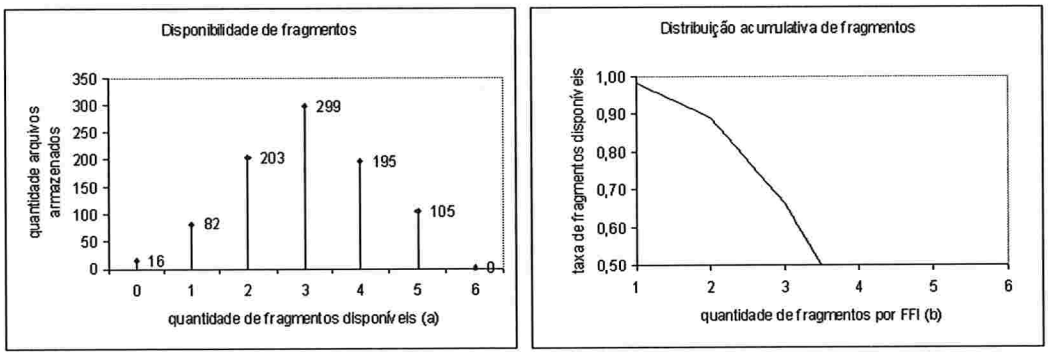

Figura 7.1: Disponibilidade de fragmentos por arquivos armazenados

A figura 7.1 mostra os resultados obtidos da execução do sistema armazenando os arquivos codificados em 6 fragmentos e sem o uso de mecanismo nenhum para a recuperação dos mesmos. Nessa figura 7.1 (a) podemos observar que o 33.44% (301 fragmentos) dos arquivos armazenados foram perdidos como consequência da saída de 750 ADRs do sistema. Podemos observar também que um 30.22% (299 fragmentos) dos arquivos estão no limite de ficar indisponíveis porque eles precisam pelo menos 3 fragmentos para recuperar o arquivo original.

Utilizando o primeiro mecanismo

Neste experimento avaliamos o comportamento do mecanismo quando são considerados 3 limiares diferentes que permitem o início do mecanismo. O primeiro limiar usado é o um limiar com o valor limite 3, que permite recuperar o arquivo. O valor do segundo limiar usado é 4 e o terceiro o valor 5. Iniciamos o sistema, coletamos dados e avaliamos o comportamento do mecanismo.

A tabela 7.2 mostra os resultados após usar o primeiro mecanismo, nela podemos observar a quantidade de fragmentos por FFI que ficaram disponíveis após aplicar o primeiro mecanismo e os diferentes limiares usados.

fragmentos	limiar 3	limiar4	limiar5
0	0.00 %	0.00 %	0.00 %
1	0.22 %	0.00 %	0.00 %
2	1.00 %	0.56 %	0.33 %
3	4.89 %	3.56 %	3.22 %
4	19.89 %	11.67 %	9.67 %
5	38.22 %	35.22 %	23.56 %
6	35.78 %	49.00 %	63.22 %

Tabela 7.2: Disponibilidade de fragmentos por FFI 6-3

A figura 7.2 (a) mostra os resultados obtidos utilizando o primeiro mecanismo de substituição dos fragmentos baseado na reconstrução do arquivo original utilizando um limiar de recuperação de valor 3. O limiar usado é o valor limite que permite manter a disponibilidade do arquivo. Podemos observar que o 1.22% (11 fragmentos) dos arquivos foram perdidos devido a perda de um número de fragmentos superior ao valor limite necessário para mantê-lo disponível, além disso percebemos que o 4.89% (44 fragmentos) dos arquivos estão no limite de ficar também indisponíveis.

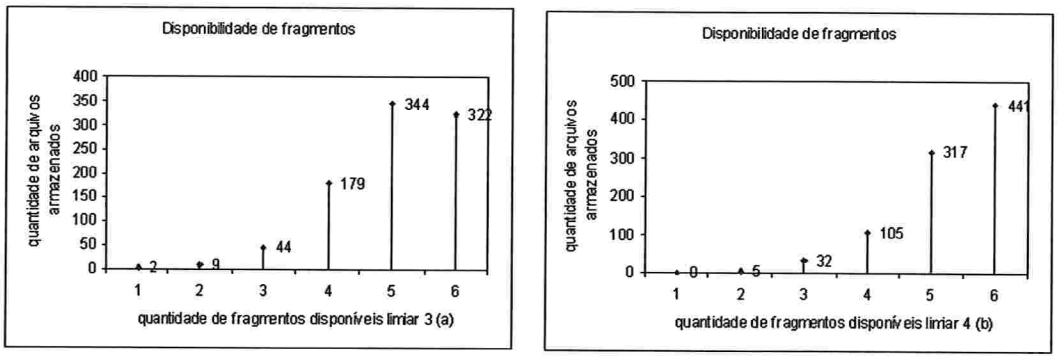

Figura 7.2: Disponibilidade de fragmentos por arquivos armazenados usando primeiro mecanismo

Na figura 7.2 (b) observamos a disponibilidade dos fragmentos utilizando um limiar de valor 4. Utilizando este limiar observamos que o 0.56% (5) dos arquivos armazenados foram perdidos como consequência da perda de fragmentos suficientes para recuperar o arquivo original. O 3.56% (32)

dos arquivos estão no limite de ficar indisponíveis.

Na figura 7.3 (a) e na tabela 7.2, observamos a disponibilidade dos fragmentos utilizando um limiar de valor 5. A utilização de um limiar de valor 5 implica que um 0.33% (3 fragmentos) dos arquivos são perdidos. Também vemos que o 3.22% (29 fragmentos) dos arquivos estão no limite de ficar indisponíveis. Na figura 7.3 (b) podemos observar a taxa de fragmentos disponíveis para todos os limiares adotados. Percebemos que quanto maior seja o limiar adotado, maior será a quantidade de fragmentos disponíveis por FFI.

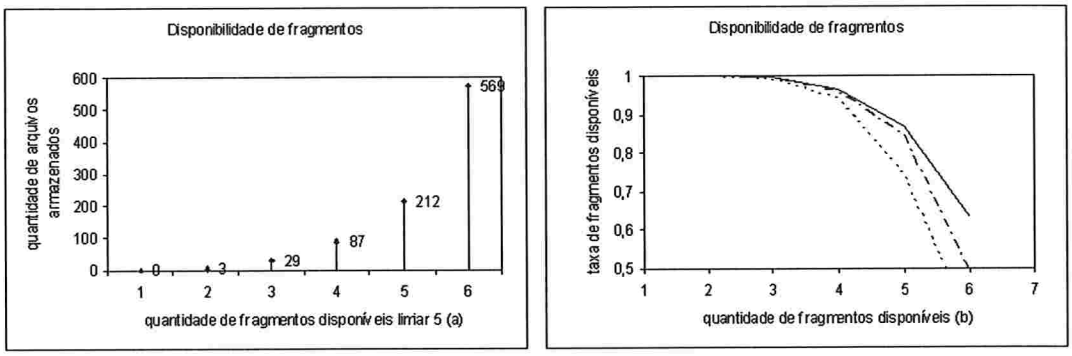

Figura 7.3: Disponibilidade de fragmentos por arquivos armazenados usando primeiro mecanismo

Podemos observar uma melhoria na disponibilidade dos fragmentos quando incrementamos o valor do limiar, esta melhoria implica também o incremento das chamadas para o início do mecanismo. Toda vez que a lista de fragmentos de um FFI é atualizada excluindo o fragmento perdido como consequência da saída do ADR que o armazenava, é verificado se o valor do limiar é alcançado. Se o limiar é alcançado, o mecanismo é iniciado. Cabe ressaltar que mesmo aplicando um limiar muito agressivo (valor 5), ainda temos perda de arquivos. Esta perda de arquivos esta relacionada com a quantidade de falhas que apresenta o sistema. À medida que o sistema apresenta uma grande quantidade de falhas e os arquivos estão codificados usando um baixo nível de redundância, a disponibilidade é comprometida.

Na segunda parte da avaliação do primeiro mecanismo, simulamos também o armazenamento de 900 arquivos agora codificados em 12 fragmentos redundantes dos quais 6 são suficientes para recuperar o arquivo original. Depois de terminado o armazenamento, simulamos as falhas em todos os aglomerados que compõem o sistema. As falhas implicam a saída do 50% das máquinas destinadas ao armazenamento dos fragmentos em cada aglomerado. O total de mortes de ADRs simuladas é a mesma (750).

Avaliamos o comportamento do mecanismo quando no sistema são armazenados os arquivos codificados em 12 fragmentos redundantes sendo 6 fragmentos necessários para recuperar o arquivo

original. Avaliamos o desempenho do mecanismo estabelecendo 4 limiares diferentes que indicam o início do mecanismo. O primeiro valor do limiar usado é o valor limite (6), logo são realizados mais 3 experimentos usando limiares com valores 7, 8 e 9.

A tabela 7.3 mostra os resultados após usar o primeiro mecanismo, nela podemos observar a quantidade de fragmentos por FFI que ficaram após aplicar o primeiro mecanismo e os diferentes limiares usados.

fragmentos	limiar 6	limiar7	limiar8	limiar9
0	0.00 %	0.00 %	0.00 %	0.00 %
1	0.00 %	0.00 %	0.00 %	0.00 %
2	0.00 %	0.00 %	0.00 %	0.00 %
3	0.00 %	0.00 %	0.00 %	0.00 %
4	0.11 %	0.00 %	0.00 %	0.00 %
5	1.11 %	0.44 %	0.00 %	0.00 %
6	4.78 %	1.56 %	0.67 %	0.11 %
7	23.44 %	6.56 %	1.44 %	0.22 %
8	20.22 %	19.22 %	3.78 %	1.00 %
9	13.33 %	19.22 %	26.56 %	5.00 %
10	11.22 %	17.56 %	22.56 %	34.56 %
11	14.56 %	20.22 %	25.44 %	33.56 %
12	11.22 %	15.22 %	19.56 %	25.56 %

Tabela 7.3: Disponibilidade de fragmentos por FFI 12-6

Os dados coletados mostraram que o 1.22% dos arquivos ficaram indisponíveis como consequência da saída de ADRs do sistema, percebemos também que o 4.78% dos arquivos estão no limite de ficar perdidos. Isso é o resultado de aplicar um limiar muito agressivo para o início do mecanismo, neste caso o valor limite 6. A aplicação deste limiar implica a chamada de 505 vezes para o início do mecanismo.

Aplicar um limiar de valor 7, melhora a chance de recuperar os arquivos, dado que agora o 0.44% dos arquivos ficaram indisponíveis e o 1.56% dos arquivos estão no limite de ficar perdidos. O uso deste limiar melhora a disponibilidade, mas também implica a chamada de 695 vezes para o início do mecanismo.

A aplicação do mecanismo para um limiar de valor 8 melhora a disponibilidade apresentada nos casos anteriores. Neste caso percebemos que nenhum arquivo tem chance de ser perdido e que o 0.67% dos arquivos tem pelo menos 6 fragmentos disponíveis ou estão no limite de ficar perdidos. O uso deste limiar implica iniciar o mecanismo 937 vezes. Aplicar um limiar de valor 9 incrementa a chance de que nenhum arquivo seja perdido e só o 0.11% está em risco de ser perdido quando o mecanismo foi inciado 1384 vezes.

Logo após avaliar o mecanismo usando um maior número de fragmentos na codificação e diferentes limiares, percebemos que melhorou muito do que o uso dos limiares e a redundância usada na

primeira parte dos experimentos. Isso porque temos um maior número de fragmentos armazenados disponíveis no sistema. Podemos dizer que ante a presença de muitas falhas temos que melhorar o nível de redundância usado para a codificação dos arquivos e também melhorar a escolha do limiar.

A figura 7.4 mostra a probabilidade de encontrar um número de fragmentos maior do que o limiar adotado, podemos perceber como é a melhora da distribuição dos fragmentos enquanto o limiar é incrementado. Na figura 7.4 observamos a comparativa da distribuição do primeiro limiar comparado com os outros limiares adotados, percebemos que o incremento do limiar melhora a probabilidade de encontrar um número maior de fragmentos para recuperar os arquivos

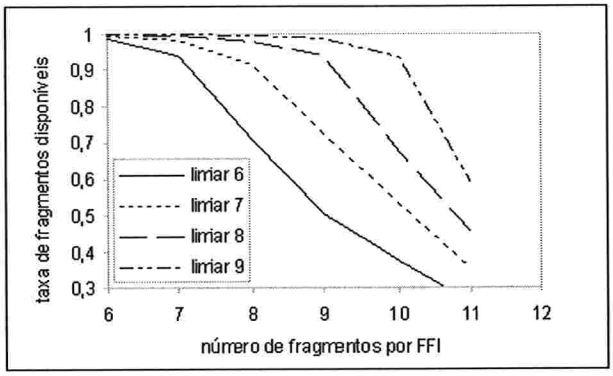

Figura 7.4: Taxa de fragmentos disponíveis por arquivos armazenados usando o primeiro mecanismo

Utilizando o segundo mecanismo

Com a execução do segundo mecanismo para o tratamento das falhas com os dois diferentes níveis de redundância apresentados conseguimos manter o 100% de FFIs com um número de fragmentos superior a 5 fragmentos. Isso acontece porque ante a presença de uma falha no sistema, o segundo mecanismo substitui imediatamente o fragmento perdido armazenando um novo fragmento baseado na cópia armazenada no mesmo aglomerado. Para permitir isso, o segundo mecanismo requer um tratamento especial no momento do armazenamento dos fragmentos. Ele requer armazenar uma cópia extra no mesmo aglomerado, isso implica um incremento do 100% do espaço de armazenamento e do duplo do tráfego intra-aglomerado requerido para o armazenamento de fragmentos e de uma cópia extra.

Só quando as duas máquinas que contém o fragmento original e cópia respectivamente saem do sistema ao mesmo tempo, o fragmento ficará perdido. Neste caso dado o nível de redundância usado a disponibilidade do arquivo ainda é alta. Isso porque agora a disponibilidade do arquivo depende da quantidade fragmentos ainda disponíveis que ainda é alta.

7.2.2 Avaliando o número de mensagens geradas

Consideramos o número de mensagens gerados durante a execução dos mecanismos quando realizamos os experimentos para os diferentes limiares adotados.

Utilizando o primeiro mecanismo

A execução do primeiro mecanismo proposto implica a geração de mensagens para atualização de FFIs, mensagens para a requisição de FFIs que contém os endereços dos fragmentos que devem ser recuperados para reconstruir o arquivo original e recuperar os fragmentos perdidos, mensagens para a procura de novos endereços para esses fragmentos recuperados, finalmente mensagens para atualização dos FFIs com os endereços dos ADRs onde os novos fragmentos foram armazenados.

A tabela 7.4 mostra o resumo das quantidades de mensagens criadas como resultado de executar o primeiro mecanismo proposto, assim como os diferentes limiares adotados quando foi realizado o armazenamento dos arquivos usando um nível de redundância de 6 fragmentos.

mensagens	limiar 3	limiar 4	limiar 5
atualização FFI	2869	2726	2511
requisitando endereço	1686	1514	1089
atualização final FFI	488	702	1089
total mensagens	5043	4942	4689
Nro vezes início	594	849	1254

Tabela 7.4: Mensagens criadas pelo primeiro mecanismo para FFI 6-3

Como observamos na tabela 7.4, percebemos que quanto maior o limiar adotado, maior é o número de vezes que o mecanismo é iniciado, embora a quantidade de mensagens geradas destinadas para a procura de novos endereços diminua. Ao mesmo tempo o número de mensagens criadas para a atualização dos FFIs com as informações finais dos novos fragmentos também é incrementado.

fragmentos	limiar 6	limiar 7	limiar 8	limiar 9
atualização FFI	5446	5278	5072	5115
requisitando endereço	3030	3475	3748	3942
atualização final FFI	1515	2085	2811	3942
total mensagens	9991	10838	11631	12999
Nro vezes início	505	695	937	1384

Tabela 7.5: Mensagens criadas pelo primeiro mecanismo para FFI 12-6

A tabela 7.5 mostra a quantidade de mensagens criadas para ser roteadas no sistema como consequência do início do primeiro mecanismo quando os arquivos foram codificados em 12 fragmentos redundantes onde 6 fragmentos são necessários para recuperar o arquivo original. Assim como no

primeiro caso podemos observar um crescimento na quantidade de mensagens geradas quando incrementamos o valor do limiar adotado.

A figura 7.5 mostra o fator de incremento do número de chamadas ao mecanismo, podemos ver o incremento do número de chamadas ao mecanismo para tentar evitar a perda de fragmentos por possíveis indisponibilidades.

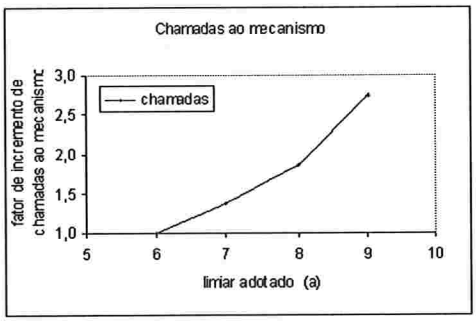

Figura 7.5: Chamadas ao mecanismo e número de mensagens criadas

Utilizando o segundo mecanismo

Dado que o segundo mecanismo não precisa reconstruir o arquivo original para obter os fragmentos perdidos, a única mensagem que ele precisa criar e rotear pela rede é uma mensagem para a atualização dos FFIs, esta mensagem de atualização tem como objetivo atualizar o endereço do fragmento contido na lista de fragmentos de cada FFI.

Na simulação do armazenamento de 900 arquivos codificados em 6 fragmentos redundantes dos quais são suficientes 3 fragmentos para recuperar o arquivo original obtemos uma carga total de 17707 mensagens trafegando pela rede, 12307 procurando endereços de rede disponíveis e 5400 mensagens destinadas para o armazenamento dos FFIs. Na realização da recuperação dos arquivos trafegam pela rede mais 1800 mensagens destinadas na procura dos FFIs respectivos. A carga geral trafegando pela rede em total de 19330 mensagens.

mensagens	6 fragmentos	12 fragmentos	24 fragmentos
atualização de FFIs	7452	15305	28648
fragmentos movimentados	7452	15305	28648

Tabela 7.6: Mensagens criadas no segundo mecanismo para diferentes codificações

A tabela 7.6 mostra a carga adicional relacionada com a substituição do fragmento a partir da cópia extra armazenada devem ser considerados 7452 mensagens relacionadas à atualização de cada FFI.

Na simulação do armazenamento de 900 arquivos codificados em 12 fragmentos redundantes dos quais são suficientes 6 fragmentos para recuperar o arquivo original obtemos uma carga total de 30632 mensagens trafegando pela rede, 25232 procurando endereços de rede disponíveis e 5400 mensagens destinadas para o armazenamento dos FFIs. Na realização da recuperação dos arquivos trafegam pela rede mais 1800 mensagens destinadas na procura dos FFIs respectivos. A carga geral trafegando pela rede em total de 32432 mensagens.

Devem ser considerados 15305 mensagens como carga adicional, relacionadas à atualização de cada FFI 7.6.

Na simulação do armazenamento de 900 arquivos codificados em 24 fragmentos redundantes dos quais são suficientes 12 fragmentos para recuperar o arquivo original obtemos uma carga total de 54433 mensagens trafegando pela rede, 49033 procurando endereços de rede disponíveis e 5400 mensagens destinadas para o armazenamento dos FFIs. Na realização da recuperação dos arquivos trafegam pela rede mais 1800 mensagens destinadas na procura dos FFIs respectivos. A carga geral trafegando pela rede em total de 56233 mensagens.

A carga adicional relacionada com a substituição do fragmento a partir da cópia extra armazenada considera 28648 mensagens relacionadas à atualização de cada FFI 7.6.

Podemos observar na tabela 7.6, as quantidades de mensagens criadas para atualização dos FFIs como resultado de codificar os arquivos em diferentes quantidades de fragmentos. Percebemos que depois de aplicar o mesmo padrão de falhas na grade que no caso do primeiro experimento, que a quantidade de mensagens para atualização de FFIs é maior enquanto maior seja a quantidade de fragmentos usados na codificação dos arquivos.

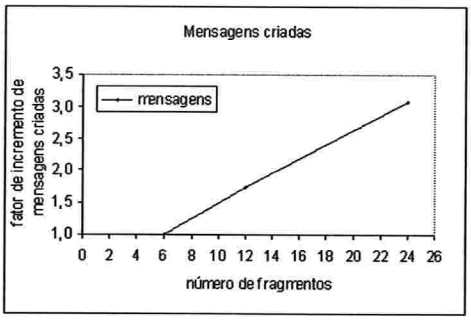

Figura 7.6: Número de total de mensagens criadas com o segundo mecanismo

Como podemos observar na figura 7.6, quanto maior seja o número fragmentos usados na codificação dos arquivos armazenados, maior será a quantidade de mensagens geradas trafegando pelo

sistema na execução do segundo mecanismo. Isso acontece porque temos mais fragmentos armazenados em cada ADR. Se toda vez que um ADR falha deve ser criada uma mensagem de atualização de um FFI por cada fragmento armazenado no ADR que apresenta a falha, o número de mensagens se incrementa por que existe o dobro de chance de atualizar os FFIs por cada fragmento perdido.

7.3 Comparação no uso dos mecanismos

Após executar e avaliar os mecanismos propostos com os diferentes níveis de redundância e limiares apresentados nas subseções 7.2.1 e 7.2.2 podemos comentar o seguinte:

7.3.1 Mecanismo de substituição de fragmentos perdidos baseado na reconstrução do arquivo original

O nível de redundância usado para a codificação dos dados guarda uma relação direta com o número de vezes que é iniciado o primeiro mecanismo. Isso porque se tem-se um baixo nível de redundância no armazenamento de arquivos e uma alta saída de ADRs do sistema. O número de vezes que será iniciado o mecanismo é maior do que o número de vezes que o mecanismo inicia se é aplicado um maior nível de redundância no armazenamento dos arquivos. Isso foi mostrado na seção 7.2.1

Verificamos também que o quanto maior seja o valor do limiar usado, maior será o número de vezes que o primeiro mecanismo é iniciado. Isso porque estamos antecipando o início do mecanismo tentando garantir que o número de fragmentos disponíveis nos FFIs seja maior do que o valor limite para recuperar o arquivo.

Cabe ressaltar que mesmo aplicando um limiar muito agressivo (valor 5) na primeira parte do experimento do primeiro mecanismo, ainda temos perda de arquivos. Esta perda de arquivos está relacionada com a quantidade de falhas que apresenta o sistema, também algumas recuperações podem falhar porque parte das máquinas que armazenam os fragmentos podem estar ocupadas no momento de requisitar o arquivo original. A medida que o sistema apresenta uma grande quantidade de falhas e os arquivos estão codificados em poucos fragmentos redundantes, a disponibilidade dos arquivos fica comprometida.

Logo após avaliar o primeiro mecanismo usando um maior índice de codificação e diferentes limiares, percebemos a melhora ao compará-los com o uso dos limiares e a redundância usada na primeira parte dos experimentos. Isso porque temos um maior número de fragmentos armazenados disponíveis no sistema. Podemos dizer que ante a presença de muitas falhas temos que melhorar o índice de codificação para os arquivos e também a escolha do limiar.

7.3.2 Tratamento de falhas baseado na cópia local de fragmento

Com a execução do segundo mecanismo para o tratamento das falhas com os dois diferentes níveis de redundância apresentados conseguimos manter um 100% de FFIs com um número de fragmentos superior a 5 fragmentos. Isso porque ante a presença de alguma falha no sistema, o segundo mecanismo substitui imediatamente o fragmento perdido armazenando um novo fragmento usando a cópia que está armazenada no mesmo aglomerado.

Para permitir uma alta disponibilidade, o segundo mecanismo requer um tratamento especial no momento do armazenamento dos fragmentos. Ele requer armazenar uma cópia extra no mesmo aglomerado, isso implica um incremento de um 100% do espaço de armazenamento e incrementar ao duplo o tráfego intra-aglomerado requerido para o armazenamento de fragmentos e de uma cópia extra.

O tráfego extra gerado pelo segundo mecanismo para o armazenamento dos fragmentos é derivado para o mesmo aglomerado ao invés de trasladar esse tráfego extra para toda a rede que compõe o sistema. Quanto maior seja a quantidade de fragmentos usados para codificar os arquivos, maior será o número de fragmentos que são armazenados nas diferentes máquinas dos aglomerados da grade. Isso de fato por que a quantidade de mensagens destinadas a atualização dos FFIS e também maior será o tráfego de dados dentro do mesmo aglomerado.

Por meio das simulações, vimos que o primeiro mecanismo proposto é o mais adequado quando consideramos o número de mensagens geradas.O segundo mecanismo, embora não precise reconstruir os fragmentos, gera um número de mensagens que é independente do grau de redundância usado.

Após avaliar os resultados obtidos dos experimentos, podemos dizer que o primeiro mecanismo proposto é o mais adequado quando codificamos os arquivos em uma quantidade alta de fragmentos, incrementando assim o nível de redundância usado. O segundo mecanismo evita a transferência de fragmentos pela rede, o que o torna vantajoso quando o fator mais importante for a quantidade de tráfego gerado na rede que conecta os aglomerados.

7.4 Resumo

Neste capítulo foram apresentados os dados experimentais obtidos como consequência de simular e avaliar o uso dos mecanismos propostos. Vimos como o nível de redundância usado é importante para manter um maior nível de disponibilidade no uso do primeiro mecanismo, embora quando comparado com o segundo mecanismo esse nível de redundância implica maior troca de mensagens no sistema. Apresentamos uma análise comparativa sobre o comportamento dos mecanismos e as implicações do seu uso no sistema. No capítulo seguinte apresentamos alguns trabalhos relacionados.

Capítulo 8

Trabalhos Relacionados

Objetivo das estratégias de replicação é manter um número de réplicas no sistema que permita manter a disponibilidade dos dados durante a presença de falhas. Pode-se considerar também uma otimização para tentar minimizar a quantidade de réplicas que devem ser criadas quando as falhas acontecem.

Diversos sistemas de armazenamento incluem estratégias de replicação para minimizar as consequências das falhas apresentadas e manter disponibilidade dos dados neles armazenados.

CFS [DKK+01] provê armazenamento de dados no nível de blocos de dados. Considera que o espaço de armazenamento não é restritivo, por isso não utiliza algum tipo de codificação para o armazenamento dos dados.

Para incrementar a disponibilidade utiliza replicação dos dados armazenados e caching de blocos populares. CFS lida com as falhas dos nós replicando os blocos de dados em aqueles nós que sucedem o nó por eles responsável. Na presença de uma falha de um nó, seu sucessor passa automaticamente a ser o responsável pelo gerenciamento dos blocos anteriormente gerenciados pelo nó que apresentou a falha. Como esse sucessor já possui uma copia dos dados, então ele poderá responder as requisições normalmente.

O objetivo de replicar os blocos é também permitir a escolha dó nó mais rápido no momento de transferir o bloco. O cliente solicita uma lista de sucessores do identificador junto com uma estimativa das latências, então o cliente escolhe o nó com menor latência. CFS não faz o tratamento para minimizar a quantidade de réplicas criadas para substituir as réplicas perdidas consequência de falhas.

Diferentemente do CFS, nosso trabalho utiliza codificação de dados em fragmentos redundantes usando o algoritmo de dispersão de informação (IDA) para o armazenamento dos mesmos em nós do sistema, além de utilizar Pastry para o roteamento de mensagens. Nosso caso para tratamento

de falhas realizaremos a implementação de um mecanismo de substituição dos fragmentos perdidos. Para melhorar o mecanismo estabeleceremos uma proposta de replicação fragmentos no mesmo aglomerado.

PAST [DR01], diferentemente do CFS, provê armazenamento de arquivos por inteiros. Ao inserir um novo arquivo, o usuário indica um número de k réplicas que devem ser criadas. Uma cópia do arquivo por inteiro é armazenada nos k nós cujos identificadores de nós estão mais próximos ao identificador do arquivo. A replicação tem o efeito de balancear a carga das requisições pelo arquivo e reduzir latências de acesso devido a como Pastry roteia as mensagens, sempre dando prioridade aos nós que estejam mais próximos na rede física que os interconecta. Devido à diferença nos tamanhos dos arquivos a ser armazenados é necessário balancear o espaço de armazenamento ainda disponível entre o número de nós vizinhos.

Diferentemente de PAST, no OppStore os arquivos são armazenados como fragmentos redundantes, além de considerar situações de indisponibilidade dos recursos de armazenamento e se for o caso iniciar o processo de substituição de fragmentos perdidos que estão armazenados nos nós do sistema. Consideraremos numa primeira avaliação o armazenamento de um fragmento adicional, toda vez que vai se armazenar um fragmento num aglomerado uma cópia dele será armazenado no mesmo aglomerado. Numa segunda tentativa estabeleceremos quantidades de fragmentos dependendo do tipo de falha apresentada que servirão para indicar o inicio do processo de substituição de fragmentos perdidos.

OceanStore [KBC+00] replica e armazena os fragmentos de dados sob diferentes servidores do sistema para incrementar a disponibilidade quando falhas acontecem nas infra-estruturas de armazenamento, neste caso por armazenar várias réplicas de fragmentos o uso de algum mecanismo para tratamento de fragmentos perdidos não é usado. Em nosso caso, devemos estabelecer um mecanismo para iniciar a substituição dos fragmentos perdidos. Além disso, avaliamos a tentativa de armazenar uma cópia adicional de cada fragmento em uma outra máquina do mesmo aglomerado toda vez que se armazena algum fragmento.

Neutralizer [YDL09], é um detector de falhas auto-configurável. Estabelece um mecanismo para manter um equilíbrio entre réplicas criadas e réplicas perdidas, para isso estabelece um valor objetivo de réplicas que devem ser mantidas e detectores baseados em tempos para diferenciar as falhas permanentes de falhas transientes. Diferenciando os tipos de falhas pode minimizar a criação de novas réplicas e os custos que implica.

Assim como no Neutralizer o processo de recuperação dos nós que podem ficar perdidos por um tempo, mas voltam depois no sistema pode ocasionar que num determinado momento muitos fragmentos para o mesmo arquivo se encontrem armazenados no sistema. Essa situação deve ser

levada em conta para otimizar o espaço de armazenamento para minimizar a quantidade de réplicas que devem ser criadas. No nosso trabalho estabeleceremos como controlar esse número de fragmentos que poderiam incrementar os custos de armazenamento. Para isso podemos tentar utilizar um detector da quantidade de fragmentos disponíveis e liberar o excesso de fragmentos se for o caso.

JUXMEM [ABJ05], propõe uma arquitetura hierárquica para serviços de armazenamento, atualização e compartilhamento de dados em grade, baseados em sistemas peer-to-peer. Assim, provê acesso a dados mutáveis em ambientes voláteis. Utiliza a especificação JXTA, uma plataforma livre criada pela Sun Microsystems em 2001, numa comunicação entre dispositivos sem considerar sua localização física e tecnologia de rede no qual se encontram instalados.

A arquitetura do JUXMEM, consiste em uma federação distribuída de aglomerados, com o objetivo de prover serviços para o compartilhamento de dados em grades. Cada aglomerado, está composto por um conjunto de máquinas agrupadas numa rede peer-to-peer. Em cada aglomerado as máquinas são classificadas como gerenciadoras ou provedoras.

JUXMEM disponibiliza uma única máquina para ser responsável pelas máquinas do aglomerado, classificada como máquina gerenciadora. As outras máquinas funcionam como máquinas provedoras, que compartilham memoria e espaço em disco para o armazenamento de dados.

O armazenamento dos dados é independente do cliente que os disponibiliza, onde cada bloco de dados armazenado é replicado sobre um número fixo de máquinas provedoras que podem estar organizadas em grupos de dados, de acordo com os dados replicados que elas mantêm.

O JUXMEM, provê transparência na localização de dados e replicação como forma de prover um tratamento de falhas apresentadas nas máquinas do sistema.

8.1 Resumo

Neste capítulo foram citados alguns dos trabalhos relacionados com o armazenamento distribuído de dados, apresentamos uma descrição, assim como as características deles. Nesses trabalhos foram usadas estratégias de replicação de dados armazenados para prover tolerância a falhas quando acontecem perda das máquinas que armazenam os dados, tirando vantagens das características das redes peer-to-peer. No capítulo seguinte apresentamos as conclusões deste trabalho.

Capítulo 9

Conclusões

Para lidar com o problema do gerenciamento de grandes quantidades de dados é importante que os recursos de armazenamento sejam auto-gerenciáveis e que o sistema possa tolerar uma possível grande quantidade de falhas que podem acontecer com esses recursos. Sistemas peer-to-peer oferecem essas características ao permitir que os recursos sejam agregados com facilidade mantendo a qualidade dos serviços mesmo na presença de falhas no sistema.

OppStore aproveita as características das redes peer-to-peer para manter a disponibilidade dos dados que ele pode armazenar e gerenciar, dado que as redes peer-to-peer se adaptam bem com populações variáveis de nós que a formam.

O principal objetivo para este trabalho foi desenvolver mecanismos que permitam manter a disponibilidade dos arquivos armazenados no middleware OppStore. Estes mecanismos permitem a recuperação automática dos fragmentos de arquivos armazenados que são perdidos devido à saída imprevista de nós do sistema e permitem que estes fragmentos estejam disponíveis.

Propusemos dois mecanismos, o primeiro precisa da recuperação do arquivo original para logo após recuperá-lo obter os fragmentos perdidos. O segundo mecanismo implica o armazenamento de uma cópia extra do fragmento no mesmo aglomerado quando inicia o processo de armazenamento de arquivos.

Os experimentos realizados indicam que os níveis de redundância usados para a codificação dos arquivos guardam uma relação direta com o número de vezes que é iniciado o primeiro mecanismo proposto. Isso porque se tem um baixo nível de redundância no armazenamento de arquivos e um grande número de ADRs saindo do sistema. O número de vezes que será iniciado o mecanismo é maior do que o número de vezes que o mecanismo inicia sua execução se é aplicado um maior nível de redundância no armazenamento dos arquivos. Isso foi mostrado na seção 7.2.1

Verificamos também que quanto maior o valor do limiar, maior será o número de vezes que o

primeiro mecanismo será iniciado. Isso porque estamos antecipando as possíveis indisponibilidades de fragmentos armazenados em diferentes ADRs de acordo com o grande número de falhas.

Quando temos uma grade composta por máquinas que se tornam indisponíveis com frequência ou quando máquins se juntam ou saem da grade com frequência, é recomendável usar um maior nível de redundância para o armazenamento dos arquivos, uma vez que assim aumentamos sua disponibilidade.

Com a execução do segundo mecanismo para o tratamento das falhas com os dois diferentes níveis de redundância apresentados conseguimos manter um 100% de FFIs com um número de fragmentos superior a 5 fragmentos. Isso porque ante a presença de uma falha no sistema, o segundo mecanismo substitui imediatamente o fragmento perdido armazenando um novo fragmento baseado na cópia armazenada no mesmo aglomerado.

Para permitir uma alta disponibilidade, o segundo mecanismo requer um tratamento especial no momento do armazenamento dos fragmentos. Ele requer armazenar uma cópia extra no mesmo aglomerado, isso implica um incremento do 100% do espaço de armazenamento e incrementar ao duplo o tráfego intra-aglomerado requerido para o armazenamento de fragmentos e de uma cópia extra. O tráfego extra gerado pelo segundo mecanismo para o armazenamento das cópias dos fragmentos utiliza a rede interna dos aglomerado, ao invés de gerar tráfego na rede que conecta os diferentes aglomerados.

9.1 Trabalhos futuros

Quando consideramos a avaliação dos mecanismos, consideramos a simulação das operações de armazenamento e recuperação de dados na grade. Seria muito importante a implementação completa do mecanismo, incluindo a reconstrução dos fragmentos, e não apenas a simulação.

Seria muito importante fazer a realização de experimentos reais. Colocar o sistema para rodar em uma grade real por um período longo para verificar o seu funcionamento, uso de recursos e buscar por novas otimizações. Outros tipos de simulações também ajudariam a entender melhor o comportamento do sistema.

Poderíamos melhorar o desempenho do primeiro mecanismo quando consideramos saídas temporárias das máquinas quando é feito uma análise do padrão de saída de máquinas da grade no tempo, e assim, otimizar a chamada do inicio do mecanismo.

Percebemos que poderiam acontecer saídas temporárias de máquinas da grade. Passado um

tempo sem estabelecer comunicação com o CDRM responsável pelo aglomerado, os fragmentos nelas armazenados são considerados perdidos. Contudo, essas máquinas que apresentam saídas temporárias podem voltar ao sistema trazendo como consequência o incremento do número de fragmentos armazenados para determinados arquivos. Seria interessante fazer o tratamento do excesso de fragmentos que se poderiam ter os por arquivos armazenados.

9.2 Considerações Finais

Ao realizar o desenvolvimento, implementação e simulação dos mecanismos, vimos que é importante realizar uma grande quantidade de experimentos para avaliar seu desempenho em diferentes situações e com diferentes parâmetros.

Seria de grande importância realizar experimentos em grades reais. Mas para tal, teríamos que realizar a implementação completa dos mecanismos. Além disso, experimentos são demorados, podemos se estender por meses, de modo que os experimentos ficaram fora do escopo deste trabalho.

De fato, o objetivo foi contribuir com as operações de armazenamento e recuperação de dados no OppStore, implementando mecanismos para o tratamento de falhas. Poderiam-se adotar novas idéias para otimizar esses mecanismos, assim como comparar seu desempenho com outras abordagens adotadas quando é realizado o armazenamento distribuído de dados em grades computacionais. Estes pontos são trabalhos futuros a serem desenvolvidos.

Referências Bibliográficas

[ABJ05] G. Antoniu, L. Bougé, e M. Jan. JuxMem: An adaptive supportive platform for data sharing on the grid. *Scalable Computing: Practice and Experience*, 6(33):45–55, 2005. 75

[ACGC05] N. Andrade, L. Costa, G. Germoglio, e W. Cirne. Peer-to-peer grid computing with the OurGrid Community. Em *Proceedings of the SBRC*, páginas 1–8. Citeseer, 2005. 33

[ATS04] S. Androutsellis-Theotokis e D. Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, 2004. 2, 5, 6, 7, 8

[Avi67] A. Avižienis. Design of fault-tolerant computers. Em *Proceedings of the November 14-16, 1967, fall joint computer conference*, páginas 733–743. ACM New York, NY, USA, 1967. 39

[Blo] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. 20

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, e T.W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, páginas 46–66, 2001. 7, 8, 15

[dC08] R.Y. de Camargo. Armazenamento distribuido de dados e checkpointing de aplicaçoes paralelas em grades oportunistas. *SBC*, página 17, 2008. xi, 2, 5, 23, 27, 28, 31, 32, 47

[DKK+01] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, e I. Stoica. Wide-area cooperative storage with CFS. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, 2001. xi, 2, 15, 16, 18, 73

[dpF] Sitio do projeto FreePastry. http://www.freepastry.org/. Último acesso em: janeiro 2010. 47

[dpG] Sitio do projeto Gnutella. http://www.gnutella.com. Último acesso em: junho 2008. 7, 8, 15

[dpK] Sitio do projeto Kazza. http://www.kazaa.com. Último acesso em: junho 2008. 7

[dpN] Sitio do projeto Napster. http://www.napster.com. Último acesso em: maio 2008. 7, 15

[DR01] P. Druschel e A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. Em *Proc. HotOS VIII*, 2001. 2, 15, 18, 74

[GKG+] A. Goldchleger, F. Kon, A. Goldman, M. Finger, e G.C. Bezerra. InteGrade: object-oriented Grid middleware leveraging the idle computing power of desktop machines. xi, 27, 33, 34

[JJ05] H. Jiang e S. Jin. Exploiting dynamic querying like flooding techniques in unstructured peer-to-peer networks. Em *13th IEEE International Conference on Network Protocols, 2005. ICNP 2005*, página 10, 2005. 7

[KBC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, et al. OceanStore: an architecture for global-scale persistent storage. *ACM SIGARCH Computer Architecture News*, 28(5):190–201, 2000. 2, 15, 19, 74

[KCD+00] F. Kon, R.H. Campbell, M. Dennis, M.K. Nahrstedt, e F.J. Ballesteros. 2K: A distributed operating system for dynamic heterogeneous environments. Em *in 9th IEEE International Symposium on High Performance Distributed Computing*, 2000. 36

[LCP+05] K. Lua, J. Crowcroft, M. Pias, R. Sharma, e S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, páginas 72–93, 2005. 2, 5, 6, 7, 8

[MM02] P. Maymounkov e D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. Em *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS 02)*, volume 258, página 263. Springer, 2002. 9

[MNR02] D. Malkhi, M. Naor, e D. Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. Em *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, páginas 183–192. ACM New York, NY, USA, 2002. 9

[PRR99] CG Plaxton, R. Rajaraman, e AW Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32(3):241–280, 1999. 20

[Rab89a] M.O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348, 1989. 23, 40, 49

[Rab89b] M.O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348, 1989. 29, 31

[RD01] A. Rowstron e P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Em *18th IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, 2001. xi, 5, 9, 10, 12, 18, 28

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, e S. Schenker. A scalable content-addressable network. Em *Proceedings of the 2001 SIGCOMM conference*, volume 31, páginas 161–172. ACM New York, NY, USA, 2001. 9, 10

[SMK+01] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, e H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Em *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, páginas 149–160. ACM New York, NY, USA, 2001. 9, 16

[TTL03] D. Thain, T. Tannenbaum, e M. Livny. Condor and the Grid. *Grid Computing: Making the Global Infrastructure a Reality*, páginas 299–335, 2003. 33

[VBR06] S. Venugopal, R. Buyya, e K. Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Computing Surveys (CSUR)*, 38(1), 2006. 1, 24

[VI97] S. Vinoski e I.T. Inc. CORBA: integrating diverse applications within distributedheterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997. 33

[WK02] H. Weatherspoon e J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. Em *Proc. of IPTPS*, volume 2. Springer, 2002. 24

[YDL09] Z. Yang, Y. Dai, e X. Li. The Neutralizer: a self-configurable failure detector for minimizing distributed storage maintenance cost. *Concurrency and Computation: Practice and Experience*, 21(2), 2009. 20, 74

[ZKJ01] B.Y. Zhao, J. Kubiatowicz, e A.D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. *Computer*, 74, 2001. 9