Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Aplicação de RAID em Sistema de Arquivos Distribuídos

Diogo Assis Ferreira
Getúlio Yassuyuki Matayoshi

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Eduardo A. P. Alchieri

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Rodrigo Bonifacio de Almeida

Banca examinadora composta por:

Prof. Dr. Eduardo A. P. Alchieri (Orientador) — CIC/UnB
Prof. Dr. Edson Ishikawa — CIC/UnB
Prof. Ms. Marcos Fagundes Caetano — CIC/UnB

CIP — Catalogação Internacional na Publicação

Ferreira, Diogo Assis.

Aplicação de RAID em Sistema de Arquivos Distribuídos / Diogo Assis Ferreira, Getúlio Yassuyuki Matayoshi. Brasília : UnB, 2016.
147 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. RAID, 2. BFT SMaRt, 3. Sistema de arquivo distribuído

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Aplicação de RAID em Sistema de Arquivos Distribuídos

Diogo Assis Ferreira
Getúlio Yassuyuki Matayoshi

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Eduardo A. P. Alchieri (Orientador)
CIC/UnB

Prof. Dr. Edson Ishikawa
CIC/UnB

Prof. Ms. Marcos Fagundes Caetano
CIC/UnB

Prof. Dr. Rodrigo Bonifacio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 24 de Junho de 2016

Agradecimentos

Foi uma tarefa árdua chegar ao momento atual com a monografia completa, os créditos conquistados e a colação marcada. Digo que a missão foi complicada pois exigiu dezenas de horas de estudo, noites em claro, programas que não compilavam, *segmentation fault* aparentemente inexplicáveis, cálculos que não batiam e diversos finais de semana na faculdade. Porém nenhuma dessas frustrações sobrepõe a satisfação de conseguir superar tais problemas, a conta realizada com precisão, o programa otimizado que executa com exatidão e a prova com sabor de vitória, tudo levando àquela bela e merecida noite de sono antes do ciclo recomeçar. Não foi fácil, mas foi enriquecedor, o quanto aprendi e o quanto vivenciei. São experiências preciosas que carregarei com carinho e orgulho. Entretanto, devo ressaltar que essa trajetória não foi percorrida sozinha. Em todo o momento eu recebia o apoio de pessoas importantes, direta ou indiretamente. Agradeço à minha família, minha namorada e meus amigos por todo o suporte, compreensão e carinho que recebi durante todo o trajeto, sou muito grato a todos. Contudo, gostaria de expressar minha gratidão em especifico ao meu orientador, que teve muita paciência, boa vontade e bom humor para me ajudar nessa parte final do curso e a meu parceiro, Getúlio, que foi de fundamental importância para a conclusão deste trabalho.

Diogo Assis Ferreira

Agradeço especialmente ao meu orientador prof. Alchieri, por ter me dado bastante apoio durante o desenvolvimento deste trabalho, esclarecendo todas as dúvidas e dando várias dicas úteis para superar as dificuldades que enfrentei neste último desafio do curso. Também ao meu parceiro Diogo, que foi grande ajuda para escrever esta monografia, cobrindo as partes que sentia dificuldades.

Getúlio Yassuyuki Matayoshi

Resumo

Este trabalho tem como foco a construção de um sistema de arquivos distribuídos que é tolerante a falhas, sem prejuízo de desempenho e que crie menos *overhead* do que o sistema apresentado pelo método de replicação de dados tradicional. A fim de alcançar tal objetivo foram utilizadas as características inerentes ao conceito do RAID 0, RAID 1 e RAID 5, as quais possibilitaram o teste das latências e o teste do *throughput* das operações de leitura e escrita com arquivos de quatro tamanhos distintos, sendo esses 1KB, 100KB, 1MB e 10MB. A coleta e a análise dos dados resultou na conclusão de que o sistema alcança o objetivo proposto, sendo que, quando executado sob as parametrizações do RAID 5, o desempenho geral é superior se comparado aos outros dois modelos de RAID implementados.

Palavras-chave: RAID, BFT SMaRt, Sistema de arquivo distribuído

Abstract

The focus of this work is the construction of a fault tolerant distributed file system, without compromising performance, that have a minor overhead than the system presented by the traditional data replication method. In order to achieve this goal, inherent characteristics of the concept of the RAID 0, RAID 1 and RAID 5 were used, enabling the tests of latencies and throughput of the reading and writing operations with four distinct file sizes, these being 1KB, 100KB, 1MB and 10MB. The gathering and analysis of these data resulted in the conclusion that the system reaches the proposed objective, being that, when run under the parameterizations of RAID 5, the overall perfomance is superior if compared to the other two implemented RAID models.

Keywords: RAID, BFT SMaRt, Distributed File System

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este capítulo sintetiza uma pequena motivação do porquê um sistema de arquivos distribuído tolerantes a falhas é de fundamental importância para que um sistema distribuído possa ser considerado robusto e confiável, além de apresentar motivos para a relevância de um sistema distribuído como um todo. Utilizando a contextualização como base, ainda é abordado o objetivo geral e os específicos que desejamos alcançar neste trabalho. Por fim, um breve sumário sobre a disposição dos próximos capítulos deste trabalho é apresentado.

Tradicionalmente a ciência era dividida em três paradigmas: empírica, lógica e computacional, porém o cientista computacional Jim Gray idealizou um quarto chamado de *data-intensive science* [11]. Tal paradigma caracteriza-se pelo intenso processamento de *big data* afim de conseguir informações úteis. *Big data* consiste de um conjunto extremamente amplo de dados não estruturados. Justamente pela quantidade massiva de dados que compõe o *big data*, é impraticável utilizar computadores convencionais para seu processamento.

Atualmente a taxa de dados processados e armazenados globalmente aumenta anualmente. Tal crescimento não foi influenciado apenas pelos estudos científicos e grandes empresas, visto que grande parcela da população criou o abito de acessar diariamente as mais diversas modalidades de redes sociais, além de utilizar as facilidades da computação nas nuvens disponibilizada por empresas como o Dropbox ou a Google. Como exemplo da popularização de tais serviços, o *Dropbox* divulgou possuir mais de 300.000.000 usuários cadastrados [3] em 2014, onde cada usuários dispõem um espaço mínimo de 2GB para armazenar seus arquivos remotamente, desta forma, para suprir as necessidades de seus clientes, são necessários no mínimo 600PB de espaço de armazenamento. Sobre o ramo das redes sociais, existe o caso do *Facebook*, considerada em 2012 como a mario em seu ramo de atuação, cujos usuários produzem cerca de 600TB diariamente [2]. Enquanto que na área de pesquisas, A Organização Europeia para a Pesquisa Nuclear (*The European Organization for Nuclear Research - CERN*), o maior laboratório de física de partículas do mundo, informou em 2013 que ao longo de 20 anos produziu 100PB de dados em seus experimentos [1].

Os casos apresentados anteriormente demonstram o imenso volume de dados que são gerados, processados e armazenados diariamente em todo o globo. Nesses casos em que o

volume de dados armazenados chega na ordem de *petabytes* faz-se necessário buscar formas eficientes e seguras para salvar todas as informações. Por exemplo, utilizando-se discos rígidos de 2 TB, serão necessários 50.000 dispositivos para conseguir a capacidade total de 100 PB. Tamanha quantidade de equipamentos seria impossível de serem instaladas em um único servidor, o que acarreta na necessidade de se utilizar um sistema distribuído com vários servidores, ditos como nós, trabalhando conectados através de uma rede. Desta forma, é indispensável a utilização de um sistema de arquivos distribuído (SAD) para o gerenciamento de todos os arquivos espalhados entre todos os servidores. Como exemplos reais de SAD, temos o *Google File System*, desenvolvido e utilizado pela própria *Google* [9], O *Apache Hadoop*, utilizado pelo *Facebook* [5],além do *Amazon S3*, utilizado pelo *Dropbox* [4].

Visto que em um SAD os arquivos estão armazenados distribuidamente entre vários nós, a falha de um nó pode inviabilizar o uso do sistema como um todo. Tal situação ocorre pelo fato de que os arquivos armazenados pelo nó faltoso ficam inacessíveis para o restante dos nós do sistema, deste modo inviabilizando a recuperação de alguns ou até mesmo todos os dados contidos no SAD. Somado a isso, ainda temos o agravante de que a probabilidade de ocorrerem falhas em algum dos equipamentos cresce proporcionalmente com o aumento dos componentes envolvidos, desta forma, o incremento de escala pode resultar na redução da confiabilidade no sistema.

É fundamental encontrar formas de contornar os problemas enfrentados pelos sistemas distribuídos, de modo a torná-lo tolerante à falhas. Para o caso de dados inacessíveis, o método de maior simplicidade é o de replicação. Na replicação os dados contidos em um nó são copiados integralmente para outros nós, assim criando cópias de segurança chamadas réplicas. Deste modo, mesmo quando algum nó apresentar problemas, o sistema ainda será capaz de recuperar qualquer arquivo que estivesse armazenado no nó defeituoso. O grande gargalo dessa abordagem é o desperdício de espaço, visto que cada réplica irá consumir a mesma quantidade de espaço de armazenamento que o arquivo original. Em sistemas que lidam com *big data*, a redundância total de dados muitas vezes é impraticável.

No contexto dos sistemas distribuídos o conceito de paralelismo é muito presente, característica essa que acarreta no aumento de máquinas conectadas pela rede e consequentemente a probabilidade de alguma dessas máquinas sofrer erros de *hardware* ou *software*. Quando qualquer falha dessa natureza ocorrer, é necessário que contra medidas tenham sido implementadas a fim de minimizar os danos. Uma dessas medidas é que o sistema de arquivos distribuído seja tolerante a falhas, tolerância essa que pode ser alcançada utilizando-se a replicação total dos dados, entretanto esse *overhead* ocupa espaços de armazenamento que poderiam ser utilizados para novos dados.

Os conceitos de RAID podem ser utilizados a fim de aumentar o desempenho e a confiabilidade de um sistema. Durante o planejamento da tecnologia RAID vários pontos foram levados em consideração, um deles foi como aumentar a taxa de transferência da unidade de armazenamento. O modo escolhido foi utilizar o paralelismos. Com um conjunto de discos onde cada um deles executa de forma independente e paralela aos outros, é possível realizar acesso simultâneo em vários discos do conjunto, assim aumentando a

vazão total de dados transferidos e resultando no aumento de desempenho do sistema. Enquanto que na área de segurança, exitem distintas formas de proteção dos dados baseadas na redundância, tais como espelhamento, replicação e paridade.

Os pontos fortes da tecnologia RAID podem ser extrapoladas para um sistema de arquivos distribuídos a fim de suprir suas fragilidades sobre proteção de dados e problemas de confiabilidade.

1.1 Objetivo

O objetivo deste trabalho é a construção dum sistema de arquivos distribuídos robusto, eficiente e tolerante a falhas aplicados aos conceitos da tecnologia RAID. Escolhendo e implementando diferentes níveis de RAID a fim de possuir uma melhor base de comparação. Este trabalho também abrange a avaliação de desempenho do sistema desenvolvido, contendo análise critica e qualitativa do desempenho obtido pelo sistema sobre testes exaustivos realizados em uma plataforma com várias máquinas remotas distribuídas.

1.2 Estrutura do trabalho

Os capítulos estão organizados de seguinte forma. O Capítulo 2 apresenta os conceitos sobre sistema de arquivos distribuído, explicando as principais características que esse tipo de sistema apresenta. O Capítulo 3 apresenta os princípios do *BTF-SMaRt*, uma biblioteca que realiza a replicação de serviços para prover tolerância a falhas no sistema, assim como os protocolos centrais e sua implementação. O Capítulo 4 apresenta os conceitos da tecnologia RAID. O Capítulo 5 descreve o desenvolvimento do sistema proposto. O Capítulo 6 explica como os experimentos foram realizados além de analisar os resultados obtidos. Por fim, o sétimo capítulo apresenta a conclusão deste trabalho, além dos trabalhos para futuro.

Capítulo 2

Sistema de arquivos distribuído

Este capítulo aborda as características de um sistema de arquivos distribuído. Primeiramente é explanado brevemente conceitos de um sistema de arquivos, bem como as abstrações e o metadado. A seguir o texto avança apresentando em detalhe as características de um sistema de arquivos distribuído, relacionadas a nosso trabalho.

2.1 Sistema de arquivos

2.1.1 Conceitos

O sistema de arquivos fornece ao sistema operacional os mecanismos para gerenciar as tarefas de criação, manipulação, armazenamento e recuperação de arquivos. Isto ocorre pois trata-se de um sistema fundamental, que possui o papel de organizar, armazenar, recuperar e gerenciar as informações guardadas em mídias de armazenamentos permanentes, como o disco rígido. Por exemplo, os sistemas *FAT*, *NTFS* e *ext* trabalham com dispositivos de disco rígido ou memória *Flash*, enquanto o *ISO 9660* é um sistema de arquivos para disco óptico.

Existem duas abstrações fundamentais no sistema de arquivos, o arquivo e o diretório [10].

Arquivo

Como o próprio nome sugere, o arquivo é a entidade básica da funcionalidade de um sistema de arquivos. Arquivo são conjunto de dados onde o programas e usuários armazenam permanentemente suas informações. As informações contidas nesse conjunto de dados podem ser de vários tipos. Tais como um texto, uma imagem, uma música, dentre outros. Arquivos podem ser compostos por alguns *bytes* ou até mesmo toda a capacidade de um disco rígido.

Diretório

Além de armazenar arquivos, o sistema de arquivos deve fornecer meios de organizar vários arquivos. Diretório ou pasta é o termo usado para representar um contêiner que armazena referências para arquivos e/ou outros diretórios. Deste modo, diretórios podem armazenar arquivos e outras pastas. Esta propriedade estabelece uma construção hierárquica chamada de árvore de diretórios, possuindo como sua raiz um diretório especial nomeado de *root*, ou diretório raiz. Por isso as pastas são tão uteis como mecanismos de organização. O conceito de diretórios está presente na maioria dos sistemas de arquivos atuais, exceto para alguns sistemas incorporados em um sistema embarcado simples.

2.1.2 Metadado

Os sistemas de arquivos, como ambientes propícios para a recuperação de informações, têm na utilização de metadados a padronização das formas de representação e a possibilidade de garantia de interoperabilidade entre sistemas. Favorecendo a integridade e a acessibilidade dos recursos informacionais de forma eficiente pelo usuário final. Os metadados são "dados de dados", ou seja, as informações essenciais sobre os arquivos armazenados, indispensáveis para fazer o gerenciamento dos arquivos, indicando as características inerentes deles. A Tabela 2.1 mostra os atributos tipicamente encontrados em um metadado.

Tabela 2.1: Elementos de metadados

Informação	**Descrição**
Nome do arquivo	O nome do arquivo, incluindo o caminho do diretório.
Proprietário	O identificador do usuário dono do arquivo.
Data de criação	Data de criação do arquivo.
Data de acesso	Data do último acesso.
Data de atualização	Data da última atualização feito sobre o arquivo.
Tamanho	Espaço ocupado pelo arquivo.
Tipo de arquivo	Indica se trata-se de uma pasta ou dum arquivo de dados.
Modo de acesso	Indica a permissão para acessar o arquivo.
Bloqueio	Indica se o arquivo está bloqueado para acesso.

2.1.3 Exemplos de Implementações

São várias as abordagens para implementar um sistema de arquivos. Entre elas, o *inode* (abreviação de *index node*) é uma abordagem utilizada principalmente pelos sistemas de arquivos utilizados nos sistemas *Unix* [14]. O *inode* é uma estrutura de dados capaz de representar arquivos e diretórios. Onde cada *inode* está obrigatoriamente associado a apenas um arquivo ou um diretório. Uma das características marcantes desta abordagem é a sua forma de gerenciar os arquivos. Cada arquivo possui duas partes, uma é a informação gerencial do próprio arquivo, incluindo o metadado, e a outra é o conteúdo

do arquivo, armazenado no formato de vários blocos de dados. Desta forma, um *inode* armazena na sua estrutura o metadado e a localização dos blocos constituintes de um determinado arquivo. No caso do *inode* refente a um diretório, difere-se somente por não possuir referências para blocos de dados.

A característica de separar arquivos em informações gerenciais e conteúdo também está presente no *NTFS*, o sistema de arquivos utilizado por várias distribuições do sistema operacional *Windows* [15]. As informações gerenciais dos arquivos e dos diretórios são organizadas utilizando uma tabela chamada de *Master File Table* (MFT). Cada registro da *MTF* representa um arquivo ou um diretório, armazenando seus atributos. Atributos possuem função análoga ao do metadado, são informações gerenciais sobre arquivos ou diretórios. Por exemplo, o atributo chamado *data stream*, presente somente nos registros de arquivos, armazena a localização dos blocos de conteúdo ou o próprio conteúdo, quando este é pequeno o suficiente para caber dentro do pouco espaço reservado para o atributo. No caso de um diretório, ao invés do registro armazenar a localização dos dados, são armazenadas as referências para os registros dos arquivos e diretórios contidos nele. Desta forma, o *MTF* também utiliza uma estrutura de árvore de diretórios.

2.2 Sistema de arquivos distribuídos

O sistema de arquivos distribuído (SAD) é um tipo de sistema distribuído, que tem como proposta permitir aos usuários de computadores fisicamente distribuídos compartilhem dados e recursos de armazenamento usando um sistema de arquivo comum [13]. Vários SAD são utilizados para resolver diferentes tipos de problemas, portanto as características de um sistema variam dependendo dos requisitos do sistema. Alguns sistemas dão mais importância na taxa de transferência e outros em manter a consistência dos arquivos, por exemplo. Entretanto, em geral, a maioria dos sistemas devem ter as características essenciais para lidar com as seguintes questões a saber [7, 8, 12, 18].

2.2.1 Transparência

A transparência é uma das principais características que os sistemas distribuídos, de modo geral, apresentam. Um sistema distribuído transparente é aquele que se mostra como se fosse apenas um único sistema para os usuários, como ilustrado na Figura 2.1.

O conceito de transparência pode ser aplicado aos diversos aspectos de um sistema distribuído, os mais importantes são mostrado a seguir [18]:

- **transparência de acesso** Não necessita fornecer a localização dos recursos, ou seja, os programas devem executar os processos de leitura/escrita de arquivos remotos da mesma maneira que operam sobre os arquivos locais, sem qualquer modificação no programa. O usuário não deve perceber se o recurso acessado é local ou remoto;

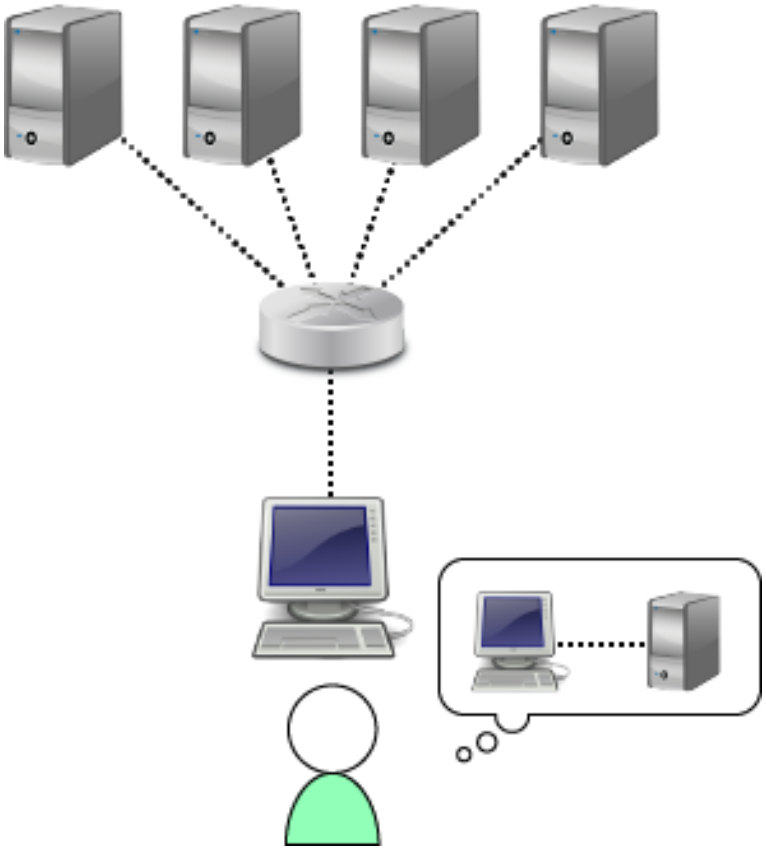

Figura 2.1: Sistema distribuído transparente

- **transparência de localização** Os programas clientes devem ver um espaço de nomes de arquivos uniforme, sem a necessidade de fornecer a localização física dos arquivos para encontrá-los, mesmo que esses arquivos se desloquem entre os servidores;
- **transparência de migração** Independente dos arquivos se moverem entre servidores, os programas clientes não precisam ser alterados para a nova localidade do grupo de arquivos. Essa característica permite flexibilidade em mover arquivos sem comprometer toda a estrutura, ou ter que refazer links entre programas clientes e o local do arquivo;
- **transparência de desempenho** O desempenho da aplicação cliente não poderá ser comprometido enquanto ocorre uma variação dos processos sobre os recursos disponíveis pelos SADs, isto é, mesmo que haja concorrência no acesso pelos arquivos isso não deve afetar os usuários;
- **transparência de escalabilidade** Os recursos computacionais podem sofrer alterações para abrigar maior poder computacional ou o ingresso de novos servidores sem prejudicar o serviço;
- **transparência a falhas** Visa garantir a disponibilidade dos arquivos ininterruptamente, e se ocorrerem falhas o programa cliente não deverá saber como elas serão

tratadas;

- **transparência de replicação** Várias cópias dos mesmos arquivos armazenados em locais diferentes para garantir a disponibilidade. A aplicação cliente deverá visualizar apenas uma cópia do mesmo arquivo, não necessitando saber a quantidade replicada e os locais.

A transparência é altamente desejável em sistemas distribuídos, mas nem sempre é possível alcançá-la ou, em determinadas situações, não convém ocultá-la. Pode-se destacar uma situação que seja mais conveniente o usuário tomar uma decisão sobre alguma falha do que o sistema distribuído tentar resolver por si só. Isso pode ser observado quando um serviço, por repetidas vezes tenta estabelecer uma comunicação com o servidor na *Internet*, neste caso o melhor é informar ao usuário sobre a falha para que ele tente novamente mais tarde.

2.2.2 Disponibilidade

A maioria dos SADs possuem serviços dependentes da rede interna ou externa, a qual ainda continua sendo um meio relativamente instável, podendo até deixar o serviço fora do ar por causa de algum inconveniente que ocorreu na rede. A sua arquitetura, que possui grande quantidade de computadores constituindo o sistema, também é um dos fatores que podem causar a indisponibilidade do serviço. Pois a probabilidade de acontecer alguma falha cresce de acordo com o aumento do número de entidades envolvidas. Assim, um SAD deve implementar um esquema para manter o acesso aos seus recursos em qualquer caso, estando sempre disponível para responder às requisições enviados pelos seus usuários. Além disso, o usuário não precisa saber como tal esquema funciona e nem como foi implementado, podendo continuar a usar o sistema mesmo com a indisponibilidade de alguns servidores. Essa restrição é incorporada para manter a propriedade de transparência do sistema.

2.2.3 Operação atômica

Uma operação é dita atômica quando ela é composta por um conjunto de sub-operações dependentes que devem executar de modo que o restante do sistema as veja como uma só operação. Por exemplo, quando um arquivo é submetido a uma operação atômica, o seu estado muda do inicial para o final sem apresentar qualquer estado intermediário durante a execução [17]. Expandido o conceito, para uma operação ser atômica ela deve satisfazer as seguintes condições:

1. Enquanto as etapas da operação estejam em progressão, nenhum entidade externa consegue perceber o estado intermediário desta operação.
2. Uma operação é efetuada somente se todas as etapas dela são concluídas com sucesso, caso contrário, se tiver uma etapa que falhou, todas as etapas serão abortadas e o estado do sistema volta para antes de executá-la.

Geralmente as operações de leitura, escrita, criação ou remoção de um arquivo apresentam propriedade atômica nos sistemas de arquivos distribuídos.

Transações são mecanismos que permitem realizar uma sequência de operações de forma atômica. Os mecanismos disponibilizam determinados comandos com os quais os usuários podem escolher quais operações serão executadas dentro das transações. Tais comandos são os de início e fim. O comando de início avisa ao sistema que todas as operações a partir daquele ponto estarão dentro da transação, enquanto que o comando de finalização indica que não virá mais nenhuma operação para aquela transação.

Caso alguma operação falhe, o sistema desfaz, ou aborta, todas as alterações que as operações anteriores realizaram. Essa ação é chamada de *rollback* ou *abort*. Contudo, caso todas as operações sejam executadas sem problemas ou erros, ao chegar no fim da transação é realizado um *commit*, ou seja, todas as alterações que foram executadas são efetivadas e persistidas, de tal forma que outros processo possam percebê-las. Deste modo, as transações implementam a semântica do tudo ou nada, ou seja, ou todas as operações são executadas com sucesso, ou nenhuma será executada. Por isso as transações são um importante mecanismo de tolerância a falhas, pois elas evitam que pequenas falhas prejudiquem a integridade de todo o sistema.

2.2.4 Tolerância à falhas e gerenciamento de falhas

O sistema deve ser capaz de continuar operacional mesmo que um ou mais componentes apresentem falhas, isolando esses componentes. Por exemplo, durante a transmissão de dados entre servidores e clientes, podem ocorrer falhas. Seja por excesso de tráfego na rede, ou por algum dos servidores estar sobrecarregado. Além disso, podem ocorrer falhas de *hardware*, inutilizando componentes físicos da rede ou dos servidores.

Esses problemas acontecem em grande parte porque os sistemas distribuídos são implementados sobre redes de computadores que não são totalmente confiáveis. Logo, a grande complexidade de sua implementação está em desenvolver meios de se trabalhar nesse ambiente propício a falhas. Um desses meios é utilizar algum protocolo de comunicação com capacidade para detecção de erros de transmissão [12]. Assim, caso alguma mensagem chegue corrompida no seu destino, o protocolo precisa ser capaz de perceber o problema e retransmiti-la. A retransmissão também deve ocorrer com as mensagens que se perderam no caminho.

Servidores são computadores, e como tal, possuem diversos componentes de *hardware* conectados entre si. Cada um desses componentes também podem apresentar falhas. Por exemplo, um disco rígido pode deixar de funcionar de um momento para outro, seja por excesso de uso ou até mesmo por descargas elétricas. Para contornar esses problemas, é aconselhável o desenvolvimento de soluções desde a redundância física dos equipamentos, realizada via *hardware*, ou redundância controlada pelo próprio sistema distribuído. O qual cuidaria de replicar os dados, para evitar a perda dos mesmos.

Independente do problema, o sistema deve evitar que o cliente fique aguardando uma resposta por muito tempo, além de garantir que seus dados não sejam danificados ou até mesmo perdidos. Isso significa que o serviço precisa ter disponibilidade e confiabilidade. Porém, muitas vezes essas características se conflitam. Por exemplo, uma forma de garantir a confiabilidade é implementar redundância dos dados. Porém a complexidade que isso gera pode aumentar ainda mais a carga do servidor, comprometendo a disponibilidade, pois as respostas aos clientes seriam mais lentas. Outro mecanismo que auxilia a confiabilidade é a transação. Ela evita que o conteúdo de algum arquivo fique em um estado inconsistente caso haja uma queda de algum servidor ou do cliente durante a execução de alguma operação.

Das possíveis falhas, pode-se destacar os seguintes problemas:

- **falha por queda**, problema físico ou lógico no servidor, causando o travamento do sistema operacional;
- **falha por omissão**, significa o não recebimento das mensagens, quer seja por causa de mensagens não aceitas ou por não enviar uma resposta depois da ação concluída;
- **falha de temporização**, ocorre quando uma reposta está fora do intervalo de tempo adequado;
- **falha de resposta** consiste da resposta emitida estar incorreta, ou seja, o retorno não condiz com a solicitação;
- **falha arbitrária**, também conhecida como falha bizantina, ocorre quando um servidor envia mensagens inadequadas, mas que não são consideradas como incorretas. Também pode ser associada a um servidor que está atuando maliciosamente e, portanto, emitindo respostas erradas de forma proposital.

2.2.5 Replicação de arquivos

No contexto de um sistema de arquivos distribuído, existem, basicamente, dois motivos que justificam a utilização da abordagem de replicação. Um deles é a distribuição da carga de acesso entre todos os servidores que constitui o sistema. Se existir um arquivo que é acessado frequentemente, a transmissão constante do arquivo pode causar sobrecarga no servidor que o armazena. O segundo motivo é a proteção dos dados contra falhas, pois um servidor inacessível inviabiliza o acesso aos arquivos que estão armazenados nele.

Logo, se um sistema de arquivos oferecer a funcionalidade de replicação de arquivos, a confiabilidade do serviço é generosamente aumentada. Visto que mesmo com um determinado servidor indisponível, o serviço de arquivos ainda pode continuar operacional por possuir cópias dos dados em outros pontos da rede. Assim, a replicação de arquivos provê tolerância à falhas e distribuição da carga de acesso.

O grande problema nessa abordagem é a sua implementação, pois ela pode ser muito complicada, por ser necessário manter os dados sincronizados e consistentes ao mesmo

tempo. Existem dois tipos de implementações. O primeira utiliza-se da comunicação em grupo, ou seja, sempre que ocorrer alguma alteração em algum de seus arquivos, o servidor afetado dispara uma mensagem para todos os outros servidores, informando-os da alteração. O segundo utiliza dos conceitos de votação e número de versão. Isso significa que quando um cliente solicitar pela permissão para alterar um arquivo, os servidores votarão entre si para selecionar quem possui a versão mais recente, o vencedor será o servidor padrão daquele arquivo, e o número de versão do arquivo será incrementado.

2.2.6 Escalabilidade

Os sistemas distribuídos são, em geral, projetados e configurados tendo em mente a configuração da rede. A aquisição e conexão de centenas de novos nós no sistema podem aumentar o porte da rede. A menos que esse aumento tenha sido pensado no momento do projeto da rede, dificilmente um sistema de arquivos distribuídos apresentará um bom desempenho.

Por outro lado, caso o projeto da rede tenha considerado uma grande quantidade de nós (tanto de clientes quanto de servidores) com o sistema de arquivos distribuído em vários servidores, pode resulta no incremento do custo computacional para localizar onde um determinado arquivo está armazenado fisicamente. Por exemplo, se para abrir um de seus arquivos o cliente tiver que perguntar para cada servidor do sistema se ele é o responsável por aquele arquivo, haverá congestionamento na rede com mensagens desnecessárias. Caso tente resolver tal dificuldade utilizando um servidor central para resolver todos os caminhos para todos os arquivos do sistema, indicando a localização dos mesmos, tal servidor sofrerá sobrecarga e será o gargalo do sistema.

Um sistema escalável é aquele que leva em conta tais problemas e tenta evitar sua ocorrência quando o número de clientes aumenta ou diminuem drasticamente.

2.2.7 Acesso concorrente

Diversos usuários podem acessar vários arquivos, ou os mesmos arquivos, sem sofrer danos, perda de desempenho ou quaisquer outras restrições. Isso tudo deve ocorrer sem que o usuário precise saber como o acesso é realizado pelos servidores. Assim, é necessário haver transparência de concorrência. O maior problema encontrado nas implementações dessa solução é quanto à sincronização dos arquivos, o que inclui leitura e escrita concorrente. A leitura concorrente pode ser implementada facilmente, bloqueando as operações de escrita enquanto o arquivo estiver sendo lido por ao menos um cliente. Para habilitar a escrita concorrente, todos os leitores devem ser avisados quando o arquivo for alterado, e todos os escritores precisam tomar cuidado para não sobrescrever as alterações feitas por outros.

Afim de demonstrar a complexidade desse problema, suponha duas operações bancárias ocorrendo simultaneamente para a mesma conta. Uma delas é um saque de R$100,00 e a outra um depósito de R$1000,00. Antes dessas operações, suponha que a conta possuía

R$200,00 de saldo, e também suponha que esse valor esteja armazenado em um arquivo gerenciado por um sistema de arquivos distribuídos. Quando o cliente da conta for realizar o saque, a aplicação irá armazenar em memória o valor atual do saldo, assim como acontecerá com a aplicação do outro caixa que estará recebendo o depósito. A aplicação, então, irá adicionar ao saldo o valor do depósito, e gravará no arquivo o novo saldo, ou seja, R$1200,00. Porém, a operação de saque irá subtrair do valor armazenado em memória, que para seu contexto é de R$200,00, o valor do saque, e gravará o resultado, o valor R$100,00, no mesmo arquivo, sobrescrevendo o valor lá existente. Dessa forma, o cliente perderia seu depósito.

Para evitar esse tipo de problema, as aplicações concorrentes podem agrupar um conjunto de operações no sistema de arquivos como sendo uma única transação, deixando a cargo do sistema operacional gerenciar a melhor forma de executar. Existem alguns mecanismos para o controle da concorrência. Dentre eles, destaca-se o mecanismo chamado *lock*, por ser amplamente utilizado. Tal sistema de controle de concorrência baseia-se no bloqueio do arquivo que se quer acessar antes de acessá-lo, através de uma chamada de *lock* para o sistema operacional. Caso um segundo processo queira usar o mesmo arquivo, ele tentará realizar o bloqueio, também usando o comando de *lock*, e o sistema operacional o avisará que esse arquivo já encontra-se bloqueado. Cabe ao processo decidir se espera na fila pelo desbloqueio ou se continua seu processamento sem o acesso ao arquivo. O desbloqueio é realizado pelo processo detentor do arquivo, através de uma chamada de *unlock* para o sistema operacional.

Os bloqueios permitem tornar as transações serializáveis, isto é, o resultado da operação de várias transações simultâneas é o mesmo obtido se elas fossem realizadas uma após a outra [12]. Um protocolo capaz de permitir essa serialização, é o protocolo de bloqueio de duas fases. Sendo que na primeira fase ocorreria o bloqueio de todos os arquivos a serem utilizados durante a transação, e na segunda fase ocorreria a liberação conjunta de todos os arquivos, somente após a realização de todas as operações.

2.2.8 Segurança

Os recursos devem ser protegidos contra acessos ilegais, permitindo somente a execução das operações solicitadas de um usuário conhecido. Compartilhar arquivos entre vários ambientes e usuários é uma das vantagens que os sistemas de arquivos distribuídos permitem. Porém, deixar que pessoas indevidas acessem arquivos confidenciais, como, por exemplo, extrato de conta bancária, é um grande problema. Dessa forma, é fundamental adotar mecanismos de segurança, para evitar que pessoas desautorizadas tenham acesso aos dados restritos do sistema.

Sistemas como o *Unix* adotam um método baseado em permissões para controlar o acesso aos seus arquivos. Cada arquivo possui informações sobre quais usuários podem acessá-lo e de qual maneira. Nos sistemas distribuídos que utilizam sistemas operacionais baseados em *Unix*, quando um servidor recebe uma solicitação para enviar dados de um determinado arquivo, também são recebidas informações sobre o usuário que está

tentando realizar o acesso. Essas informações adicionais permitem verificar se o usuário possui permissão para realizar a solicitação, fazendo uma comparação com as informações de permissões do arquivo.

Outro modo de implementar um controle de segurança, é um sistema baseado em capacidades. Consiste de o cliente enviar ao servidor uma prova de que ele possui a capacidade de acessar um determinado arquivo. Na primeira vez que o usuário acessar um arquivo, o próprio usuário deve enviar ao servidor sua identificação, e o servidor, por sua vez, retorna um código comprovando a capacidade do usuário de acessar o arquivo. Nas próximas requisições, o cliente não precisará se identificar, bastando apenas enviar a prova de sua capacidade. É fundamental que as provas de capacidade não sejam fáceis de serem forjadas.

2.2.9 Nomes e localização

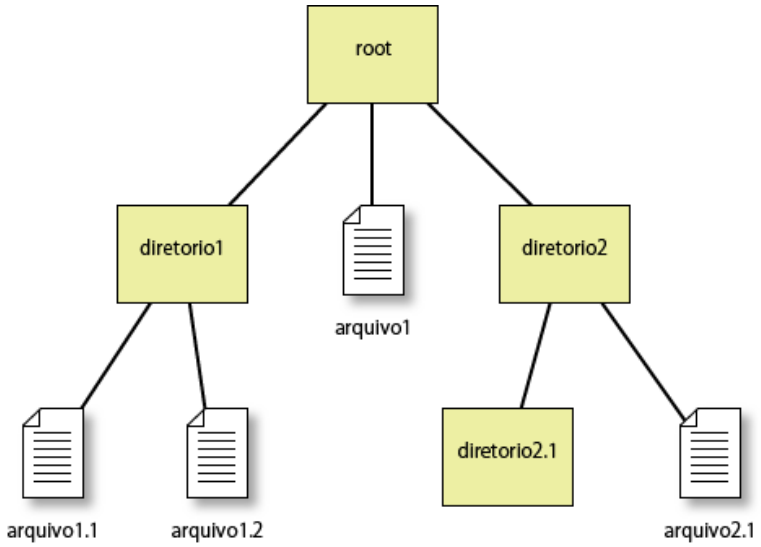

Figura 2.2: Árvore de diretórios

Qualquer arquivo armazenado em um sistema de arquivos possui um nome e um caminho, responsáveis por identificá-lo unicamente por todo o escopo do sistema. Um caminho representa um nó de uma estrutura de diretórios. A estrutura pode ser representada como uma árvore, como mostrado na Figura 2.2. A raiz dessa árvore é costumeiramente referenciada como o diretório raiz, nos sistemas *Windows* é chamada de C:. Dessa forma, para localizar um arquivo em uma árvore de diretórios basta iniciar a busca pelo diretório raiz, e ir percorrendo a árvore até que o arquivo seja encontrado ou chegue ao fim da árvore.

A forma como o nome e o caminho são definidos dependem do sistema operacional. Por exemplo, nos sistemas baseados em *Unix* um caminho é definido como uma sequência de nomes de diretórios, todos delimitados pelo caractere "/". O último nome dessa

sequência é o nome de um arquivo ou de um diretório.

No contexto dos sistemas distribuídos, é desejável adicionar o endereço da máquina em que o arquivo está armazenado no inicio do caminho do arquivo. Porém, deve-se deixar tal modificação transparente para o usuário.

2.3 Conclusões do Capítulo

Este Capítulo trabalhou o conceito de um sistema de arquivos, e o expandiu para abranger as necessidades de um sistema distribuído. Desta forma, apresentando os conceitos e características de um sistema de arquivos distribuído.

Capítulo 3

RAID

Este capítulo está dividido em duas seções nas quais serão apresentados os conceitos de RAID e as peculiaridades de suas principais categorias. A primeira seção do capítulo apresenta os conceitos de RAID e, posteriormente, divide-se em subseções para os níveis RAID 0, RAID 1, RAID 2, RAID 3, RAID 4, RAID 5, RAID 6 e RAID 50. A segunda seção finaliza com conclusões sobre o capítulo

3.1 Conceitos

O desempenho da unidade de processamento e da memória principal cresceu rapidamente. Entre os anos de 1974 e 1984, esse desempenho aumentou cerca de 40% ao ano. Deste modo uma instrução de CPU que realiza uma operação de entrada ou saída para acessar um disco tem a possibilidade de se tornar um grande gargalo, devido ao fato de que o dispositivo de armazenamento magnético, em questão de velocidade de acesso, não acompanha o crescimento dos dois componentes principais, o processador e a memória.

O termo RAID é originalmente a sigla de *redundant array of inexpensive disk*(vetor redundante de discos econômicos), no entanto, atualmente é mais conhecido como *redundant array of independent disk*(vetor redundante de discos independentes). O RAID se baseia no uso de discos extras para aumentar o desempenho do processo de leitura e escrita da unidade de armazenamento, ou para recuperar a informação original em caso de uma falha num disco. Patterson, um dos desenvolvedores da tecnologia RAID, demonstrou que sem um controle de tolerância à falhas grandes vetores de discos econômicos não podem ser considerados úteis devido a sua baixa taxa de confiabilidade [16].

A proposta inicial de Patterson ao desenvolver o RAID era de que, para superar o obstáculo da falta de confiabilidade era necessário o uso de discos extras que possuíssem informação redundante que possibilitassem a recuperação total das informações originais no caso da falha de algum disco. Vale ressaltar que a proposta inicial do RAID foi desenvolvida utilizando o *hardware* como referencia. No entanto, em seu artigo original [16], Patterson frisa que as ideias centrais do projeto poderiam ser aplicadas facilmente na implementação de *software*. Este ponto é de fundamental importância para este trabalho. A ideia central apresentada por Patterson era de quebrar os vetores em grupos confiáveis, onde cada grupo possua discos extras, os quais possuem informação redundante que se-

riam utilizados para manter a confiabilidade dos grupos. Dessa forma, quando um disco falhar, o disco em falha será substituído por um novo disco em um espaço curto de tempo, e a informação que estava contida no antigo disco será totalmente reconstruída no novo disco utilizando as informações redundantes contidas no vetor. O tempo de espera ficou conhecido como *mean time to repair (MTTR)* ou, em uma tradução livre, "tempo de conserto". O MTTR pode ser reduzido se o sistema possuir discos extras que funcionem como peças sobressalentes em estado de prontidão, de forma que, quando um disco falha, ele é trocado por um desses discos extras de forma eletrônica, sem que haja a necessidade de intervenção humana (bastando apenas que, de tempos em tempos, um profissional técnico troque os discos defeituosos por novos discos). Patterson, ao desenvolver o conceito de RAID, assumiu que as falhas de discos são independentes e exponenciais.

O RAID é separado em diferentes níveis, sendo que cada nível possuiu seus objetivos distintos e características intrínsecas. Os principais níveis de RAID serão apresentados e explicados nas próximas subseções deste trabalho.

3.1.1 RAID 0

Também conhecido como fracionamento, pois os dados são fracionados igualmente entre dois ou mais discos, sem a preocupação com informação de paridade, tolerância a falhas ou redundância. Devido ao fato da completa falta de tolerância a falhas no RAID 0, a perda de um disco acarreta na perda (sem qualquer chance de recuperação) de todos os dados contidos no disco e da inutilização de todas as outras frações dos dados que estavam armazenadas nos outros discos do vetor, visto que estes arquivos jamais serão completamente recuperados.

Por esta razão, o RAID 0 é usado apenas nos casos em que deseje aumentar o desempenho do sistema, ou como forma de criar uma grande unidade lógica de armazenamento que utilize dois ou vários discos físicos. Inclusive, é possível modelar um sistema em RAID 0 que utilize discos de diferentes capacidades. Entretanto, é importante frisar que os discos com maior quantidade de armazenamento disponível serão limitados ao espaço máximo do menor disco do vetor. Por exemplo, em um vetor montado com dois discos, um com 500GB e um segundo de 280GB, possuiria o total de 560GB de armazenamento, ou seja, 280 X 2. Nitidamente o disco de 500GB está sendo subutilizado neste vetor.

A Figura 3.1 representa o diagrama de um vetor modelado com o RAID 0 utilizando-se dois discos. Como os dados contidos nos discos 0 e 1 podem ser recuperados simultaneamente, isso resulta na ilusão de que o *drive* é mais rápido.

Desempenho

RAID 0 pode ser usado em áreas onde o desempenho é crucial, considerando que a integridade dos dados seja de pouca relevância, como ocorre no caso de jogos *online*.

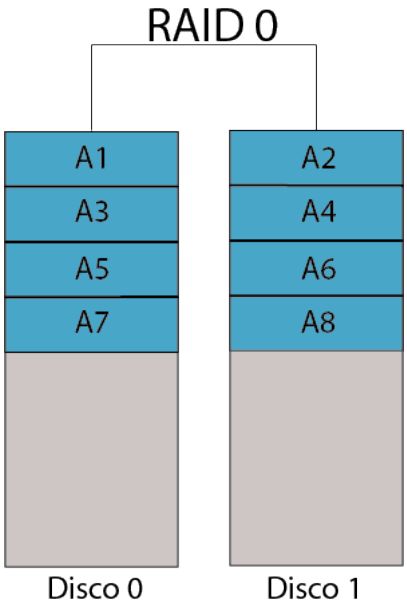

Figura 3.1: Diagrama do RAID 0.

3.1.2 RAID 1

RAID 1 é também conhecido como espelhamento, pois este nível consiste de uma cópia exata de um conjunto de dados espalhado por dois ou mais discos. A configuração do RAID 1 não proporciona paridade ou fracionamento de dados, visto que os dados são apenas espelhados dentro de todos os discos pertencentes do vetor, sendo que o espaço de armazenamento do vetor não pode ser maior do que o disco membro com o menor espaço de armazenamento. O vetor continuará funcional enquanto ao menos um membro do vetor esteja disponível. Esta configuração é útil para sistemas onde a confiança e o desempenho de leitura seja mais importante do que performasse de escrita ou um armazenamento eficiente.

A Figura 3.2 representa o diagrama de um vetor modelado com o RAID 1 utilizando-se dois discos.

Desempenho

Qualquer solicitação pode ser atendida por qualquer disco do vetor. Sendo que no caso de solicitações de escrita o desempenho é comprometido pois é necessário que ela seja replicada em todos os discos membros do vetor.

3.1.3 RAID 2

A configuração RAID 2 fraciona o dado ao nível do *bit* ao invés de blocos, além de utilizar a técnica do Código de Hamming para a correção de erros. O eixo de rotação

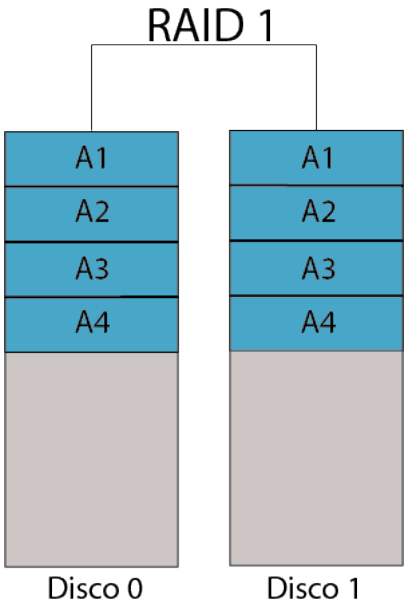

Figura 3.2: Diagrama do RAID 1.

de todos os discos são sincronizados e o dado é fracionado de forma que cada sequência de *bits* sejam armazenadas em discos diferentes. A paridade do código de hamming é calculada sobre os *bits* correspondentes e armazenadas em ao menos um disco de paridade.

A Figura 3.3 representa o diagrama de um vetor modelado com o RAID 2 utilizando-se sete discos, sendo três deles utilizados como discos de paridade.

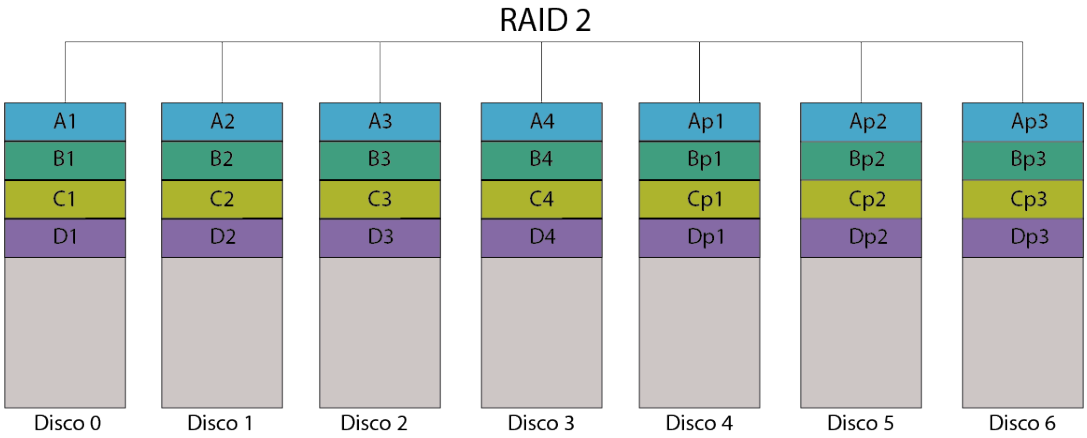

Figura 3.3: Diagrama do RAID 2.

Desempenho

Por utilizar o código de Hamming, possui forte tolerância a falhas. Entretanto, como os discos rígidos atualmente possuem correção interna de erros, o RAID 2 acabou perdendo seu propósito.

3.1.4 RAID 3

Diferente do RAID 2, o RAID 3 fraciona os dados ao nível de *byte*, tendo um disco dedicado de paridade. O eixo de rotação dos discos são sincronizados e o dado é fracionado de forma que cada sequência de *bytes* sejam armazenadas em discos diferentes. A paridade é calculada sobre os *bytes* correspondentes e armazenadas no disco de paridade. Apesar de existir casos de implementação, o RAID 3 dificilmente é utilizado na prática devida a forma de fracionar os dados, onde a fração ao nível de *bytes* é menos eficiente que a em nível de blocos, que será abordada posteriormente.

A Figura 3.4 representa o diagrama de um vetor modelado com o RAID 3 utilizando-se quatro discos, sendo um deles utilizado como disco de paridade.

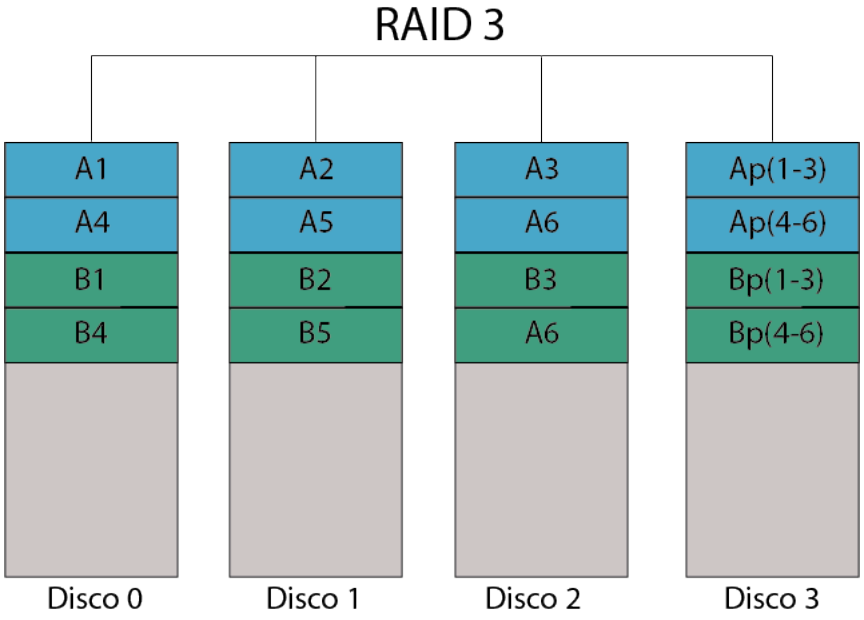

Figura 3.4: Diagrama do RAID 3.

Desempenho

Esta configuração se encaixa para aplicações que necessitam de altas taxas de transferência de grandes sequências de dados sequenciais, como por exemplo, edição de vídeos não comprimidos. Todavia, aplicações que fazem uso de leituras e escritas em localizações aleatórias do disco tendem a ter um desempenho inferior, visto que as frações de um

mesmo dado estão espalhadas nas mesmas seções dos vários discos do vetor.

3.1.5 RAID 4

Diferente dos RAID 2 e 3, o RAID 4 fraciona os dados ao nível do bloco de dados, tendo um disco do vetor dedicado a paridade.

A Figura 3.5 representa o diagrama de um vetor modelado com o RAID 4 utilizando-se quatro discos, sendo um deles utilizado como disco de paridade.

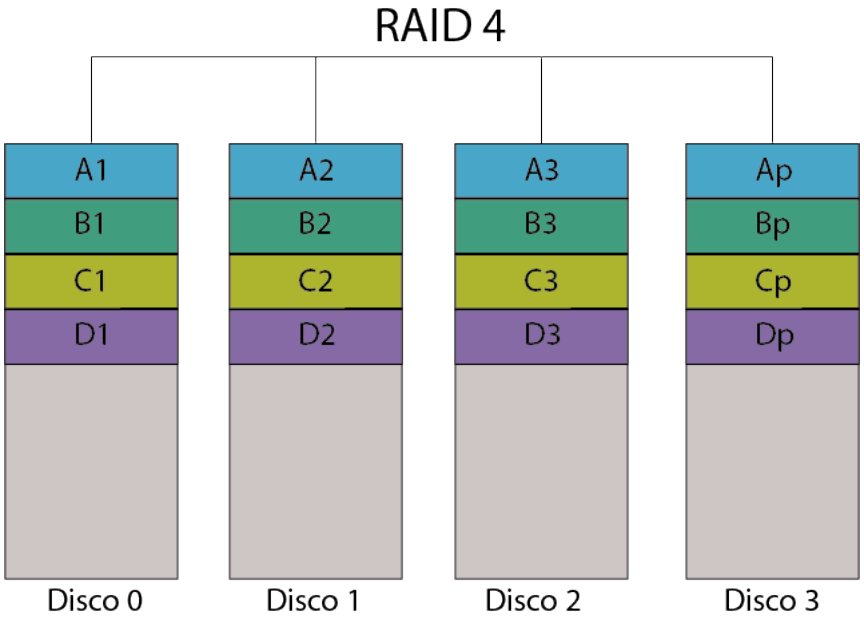

Figura 3.5: Diagrama do RAID 4.

Desempenho

Sempre que os dados são escritos no vetor, o novo dado sobre paridade deve ser recalculado e escrito para o respectivo disco de paridade antes que qualquer requisição de escrita seja realizada. Por causa dessas operações de leitura e escrita, o disco de paridade é o fator limitante do desempenho total do vetor.

3.1.6 RAID 5

Similar ao RAID 4, o RAID 5 fraciona os dados ao nível do bloco de dados, porém tendo as informações de paridade espalhadas entre os discos do vetor. Isto exige que todos os discos (exceto um) tenham os dados. No caso da falha de apenas um disco, leituras subsequentes podem ser calculadas utilizando-se os dados restantes e das informações de

paridade para recuperar o bloco faltoso. Entretanto, caso o disco que está indisponível é o portador dos dados de paridade, não acarreta na necessidade de nenhum cálculo a mais. É importante frisar que por suas características, o RAID 5 necessita de ao menos três discos.

A Figura 3.6 representa o diagrama de um vetor modelado com o RAID 5 utilizando-se quatro discos.

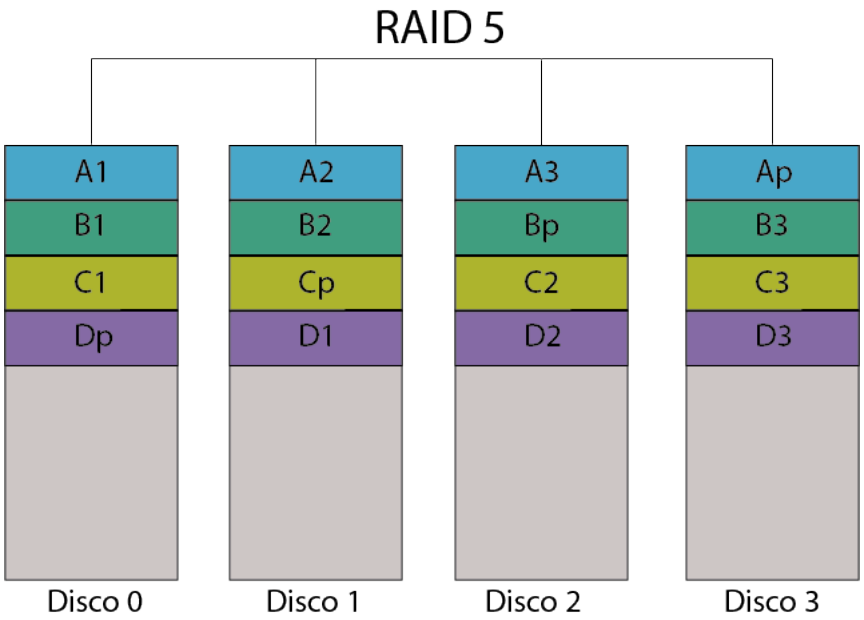

Figura 3.6: Diagrama do RAID 5.

Desempenho

O RAID 5 sofre pelo tempo demasiadamente grande necessário para reconstruir um arquivo a partir dos blocos restantes e do bloco de paridade, além da chance de mais de um disco ficar indisponível durante a fase de recuperação. Como a recuperação de um bloco de dados requer a leitura de todos os blocos de dados, abre-se a chance de perder todos os dados do vetor caso ocorra a perda de um segundo disco.

3.1.7 RAID 6

Similar ao RAID 5, o RAID 6 fraciona os dados em nível do bloco de dados, porém tendo as informações de paridade espalhadas entre os discos do vetor e duplicadas. A novidade da paridade dupla garante tolerância a falhas até o caso da perda de dois discos. RAID 6 exige a utilização de no mínimo quatro discos.

Semelhante ao caso do RAID 5, a queda de um disco acarreta na redução do desempenho de todo o vetor até que o disco defeituoso tenha sido substituído.

A Figura 3.7 representa o diagrama de um vetor modelado com o RAID 6 utilizando-se cinco discos.

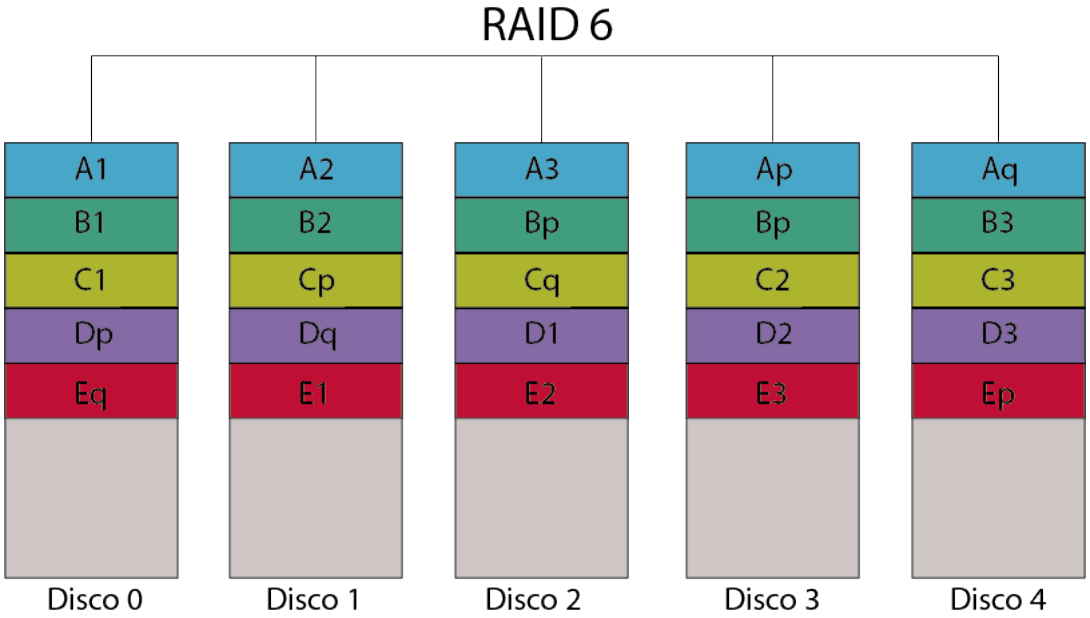

Figura 3.7: Diagrama do RAID 6.

Desempenho

RAID 6 não possui penalidade de desempenho sobre operações de leitura, contudo, possui penalidade de desempenho nas operações de escrita devido ao *overhead* associado aos cálculos de paridade.

3.1.8 RAID 50

Existe o termo originalmente conhecido como RAID híbrido, o qual é definido pela aplicação de um nível de RAID sobre outro vetor formado por outro nível de RAID. O último vetor gerado é conhecido como o vetor do topo. Logo, no caso do RAID 50 (ou RAID 5+0) é uma combinação híbrida que usa as técnicas de RAID 5 com paridade em conjunção com a segmentação de dados do RAID 0. Em outras palavras, o RAID 50 não passa da aplicação do RAID 0 sobre um vetor de RAID 5.

A Figura 3.8 representa o diagrama de um vetor modelado com o RAID 50 utilizando-se três conjuntos de quatro discos cada.

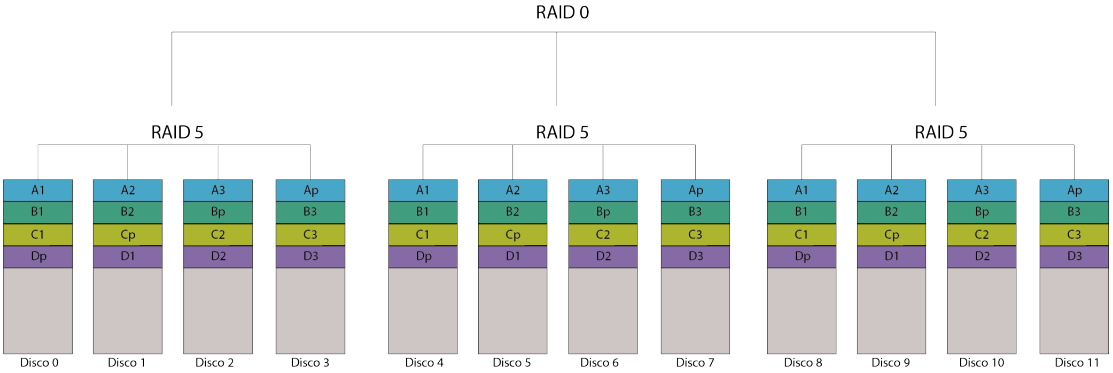

Figura 3.8: Diagrama do RAID 50.

Desempenho

Possui alta taxa de transferência, porém com alto custo de manutenção e expansão.

3.2 Conclusões do Capítulo

Neste capítulo foi apresentado os conceitos básicos do que é RAID, suas características positivas e negativas, além de abordar sobre as topologias mais importantes do RAID, ou seja, RAID 0, RAID 1, RAID 2, RAID 3, RAID 4, RAID 5, RAID 6 e RAID 50. Vale destacar que ainda existem outros níveis de RAID oriundos da combinação destes níveis básicos apresentados.

Capítulo 4

BFT-SMaRt

Este capítulo detalha o *BFT-SMaRt* apresentando seu conceito, princípios de projeto, protocolos centrais, reconfiguração, implementação, configurações alternativas e finalizando nas conclusões do capítulo.

4.1 Conceitos

Na área da computação, a replicação de máquina de estado (SMR) é um método para implementação de um serviço tolerante a falha, onde um conjunto de servidores trabalham coordenadamente para prover determinado serviço para a(s) máquina(s) cliente(s).

O *BFT-SMaRt* é uma biblioteca de código aberto criada recentemente em linguagem Java. É composta por um conjunto de classes capaz de implementar um sistema distribuído robusto consistindo pela replicação das máquinas de estado tolerantes a falhas bizantinas. Falha bizantina ocorre quando uma máquina apresenta um comportamento arbitrário fora de sua especificação, por exemplo, quando um servidor é invadido e começa a rodar código malicioso. Além disso, o *BFT-SMaRt* apresenta confiabilidade, modularidade, reconhecimento de sistema *multicore*, suporte à reconfiguração e interface flexível de programação, como foi apresentado por Alchieri et al em [6].

Novamente em [6] é abordado como nos últimos anos a discussão sobre replicação de máquina de estado (*State Machine Replication - SMR*) tolerantes a falhas bizantinas (*Byzantine Fault-Tolerant - BFT*) tem se acirrado contendo pouco avanço prático, apenas protótipos usados para validar ideias apresentadas em artigos, assim dificultando a aplicação dessa prática em aplicações reais. Os autores acreditam que isto seja devido à falta de implementações de um SMR BFT robusto. O *BFT-SMaRt* foi proposto tendo em mente contornar tal dificuldade, o qual almeja tanto alta performance em execuções livre de falhas quanto corretude mesmo com replicas que apresentam comportamento arbitrário. Incluindo o desenvolvimento de protocolos para transmissão de estados e reconfiguração.

4.2 Princípios do Projeto

4.2.1 Modelo Harmonioso de Falhas

BFT-SMaRt tolera falhas bizantinas não-maliciosas por padrão. Num modelo de sistema realista mensagens podem ser enviadas, rejeitadas ou corrompidas enquanto processos podem executar de forma anormal sem que hajam terceiros envolvidos. Além disso, também é possível configurar o *BFT-SMaRt* para que ele lide com falha bizantinas maliciosas, para tal ele provem assinaturas criptografadas.

4.2.2 Simplicidade

A ênfase na corretude dos protocolos levou aos projetistas evitarem otimizações no código-fonte que poderiam acarretar em complexidade desnecessária tanto em tempo de desenvolvimento ou codificação. Este foi um dos motivos para a biblioteca ter sido desenvolvida em Java ou invés de outras linguagens de programação em alto nível, como C/C++ ou Python.

4.2.3 Modularidade

BFT-SMaRt implementa um protocolo SMR modular que utiliza uma primitiva de consenso bem definida. Além de módulos responsáveis por garantir uma comunicação ponto-à-ponto confiável, ordenação de solicitações de clientes e o consenso entre SMRs, o *BFT-SMaRt* também implementa módulos de transferência de estados e reconfiguração, os quais são totalmente separados do protocolo de *agreement*.

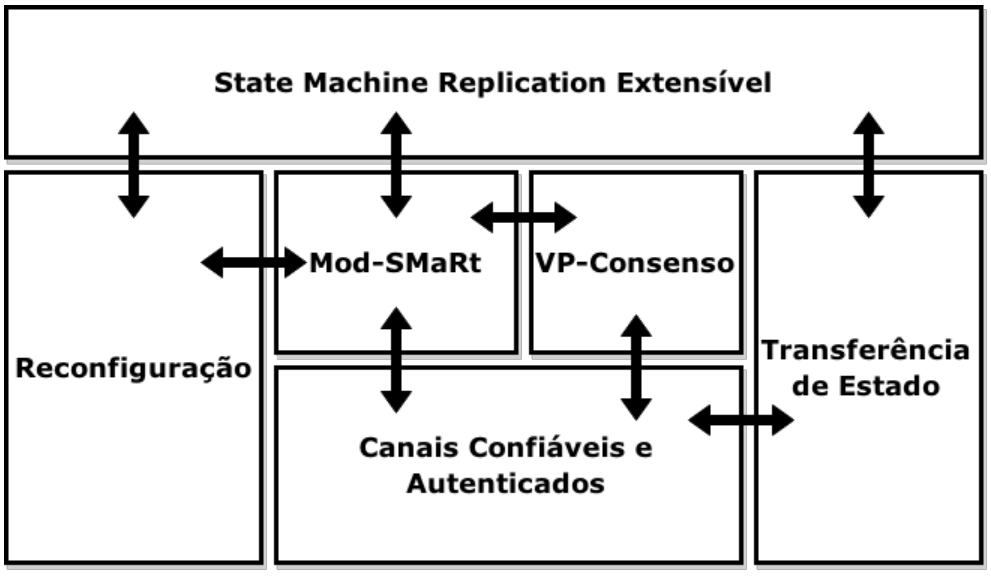

Figura 4.1: A modularidade do BFT-SMaRt. Adaptado de Alchieri [6]

4.2.4 Interface de Programação de Aplicação Simples e Extensível

A biblioteca java encapsula toda a complexidade de uma replicação de máquinas de estado tolerante a falhas bizantinas (BFT SMR) dentro de uma API que pode ser utilizada para a implementação de serviços determinísticos. Utilizando métodos *invoke*(comando) para enviar comandos às réplicas que implementam o método *execute*(comando) cujo objetivo é processar o comando recebido. Entretanto, caso a aplicação necessite de comportamentos especializados não suportado por este modelo de programação é possível utilizar outras chamadas ou *plug-ins* tanto no lado do cliente quanto do servidor.

4.2.5 Percepção de ambiente Multi-Core

BFT-SMaRt é capaz de aproveitar a arquitetura *multicore* dos servidores para diminuir o tempo de processamento em regiões críticas do protocolo.

4.2.6 Modelo do Sistema

BFT-SMaRt assume o modelo usual para sistemas BFT SMR: são necessárias n $\geq$ 3f+1 réplicas para tolerar f falhas bizantinas. Entretanto, visto que o sistema suporta reconfiguração, é possível modificar n e f em tempo de execução. Além disso, o sistema permite ser configurado para utilizar apenas n $\geq$ 2f+1 réplicas para tolerar f falhas de sistema. Porém, independente da configuração, o sistema necessita de conexões ponto-a-ponto confiáveis entre os processos de comunicação. Essa conexão é realizada utilizando *message authentication code* (MAC) sobre o protocolo TCP/IP.

4.3 Protocolos Centrais

4.3.1 *Total Order Multicast*

Em um sistema distribuído, um algoritmo de ordenamento total é um protocolo de mensagem *broadcast* que garante a entrega das mensagens de forma confiável e na mesma ordem para todos os participantes.

O ordenamento total *multicast* é possível graças ao *Mod-SMaRt*, um protocolo modular que implementa BFT SMR utilizando uma primitiva de consenso. Durante sua fase de execução normal, a qual ocorre na ausência de falhas e na presença de sincronismo entre as outras réplicas, clientes enviam suas solicitações para todas as réplicas e aguardam pela resposta. O ordenamento total é alcançado através de uma sequência de instâncias de consenso, cada uma delas decidindo sobre um lote de solicitações de clientes. Cada instância é composta por três passos de comunicação. O primeiro passo solicita que o líder do consenso envie uma mensagem de *PROPOSE* para cada réplica. Esta etapa é seguida por duas etapas de mensagens de todos para todos compostas de mensagens *WRITE* e

ACCEPT. Onde as mensagens de *PROPOSE* contém o lote de solicitações, *WRITE* e *ACCEPT* contém o *hash* criptografado do lote.

Quando uma falha ocorre ou alguma replica encontra-se dessincronizada das demais, o *Mod-SMaRt* pode mudar para a fase de sincronização. Durante esta fase um novo líder é eleito e as réplicas são forçadas a entrarem na mesma instância de consenso. Este "pulo" pode causar com que algumas réplicas ativem o protocolo de transferência de estados.

4.3.2 Transferência de Estado

A fim de implementar uma SMR que possa ser usada na prática, faz-se necessário que as réplicas possam ser reparadas e reintegradas ao sistema sem que todo o sistema de replicação seja reiniciado. Para garantir tal característica, o *BFT-SMaRt* implementa algumas ideias chaves: (1) armazenar o log dos lotes de operações em execução em apenas um disco, (2) tirar instantâneos de estados (*snapshots*) em diferentes pontos da execução em várias réplicas e (3) realizar transferência de estados de forma colaborativa, cada réplica enviando diferentes partes do estado para a réplica que está sendo recuperada.

4.4 Reconfiguração

O *BFT-SMaRt* provê um protocolo especial que permite a adição ou execução de réplicas em tempo de execução. Porém, tal processo só pode ser iniciado pelos administradores executando um cliente com permissão de gerenciamento (*View Manager*), por motivos de segurança.

4.5 Implementação

BFT-SMaRt foi desenvolvido contendo menos de treze mil e quinhentas linhas de código Java distribuídos em cerca de noventa arquivos. Tal característica é significativamente menor do que ocorre em sistemas similares que geralmente possuem mais de vinte mil linhas de código.

Um ponto chave quando se está implementando um mediador (*middleware*) de replicação de alta vazão é como separar as várias tarefas do protocolo em uma arquitetura eficiente e robusta. No caso de uma replicação de máquinas de estado tolerante a falhas bizantinas (BFT SMR) existem dois requisitos adicionais: o sistema precisa lidar com centenas de clientes e resistir a possíveis comportamentos maliciosos tanto por parte dos clientes quantos das outras replicas.

A Figura 4.2 apresenta a arquitetura central com as *threads* usadas para o processamento das mensagens arquitetadas pela implementação do protocolo. Nesta arquitetura, todas as *threads* comunicam através de filas delimitadas. A figura também mostra qual

thread alimenta e consume informações de cada fila.

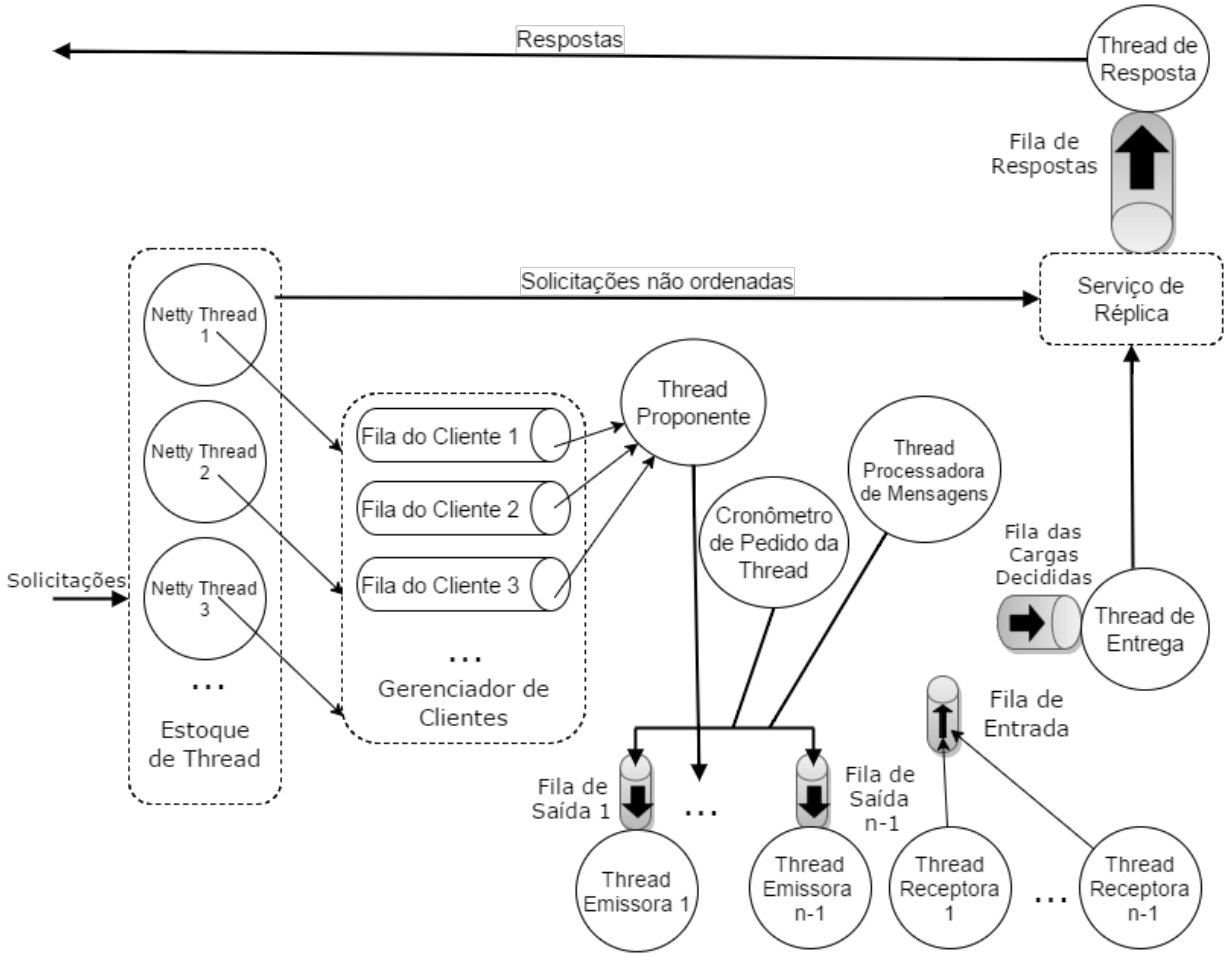

Figura 4.2: Processamento de mensagens arquitetadas entre replicas do *BFT-SMaRt*. Adaptado de Alchieri [6]

As solicitações dos clientes são recebidas através do estoque de *thread* provida pelo *Framework* de comunicação *Netty*. Assim que uma mensagem proveniente do cliente é recebida, é verificado se trata-se de uma solicitação ordenada ou não ordenada. Solicitações não ordenadas, as quais são geralmente aplicadas por comandos de apenas leitura, são entregadas diretamente para o serviço de implementação. No caso de uma solicitação ordenada, elas são entregues para o gerenciador de clientes, o qual verifica a integridade da solicitação, caso esteja integra, a solicitação é encaminhada para a fila do respectivo cliente. Perceba que o endereço MAC dos clientes são verificados pelo *Netty threads*, desta forma as máquinas *multi-core* e multi-processadas vão naturalmente aproveitar de seu poder para conquistar uma alta vazão.

A *thread* proponente é responsável por juntar uma carga de solicitações e transmitir a mensagem *PROPOSE* do protocolo de consenso. O BFT-SMaRt preenche carga com solicitações pendentes até que: (a) seu tamanho alcance o máximo definido no arquivo de

configuração; ou (b) não haja mais solicitações sobrando para serem adicionadas. Esta *thread* só está ativa na replica líder.

Cada mensagem m que deve ser enviada de uma réplica para outra é colocada na fila de saída pela qual uma *thread* emissora vai serializar a mensagem m, produzir o endereço MAC que será anexado na mensagem e, enfim, enviar utilizando *sockets TCP*. Do ponto de vista da réplica que irá receber a mensagem, uma *thread* receptora vai ler m, autenticar (validar seu MAC), desserializar e colocar na fila de entrada, onde todas as mensagens recebidas de outras réplicas são armazenadas em ordem para serem processadas.

A *thread* processadora de mensagens é responsável por processar as mensagens provenientes do protocolo *BFT SMR*. Esta *thread* carrega as mensagens da fila de entrada e as processam caso façam parte do consenso que está sendo executado, entretanto, caso a mensagem pertença a um consenso que ainda será executado, ela é processada posteriormente, quando seu consenso estiver ativo. Caso a mensagem não se encaixe em nenhum dos dois cados, ela é apenas descartada.

Quando um consenso chega ao fim em uma réplica, ele é marcado como decidido e a carga que o possui também é marcada como decidida e colocada na fila das cargas decididas. Então, a *thread* de entrega é chamada para coletar as cargas que estejam armazenados nesta fila, desserializar todas as solicitações da carga, remover cada uma delas das filas de seus respectivos clientes e marcar o consenso corrente como finalizado. Após isso, a *thread* de entrega invoca o serviço de réplica para executar a solicitação e gerar a reposta correspondente. Quando terminar de gerar a resposta, o serviço de réplica a adiciona na fila de resposta. A *thread* de resposta carrega as respostas armazenadas nessa fila e as manda para seus referidos clientes.

O cronômetro de pedido da *thread* é ativado periodicamente afim de verificar se alguma solicitação permanece como não respondida por mais tempo do que o delimitado por um tempo de *timeout* predefinido em alguma fila de solicitações. A primeira vez que este cronômetro expira para alguma solicitação, faz com que ela seja encaminhada para o líder corrente. A segunda vez que este cronômetro expira para a mesma solicitação, a instância atual do protocolo de consenso é paralisado e a fase de sincronização é ativada. A base lógica destes cronômetros é a seguinte: dada uma rede em condições normais, o *timeout* pode ser causado por algum cliente que não enviou a solicitação ao líder ou pelo líder que não encaminhou os pedidos das solicitações dos clientes. Visto que tipicamente existem muitos clientes para poucos servidores, é esperado que ocorra mais falhas no lado dos clientes do que dos servidores, por isto o protocolo do *BFT-SMaRt* assume que o erro ocorreu no lado do cliente, suspeita-se do líder somente se o problema persistir.

4.6 Configurações Alternativas

Como mencionado nas seções anteriores, por padrão, o *BFT-SMaRt* tolera falhas bizantinas não maliciosas. Entretanto, o sistema pode ser configurado para suportar dois

outros modelos de falhas.

4.6.1 Falhas de Sistema

O *BFT-SMaRt* suporta uma configuração de um parâmetro que, caso seja ativado, faz com que o sistema tolere apenas falhas de sistema. Quando esta propriedade está ativa, o sistema tolera f <n/2 (minoria simples), o que implica alterações em todos os passos necessários do protocolo, inclusive ignorando o passo de *WRITE* durante a execução do consenso. Fora essas alterações, os protocolos são os mesmos do caso de tolerância á falha bizantina.

4.6.2 Falhas Bizantinas Maliciosas

Trabalhos anteriores ao *BFT-SMaRt* demonstraram que, o uso de assinaturas com chaves públicas sobre solicitações, torna impossível para os clientes forjarem vetores MAC e forçar que o líder seja alterado. Por padrão, o *BFT-SMaRt* não utiliza assinaturas de chave pública além da utilizada para estabelecer chaves simétricas compartilhadas entre réplicas e durante a mudança do líder. Contudo, o sistema opcionalmente permite o uso de solicitações assinadas para evitar tal problema.

Os mesmos trabalhos também mostram que líderes maliciosos podem lançar ataques de degradação de performance indetectáveis, fazendo com que a vazão do sistema caia drasticamente. Até o presente momento, o *BFT-SMaRt* não apresenta meios de defesa contra este tipo de ataque.

Por fim, o fato do *BFT-SMaRt* ter sido desenvolvido em Java, faz com que seja fácil aplicar o sistema em diferentes plataformas. Tal escolha permitiu que o time de desenvolvimento evitasse falhas de um único nó causadas por eventos acidentais (por exemplo, algum *bug* ou problemas de infraestrutura) ou ataques maliciosos aproveitando de vulnerabilidades comuns.

4.7 Conclusões do Capítulo

Neste capítulo a biblioteca BFT-SMaRt foi detalhada por meio da apresentação de seu conceito, princípios de projeto, protocolos centrais, reconfiguração, implementação e configurações alternativas. Tais conceitos são fundamentais para a construção de um serviço de metadados, possuindo alta confiabilidade e disponibilidade.

Capítulo 5

Proposta de Sistema de Arquivos Distribuído

Este capítulo aborda o modelo de um sistema de arquivos distribuídos tolerante a falhas, o qual é proposto neste trabalho. O Capítulo inicia com a visão geral da arquitetura do sistema apresentando detalhes do serviço de metadados, serviço de armazenamento, servidor e cliente. Em seguida são abordadas as operações realizadas pelo sistema de arquivos distribuídos proposto. Em seguida, é explanada como foi realizada a implementação do sistema de arquivos distribuídos utilizando conceitos de RAID, focando especialmente nos serviços de metadados e de armazenamento, além dos clientes.

5.1 Arquitetura do sistema

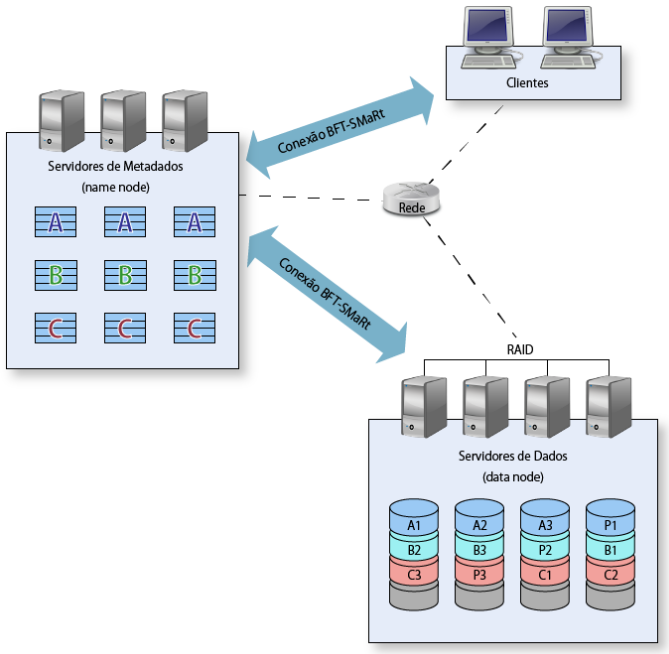

Figura 5.1: Visão geral do sistema

A arquitetura do Sistema de Arquivos Distribuídos proposto neste trabalho é composto por um ou vários clientes comunicando-se com os servidores de metadados e de armazenamento, onde cada um está conectado através da Internet ou outra rede, como a Figura 5.1 sintetiza. Neste sistema, cada servidor de armazenamento comporta-se da mesma forma que um disco rígido em um sistema RAID, armazenando de forma distribuída as partes dos arquivos e/ou as paridades associadas.

Em um sistema de arquivos local, normalmente os dados e metadados referentes ao arquivo são armazenados na mesma unidade de armazenamento. Entretanto, no caso de um sistema de arquivos distribuídos (SAD), os arquivos podem ser armazenados distribuidamente entre servidores distintos, ocasionando a necessidade de inserir metadados em todos os servidores onde os arquivos estejam armazenados. Tal característica dificulta a recuperação de um arquivo em específico, pois pode levar a uma busca completa em todos os servidores do sistema. Essa busca exige um consumo relativamente alto de recursos operacionais, isso por se tratar de uma transmissão em rede.

5.1.1 Serviço de Metadados

O sistema proposto utiliza o conceito de blocos. Cada bloco pode ser um conjunto de dados ou informações de paridade referentes a um único arquivo. O tamanho do bloco é determinado levando em consideração a quantidade de servidores de armazenamento ativos. Por exemplo, se houverem quatro servidores ativos, então o arquivo (independentemente de seu tamanho) será dividido em quatro blocos de tamanhos idênticos e, caso seja necessário, o último bloco será preenchido com *bits* "0"até que o bloco alcance o tamanho dos outros blocos. Deste modo, a quantidade de *bytes* dos blocos é diretamente proporcional ao tamanho original do arquivo.

O Serviço de Metadados é composto por servidores chamados de *name nodes*, os quais são responsáveis por gerenciar os arquivos armazenados no sistema. Para realizar tal atividade, esses servidores utilizam os metadados. No sistema proposto, as datas de criação, última modificação e último acesso, assim como tamanho, identificador do bloco de dados e nome do servidor são considerados metadados.

Também é de responsabilidade do sistema de metadados gerenciar a comunicação entre cliente e servidores de armazenamento (ou *data nodes*). Por meio do uso das informações que indicam o estado atual do sistema (instantâneos de estado) e o nível de RAID que está sendo utilizado, o Serviço de Metadados deve decidir em quantos blocos de tamanho idêntico o arquivo deve ser dividido e em quais servidores de armazenamento cada bloco deve ser armazenado.

O nível de RAID a ser executado deve ser informado no momento em que o Serviço de Metadado for inicializado. O sistema está preparado para tratar os seguintes níveis de RAID, os quais foram explorados com maior profundidade no Capítulo 4.

- RAID 0 - fracionamento simples;

- RAID 1 - espelhamento;
- RAID 5 - fracionamento com paridade espalhada entre os discos do vetor.

O Serviço de Metadados mantém em sua memória local a árvore de diretórios, a qual indica a localização lógica dos arquivos. Quando o Serviço de Metadados recebe uma solicitação proveniente do cliente para recuperar algum arquivo, primeiramente é realizada uma pesquisa na árvore de diretórios do cliente para identificar em qual diretório o arquivo está localizado. Caso o arquivo seja encontrado, são extraídos seus metadados, sua localização física, a lista de todos os servidores que possuem os seus blocos.

Devido à sua propriedade como uma interface entre clientes e o Serviço de Armazenamento, a ocorrência de alguma falha ou indisponibilidade de algum servidor degrada a execução do sistema, podendo até resultar na parada total do mesmo. Assim, é de fundamental importância a elaboração de um esquema para manter o Serviço de Metadados protegido contra falhas ou queda total. No sistema proposto, a biblioteca *BFT-SMaRt* é utilizada, a qual fornece tolerância a falhas no serviço através de replicação por máquina de estado [6].

5.1.2 Serviço de Armazenamento

O Serviço de Armazenamento, também conhecido como *data nodes*, é composto pelo grupo de servidores responsáveis pelo armazenamento físico dos dados gerenciados pelo Serviço de Metadados. No sistema proposto, o Serviço de Armazenamento não possui a necessidade de saber qual nível de RAID está sendo executado, pois a responsabilidade de manter o RAID consistente é do Serviço de Metadados. Desta forma, o Serviço de Armazenamento apenas se conecta a uma porta e aguarda que um cliente se conecte à sua respectiva porta. Quando um cliente se conectar à porta do servidor, é realizado o protocolo *handshake* e, logo em seguida, o cliente inicia a transferência dos dados que devem ser guardados pelo Serviço de Armazenamento. Ao fim da transferência, o cliente fecha a conexão, enquanto o Serviço de Armazenamento indexa o arquivo recebido, utilizando as informações adicionais que o Serviço de Metadados anexou ao arquivo. Com tais informações é possível saber o identificador tanto do arquivo quanto do cliente que o enviou. Tais dados adicionais são necessários para garantir que apenas o cliente tenha acesso aos seu arquivos, além de possibilitar identificar quais dados o cliente deseja receber de forma simples e eficiente.

5.1.3 Cliente

É o programa executado no computador do usuário final do sistema. Para cada operação que deseje realizar, o programa cliente (doravante chamado Cliente) deve primeiramente se comunicar com o Serviço de Metadados. O qual irá realizar a operação (no caso de operações envolvendo diretórios) ou informar com quais servidores de armazenamento o cliente deve se comunicar e quais blocos devem ser enviados para cada servidor (no caso

de operações envolvendo arquivos).

Após a comunicação inicial com o Serviço de Metadados, caso o cliente solicite o envio de arquivos, ele deve iniciar a comunicação com os servidores de dados informados pelo Serviço de Metadados. Caso o Serviço de Metadado esteja utilizando o RAID 5, é o cliente quem fica responsável por calcular a paridade do arquivo. Tal situação ocorre devido ao fato de que apenas o cliente conhece o arquivo. O procedimento para o cálculo da paridade é exemplificado na Figura 5.2. Outra atribuição do cliente é fazer o particionamento do arquivo em blocos de tamanhos iguais, onde cada bloco será enviado para um dos servidores de dados informados pelo Serviço de Metadados. Assim, o arquivo deve ser fragmentado de tal forma que cada servidor receba um bloco de dados. Ressalta-se que, no caso do RAID 5, um dos blocos deve conter apenas informações de paridade previamente calculadas pelo cliente.

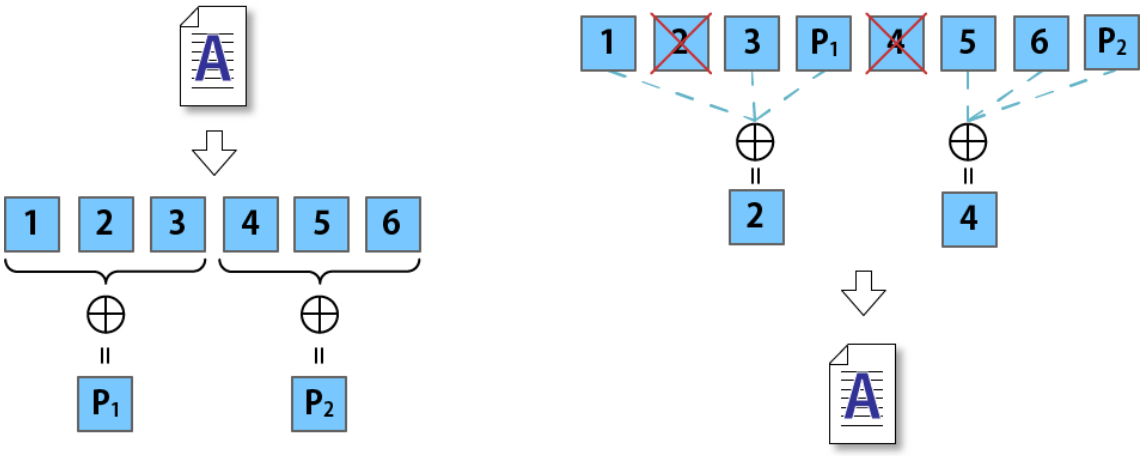

Figura 5.2: Geração de paridade

Caso a operação desejada pelo usuário final seja a de *download*, o cliente deve coletar as informações dos servidores que possuem os blocos do arquivo almejado, assim como as informações de identificação de cada bloco por meio do Serviço de Metadados. Com posse de tais informações o cliente deve iniciar a comunicação com cada um dos servidores de dados, apresentando a identificação de qual bloco cada servidor de dados deve enviar. Quando todos os servidores acabarem seus envios, o cliente deve usar os dados em sua posse para unir todos os blocos e recriar o arquivo desejado pelo usuário. Esta descrição genérica da operação sofrerá leves alterações dependendo de qual opção de RAID o Serviço de Metadados esteja utilizando. No caso do RAID 5, é possível que um dos blocos recebidos pelo cliente seja um bloco de paridade. Nesse caso, o cliente deve utilizar as informações dos outros blocos para determinar qual é o bloco de dados em falta e utilizar a paridade para recuperar tal bloco. A Figura 5.3 mostra um esquema simplificado da operação de recuperação de um arquivo, realizada pelo lado do cliente.

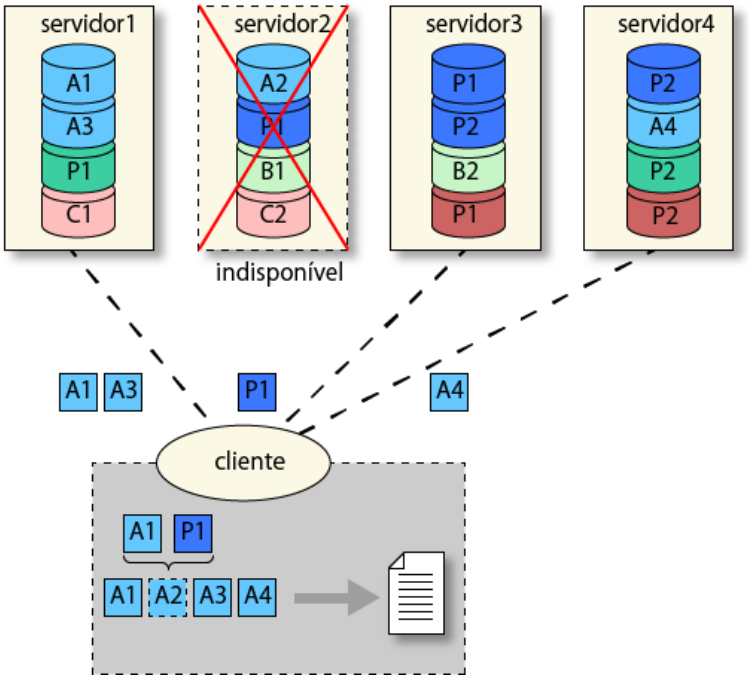

Figura 5.3: Recuperando um arquivo de uma falha

5.2 Operações no Sistema de Arquivos Distribuídos

Nesta seção serão apresentadas as operações básicas que o sistema de arquivos distribuídos (SAD) executa. Tais operações podem ser divididas em dois grupos, dependendo das entidades envolvidas. O primeiro grupo envolve entre cliente e servidor, e é composto por operações envolvendo solicitações de ações do cliente para o servidor. O segundo grupo envolve as operações que ocorrem entre servidores, voltadas para o gerenciamento do serviço, sendo que esses tipos de operações devem ocorrer de forma transparente para o cliente.

5.2.1 Cliente-Servidor

É o conjunto de operações semelhantes as que são implementadas em um sistema de arquivos local, que é aquele integrado na maioria dos sistemas operacionais centralizados.

Criar arquivo

Operação realizada inicialmente entre o cliente e o Serviço de Metadados. Nesse primeiro momento, o cliente informa ao Serviço de Metadados que deseja criar um novo arquivo, e em qual diretório ele deve ser criado, além de seus metadados. Com posse dessas informações e de qual RAID está operando, o Serviço de Metadados processa e transmite para o cliente quais os servidores do Serviço de Armazenamento para quais blocos do arquivo devem ser enviados, assim como a identificação de cada bloco. O pro-

cessamento de tais informações é diretamente influenciado pelo estado atual do sistema. O Serviço de Metadados leva em consideração a disponibilidade de cada servidor do Serviço de Armazenamento e a quantidade de espaço livre em disco que cada um possui.

A segunda parte da operação é realizada entre o cliente e os servidores do Serviço de Armazenamento. Após a comunicação com o Serviço de Metadados, o cliente sabe para quantos servidores o seu arquivo será enviado. Desta forma, o cliente fragmenta o arquivo, de tal maneira que cada servidor receba um bloco de tamanho uniforme. Caso o sistema esteja usando o RAID 5, o cliente ainda deve calcular o bloco de paridade do arquivo e enviar para um dos servidores.

Criar diretório

Operação realizada exclusivamente entre o cliente e o Serviço de Metadados. Nessa operação o cliente informa ao Serviço de Metadados que deseja criar um diretório, passando como metadados apenas o nome do novo diretório e o nome de seu diretório pai. Com tais informações, o Serviço de Metadados primeiramente verifica se já existe um diretório com o mesmo nome no diretório pai informado. Em caso negativo, ele atualiza a árvore de diretório do cliente. No caso de já existir um diretório com mesmo nome, o cliente é informado e o novo diretório não é criado.

Abrir arquivo

Operação realizada inicialmente entre o cliente e o Serviço de Metadados. Nesse primeiro momento o cliente informa ao Serviço de Metadados que deseja abrir um arquivo. Para tal, o cliente deve fornecer o nome do arquivo e o nome do diretório onde ele está salvo. A primeira ação do Serviço de Metadados é verificar se o arquivo de fato existe no diretório informado. Em caso positivo, ele ainda deve verificar se o cliente possui acesso ao arquivo. No caso do cliente possuir acesso, o Serviço de Metadados verifica em quantos blocos o arquivo foi dividido e em quais servidores do Serviço de Armazenamento cada bloco foi armazenado. Após coletar tais informações, o Serviço de Metadados informa ao cliente quais servidores de dados ele deve se comunica, e o identificador de cada bloco. Para o caso de o arquivo não existir no diretório informado, ou o cliente não possuir permissão de acesso para tal arquivo, o Serviço de Metadados deve enviar uma mensagem de erro informando a indisponibilidade do arquivo solicitado.

A segunda parte da operação é realizada entre o cliente e os servidores do Serviço de Armazenamento. Após a comunicação com o Serviço de Metadados, o cliente sabe em quais servidores os blocos de seu arquivo estão armazenados. Desta forma, o cliente inicia a comunicação com cada servidor informado. A comunicação procede da seguinte forma: o cliente informa o identificador do bloco que deseja receber, enquanto que o Serviço de Armazenamento apenas verifica a existência do bloco informado. Em caso positivo, o bloco é enviado para o solicitante. Enquanto que, em caso negativo, uma mensagem de erro alerta a inexistência do bloco solicitado. Repare que o Serviço de Metadados não

realiza nenhuma operação de controle de acesso sobre o bloco.

A terceira parte da operação é realizada exclusivamente pelo cliente. Nesta última parte, o cliente é responsável por unir todos os blocos recebidos e recriar o arquivo almejado. Vale ressaltar que essa operação sofre variações dependo do RAID sendo utilizado. Para o caso do RAID 5, um dos blocos recebido pelo cliente pode ser um bloco de paridade. Nesse caso, antes de recriar o arquivo, o cliente deve utilizar as informações de paridade contidas em tal bloco para recuperar o bloco em falta. Para os outros níveis de RAID, o cliente apenas recria o arquivo utilizando os blocos recebidos.

Abrir diretório

Operação realizada exclusivamente entre o cliente e o Serviço de Metadados. Nessa operação o cliente informa ao Serviço de Metadados que deseja abrir um diretório, passando como parâmetros apenas os nomes do novo diretório almejado e do diretório atual. A primeira ação do Serviço de Metadados é verificar a existência do diretório solicitado. Em caso positivo, o serviço também verifica se o cliente possui permissão de acesso e se o diretório está livre. Caso todas as validações sejam positivas, o servidor muda o diretório atual do cliente para o diretório solicitado. Caso algum problema seja detectado, uma mensagem de erro é disparada para o cliente e a alteração de seu diretório atual não ocorre.

Deletar arquivo

Operação realizada primeiramente entre o cliente e o Serviço de Metadados. Nessa operação, o cliente informa ao Serviço de Metadados que deseja deletar um arquivo. Em seguida, deve informar o nome do arquivo e do diretório onde ele está armazenado. Com tais informações, o Serviço de Metadados percorre a árvore de diretórios em busca do diretório requisitado. Caso seja encontrado, esse serviço inicia nele uma pesquisa pelo arquivo informado. Se o arquivo for encontrado, o Serviço de Metadados verifica se o cliente possui permissão de acesso ao arquivo e se ele não está em estado de bloqueio. Em caso positivo o arquivo é deletado da lista dos metadados e a árvore de diretório do cliente é atualizada. Antes de finalizar a conexão, o Serviço de Metadados informa ao cliente os dados sobre os blocos que compõem o arquivo.

A segunda parte da operação é realizada entre o cliente e Serviço de Armazenamento. O cliente inicia uma conexão com cada servidor de dados, onde um bloco do seu arquivo está armazenado, e o informa que o bloco deve ser deletado.

Deletar diretório

Operação realizada exclusivamente entre o cliente e o Serviço de Metadados. Nessa operação o cliente informa ao Serviço de Metadados que deseja deletar um diretório, passando apenas o nome do diretório almejado. A primeira ação do Serviço de Metadados é verificar a existência do diretório solicitado. Em caso positivo, o Serviço de Metadados

também verifica se este diretório está vazio. Se neste diretório existir algum arquivo ou subdiretório, a operação falha. Caso contrario, o Serviço de Metadados continua com a verificação, se o cliente possuir permissão de acesso e se o diretório estiver livre. Caso todas as validações sejam positivas, o Serviço de Metadados deleta o diretório. Em seguida, o Serviço de Metadados atualiza a árvore de diretórios do cliente.

Fechar um arquivo aberto

Operação realizada exclusivamente entre o cliente e o Serviço de Metadados. Nessa operação, o cliente informa ao Serviço de Metadados que deseja fechar um arquivo. Primeiramente, o Serviço de Metadados verifica se existe algum arquivo aberto pelo cliente. Caso exista, é apresentado ao cliente a lista de arquivos abertos para que ele selecione qual ele deseja fechar. Sabendo qual arquivo deve ser fechado o serviço de metadados libera o acesso ao *lock* do arquivo, desta forma fechando o arquivo. Caso o cliente não tenha um arquivo aberto, é emitida uma mensagem de erro explicativa para o usuário.

Fechar diretório

Operação realizada exclusivamente entre o cliente e o Serviço de Metadados. Nessa operação, o cliente informa ao Serviço de Metadados que deseja fechar um diretório atual. Primeiramente, o Serviço de Metadados verifica se o diretório atual é o diretório raiz. Caso seja, o cliente é informado que não pode fechar o diretório raiz. Caso não seja o diretório raiz, o diretório atual é atualizado para o diretório pai do antigo diretório atual.

Editar um arquivo

Operação realizada inicialmente entre o cliente e o Serviço de Metadados. Nesse primeiro momento, o cliente informa ao Serviço de Metadados que deseja editar um arquivo. Para tal, o cliente deve fornecer o nome do arquivo. Em posse do nome do arquivo, o Serviço de Metadados verifica sua existência no diretório corrente. Em caso positivo, ainda é preciso confirmar se o arquivo já está aberto. Verificado que o arquivo encontra-se fechado, as informações sobre o arquivo são passadas para o cliente. Com tais informações, a operação que o cliente realiza é a exclusão do antigo arquivo, seguida da criação de um novo (ou seja, a atualização do antigo arquivo). Tais operações são realizadas como descrito previamente, entretanto, existe a diferença de que o usuário não precisa informar o nome do arquivo, pois ele é criado com o mesmo nome do antigo.

Renomear um arquivo

Operação realizada entre o cliente e o Serviço de Metadados. Nessa operação, o cliente informa ao Serviço de Metadados que deseja renomear um arquivo. Além de fornecer o nome do arquivo que deve ser renomeado, ele também deve informar o novo nome. Primeiramente, o Serviço de Metadados procura pelo arquivo informado no diretório corrente. Caso o arquivo seja encontrado, ainda é necessário que o Serviço de Metadados

verifique se o cliente possui permissão de acesso ao arquivo e se o mesmo está fechado. Caso o arquivo esteja fechado e o cliente tenha permissão de acesso, o arquivo é, enfim, renomeado com o novo nome.

Renomear Diretório

Operação realizada exclusivamente entre o cliente e o Serviço de Metadados. Nessa operação, o cliente informa ao Serviço de Metadados que deseja renomear o diretório. Além de fornecer o nome do diretório que deve ser renomeado e o seu novo nome. Primeiramente, o Serviço de Metadados percorre a árvore de diretórios em busca do nome fornecido. Caso seja encontrado, o diretório é renomeado com o novo nome fornecido pelo cliente. Por fim, o metadado referente a última modificação do diretório é atualizado.

5.3 Implementação

Nesta seção serão apresentados os detalhes sobre a implementação do sistema, incluindo informações sobre a programação e algoritmos utilizados. Todo o *software* foi desenvolvido na linguagem Java, com auxilio de várias bibliotecas padrões e do *BFT-SMaRt*, o qual foi apresentado com mais detalhes no Capítulo 4.

5.3.1 Árvode de Diretórios

Nesta seção será apresentado o pacote responsável por gerenciar todas as estruturas de diretório do sistema, junto com as classes contidas nele.

- **dt**
  - DirectoryNode;
  - DirectoryTree;
  - LockList;
  - LockType;
  - Metadata.
- **dt.directory**
  - DirEntries;
  - Directory.
- **dt.file**
  - Block;
  - BlockInfo;

- BlockInfoList;
- FileDFS.

As classes contidas no pacote *dt* lidam com todas as características e operações refentes aos diretórios. **LockType** e **LockList** são utilizadas para gerenciar o controle de acesso aos diretórios. A primeira é responsável por enumerar os tipos dos estados de acesso dos diretórios, enquanto a segunda gerência uma lista, na forma de *ArrayList*, dos arquivos ou diretórios que estão abertos pelos usuários. A função desta lista é listar os programas clientes que requisitaram ao Serviço de Metadados a atualização do estado de acesso.

A classe **Metadata** contêm e manipula as informações de metadado dos arquivos. Como é mostrado no trecho de Código 5.1, a classe **Metadata** implementa a interface *Serializable* para que possa transformar o conteúdo de seus campos em sequências de *bytes* para posteriormente serem enviados através da rede.

Código 5.1: Classe Metadata

```
public class Metadata implements Serializable {
   private long creationTimeL;
   private long lastAccessTimeL;
   private long lastModifiedTimeL;

   private long size;

   private int lock;

         ...

}
```

Por fim, **DirectoryNode** e **DirectoryTree** compõem a árvore de diretórios do Serviço de Metadados. A primeira refere-se a cada nó da árvore e a segunda sobre a árvore em si, carregando o diretório raiz e os métodos das operações relacionadas.

Código 5.2: Declaração e os campos da classe Directory

```
public class Directory extends DirectoryNode {
   private HashMap<String, Directory> dirs;
   private HashMap<String, FileDFS> files;

   ...
}
```

Dentro do pacote *dt*, existem dois outros pacotes. O *directory* é responsável por tratar os diretórios em si, composto pelas classes **Directory** e **DirEntries**. A classe *Directory* trata dos diretórios em si, composta por vários métodos de operações sobre pastas além de possuir dois campos *HashMaps*. Um para armazenar as informações das subpastas e outro para as informações dos arquivos contidos no diretório referenciado. Por se tratar de um dos tipos de nó da árvore de diretórios, o **Directory** herda da classe *DirectoryNode*.

O trecho de Código 5.2 exemplifica essa característica.

A classe *DirEntries* é usada pelo Serviço de Metadados para informar aos clientes os registros do diretório aberto pelo usuário. Os registros são apresentados na forma de uma lista composta pelos nomes dos subdiretórios e arquivos contidos no diretório.

O segundo pacote dentro de *dt* é nomeado de *file*, o qual compõem as classes responsáveis pelo gerenciamento das informações sobre arquivos ou dos dados armazenados na forma de blocos. O **FileDFS** é a classe principal deste pacote, sendo encarregando pela parte das informações sobre os arquivos. Apesar disso, o **FileDFS** não possui muitas funcionalidades além de carregar a classe *BlockInfoList*, como exemplificado no trecho de Código 5.3. Note que por ser tratar de um tipo de nó da árvore de diretórios, o **FileDFS** herda as características da classe *DirectoryNode*.

Código 5.3: Classe FileDFS

```
public class FileDFS extends DirectoryNode {
   private BlockInfoList blockList;

   public FileDFS(String name, Directory parent, Metadata metadata, BlockInfoList
       blockList) {
      super(name, parent, metadata);
      this.blockList = blockList;
   }

   public BlockInfoList getBlockList() {
      return blockList;
   }

}
```

Como demonstrado no trecho de Código 5.4, o **BlockInfoList** é a classe que armazenas as propriedades do arquivo: a lista de *BlockInfo*, o tamanho dos blocos, o tipo de RAID utilizado e o número dos servidores que armazenam os blocos. A sua função principal é permitir ao serviço de metadados informar aos clientes os dados operativos de qualquer arquivo. Tal função é necessária quando se precisa enviar ou receber os blocos de dados de algum arquivo. Cada **BlockInfo** gerencia a localização de um bloco, indicando o endereço do servidor onde está armazenado e o identificador do bloco referente.

Código 5.4: Declaração e os campos da classe BlockInfoList

```
public class BlockInfoList implements Serializable {
   private ArrayList<BlockInfo> blocks;
   private long blockSize;
   private int raidType;
   private int nServers;

         ...
}
```

A classe **Block** é encarregada de tratar o conteúdo dos arquivos, gerenciando em formato de blocos. Ela é responsável pela transferência de dados dos arquivos, contidos em blocos, entre cliente e o Serviço de Armazenamento. Além de possuir a atribuição de informar os servidores o identificador de cada bloco.

5.3.2 Serviço de Metadados

Nesta seção, a implementação do Serviço de Metadados é detalhada. O serviço supracitado foi desenvolvido em quatro classes principais e uma auxiliar, as quais são listadas a seguir de acordo com a estrutura do pacote.

- server
  - ServerList;
- server.meta
  - RaidType;
  - ServerConsole;
  - ServerInfo;
  - ServerMeta.

A classe **RaidType** é apenas uma enumeração de números inteiros representando os níveis de RAID suportados pelo sistema.

Código 5.5: Classe ServerConsole

```
public class ServerConsole {

   public static void main(String[] args){
      if(args.length < 3) {
         System.out.println("Use: java ServerConsole <processId> <raidType> <nServers>
             <verbose>");
         System.exit(-1);
      }
      boolean verbose = false;
      boolean test  = true;
      if(args.length > 3) {
         if(args[3].contains("v"))
         verbose = true;
         if(args[3].contains("n"))
         test = false;
      }

      new ServerMeta(Integer.parseInt(args[0]), Integer.parseInt(args[1]),
          Integer.parseInt(args[2]), verbose, test);
   }
}
```

A classe **ServerConsole** serve apenas como uma interface de inicialização do Serviço de Metadados, no qual o *ServerConsole* verifica os parâmetros de inicialização. Caso tenha algo errado, é apresentada uma mensagem de erro e o programa é finalizado. Em contrapartida, se os parâmetros estiverem corretos, a classe *ServerMeta* é instanciada, como demonstrado no trecho de Código 5.5.

A classe **ServerList** implementa a lista de servidores de armazenamento utilizando a classe *ArrayList* do Java. Cada item da lista é composto por um objeto da classe **ServerItem**, responsáveis pelas informações sobre servidores de armazenamento, como mostrado no trecho de Código 5.6.

Código 5.6: Campos da classe ServerInfo

```
public class ServerInfo {
   private String hostName;
   private int  port;
   private long  capacity;
   private long  size;
   private long  lastID;
   private long  lastAccessTime;

         ...
}
```

A classe **ServerMeta** é o núcleo do Serviço de Metadados, possuindo como superclasse o *DefaultSingleRecoverable*, uma classe que pertence a biblioteca *BFT-SMaRt*. Dentro de seu construtor, a classe **ServerMeta** inicializa a árvore de diretórios e a lista de servidores de armazenamento, como apresentado no trecho de Código 5.7.

Código 5.7: Declaração e construtor da classe ServerMeta

```
public class ServerMeta extends DefaultSingleRecoverable {
         ...

   public ServerMeta(int id){
      new ServiceReplica(id, this, this);
      dt   = new DirectoryTree();
      list = new ServerList();
   }

         ...
}
```

O método *appExecuteOrdered*(comando), sobrescrito da superclasse e mostrado no trecho de Código 5.8, é invocado para atender às requisições por operação onde a ordem de execução importa. O tipo de operação que deve ser executada é passado como parâmetro para a variável *reqType* e retransmitida para o método correspondente, para que ele possa continuar com o processo de execução do comando solicitado.

Enquanto que as operações que podem ser executadas em qualquer ordem devem ser atendidas pelo método *appExecuteUnordered*(comando), o qual também recebe o comando de execução como parâmetro do método.

Código 5.8: Métodos para atender às requisições

```
@Override
public byte[] appExecuteOrdered(byte[] command, MessageContext msgCtx) {

      ...

   byte[] resultBytes = null;

   try {
      ByteArrayInputStream in = new ByteArrayInputStream(command);
      ObjectInputStream  ois = new ObjectInputStream(in);

      int reqType = ois.readInt();

      switch(reqType) {
         case RequestType.CREATEDIR:
         resultBytes = criateDir(ois);
         break;

         case RequestType.DELETEDIR:
         resultBytes = deleteDir(ois);
         break;

      ...

   } catch (ClassNotFoundException | IOException e) {

      ...

}

@Override
public byte[] executeUnordered(byte[] command, MessageContext msgCtx) {

      ...

}
```

Todos os métodos da classe *ServerMeta* possuem um fluxo de execução padronizado da seguinte forma: (1) A requisição é recebida; (2) A requisição solicitada é executada; (3) O resultado é retornado ao solicitante da requisição.

O método *open*, mostrado no trecho de Código 5.9, exibe o fluxo padrão supracitado para a operação de abrir um arquivo.

Código 5.9: Exemplo de método da classe ServerMeta

```
private byte[] open(ObjectInputStream ois) throws ClassNotFoundException,
    IOException {
   String  currPath = (String)ois.readObject();
   String  tgtName = (String)ois.readObject();
   long    accTime = ois.readLong();

         ...

   Directory currDir = dt.openDirectory(currPath, accTime);
   int      result   = -1;
   long     fileSize = 0;
   FileDFS  target   = null;
   BlockInfoList bList = null;

   if(currDir == null) {
      currDir = dt.getRoot();
      result = ResultType.FAILURE;
   } else if(!currDir.existFile(tgtName)) {
      result = ResultType.FILENOTEXISTS;
   } else {
      target = currDir.getFile(tgtName);
      if( ( target.isLokedW() ) &&
         ( System.currentTimeMillis()-target.getLastAccTime() ) <= 30*1000 )
      {
         result = ResultType.FILELOCKED;
      } else {
         target.lockR();
         bList = target.getBlockList();
         fileSize = target.getMetadata().size();
         result = ResultType.SUCCESS;
      }
   }

   ByteArrayOutputStream out = new ByteArrayOutputStream();
   ObjectOutputStream  oos = new ObjectOutputStream(out);

   oos.writeInt(result);
   oos.writeObject(currDir.getDirEntries());
   oos.writeObject(bList);
   oos.writeLong(fileSize);
   oos.flush();

         ...

   return out.toByteArray();
}
```

5.3.3 Serviço de Armazenamento

Nesta seção a implementação do Serviço de Armazenamento é detalhada. O serviço supracitado foi desenvolvido em quatro classes, as quais são listadas a seguir.

- server.data;
  - MetadataModule;
  - ServerConsole;
  - Operation;
  - ServerData.

A classe **ServerConsole** serve apenas como uma interface de inicialização do Serviço de Armazenamento, no qual o *ServerConsole* verifica os parâmetros de inicialização. c Caso tenha algo errado, é apresentada uma mensagem de erro e o programa é finalizado. Em contrapartida, se os parâmetros estiverem corretos, a classe *ServerData* é instanciada.

A classe **Operation** é a classe responsável por efetivamente executar as operações solicitadas ao Serviço de Armazenamento sobre os blocos de arquivos. Ela herda a classe *Thread* para realizar o processamento em *multithread*, para que a classe seja capaz de atender multiplos clientes simultaneamente. Na parte de execução mostrado no trecho de Código 5.10, é realizada a chamada do método adequado de acordo com tipo de operação solicitada.

Código 5.10: Declaração e o método de execução da classe Operation

```
public class Operation extends Thread {

        ...

   public void run() {
      if(verbose)
         System.out.println("Cliente conectado do IP "
         +clientSocket.getInetAddress().getHostAddress());

      try {
         InputStream in = clientSocket.getInputStream();
         ObjectInputStream ois = new ObjectInputStream(in);

         int  reqType = ois.readInt();
         Block block  = (Block)ois.readObject();

         switch(reqType) {
            case(RequestType.CREATE):
               create(block);
               break;

            case(RequestType.DELETE):

         ...

   }

   ...

}
```

ServerData é a classe responsável por lidar diretamente com os clientes, de modo que é ela quem sabe a porta onde o servidor deve aguardar pela conexão do cliente, a sua capacidade total de armazenamento e quanto do espaço já foi ocupado.

O trecho do código apresentado no Código 5.11 é a parte principal do fluxo de execução da classe *ServerData*. O fluxo começa com a classe esperando a conexão do cliente. Quando a conexão é realizada, o fluxo segue para a criação da instancia da classe *Operation*.

Código 5.11: Declaração e o método de execução da classe Operation

```
while(true) {
   iterations++;
   if(verbose)
      System.out.println("Aguardando cliente...");
   try {
      Socket clientSocket = serverSocket.accept();

      Operation op = new Operation(clientSocket, dirName, verbose);
      op.start();
   } catch (SocketException e) {
      e.printStackTrace();
      System.exit(0);
   }
}
```

A classe **MetaDataModule** realiza a comunicação com os servidores de metadados. Tal comunicação é alcançada utilizando as facilidades da biblioteca *BFT-SMaRt*, na qual o *MetaDataModule* age no papel de um cliente do protocolo de comunicação, solicitando serviços aos servidores de metadados.

5.3.4 Cliente

Esta seção detalha a implementação do Cliente. O serviço supracitado foi desenvolvido em cinco classes contidas no pacote *client*, as quais são listadas a seguir.

- client
  - ClientConsole;
  - ClientDFS;
  - ClientServerSocket;
  - Option;
  - ClientTest.

Observe que a classe **ClientTest** é utilizada apenas para realização de testes, na execução normal ela é ignorada pelo sistema e não deve ser chamada. O trecho de Código 5.12 demonstra a inicialização dos *threads* da classe *ClientDFS*, responsáveis por efetivamente inicializar os testes.

Código 5.12: Preparação de teste na classe ClientTest

```
long[] values = new long[numThreads];
Client[] c = new Client[numThreads];

for (int i = 0; i < numThreads; i++) {
   try {
      Thread.sleep(100);
   } catch (InterruptedException e) {
      e.printStackTrace();
   }

   System.out.println("Launching client " + (initId + i));
   c[i] = new ClientTest.Client(values, i, opsType, numberOfOps, interval);
}

      ...
```

Ao inicializar a classe **ClientTest** é necessário informar, como um dos parâmetros de entrada, qual operação os clientes devem solicitar. O trecho de Código 5.13 representa o modo como a operação solicitada é repassada para o restante do sistema. Ao fim da execução de todas as *threads*, apenas a instância portadora do atributo ID inicial imprime os resultados para o usuário antes de finalizar sua execução.

Código 5.13: Execução de teste na classe ClientTest

```
for (int i = 0; i < numberOfOps; i++, req++) {
   System.out.print("Sending req " + req + "...");

   try {
      switch(opsType) {
         case(READ):
            last_send_instant = System.nanoTime();
            cdfs.open("test_"+id+"_"+i);
            st.store(System.nanoTime() - last_send_instant);
            break;

         case(WRITE):
            last_send_instant = System.nanoTime();
            cdfs.create("test", "test_"+id+"_"+i);
            st.store(System.nanoTime() - last_send_instant);
            break;

      ...

   if (id == initId) {
      System.out.println(this.id + " // Average time for " + numberOfOps + "
          executions (-10%) = " + st.getAverage(true) / 1000 + " us ");

      ...

   }

      ...
```

A classe **ClientConsole** serve apenas como uma interface entre o usuário e o *software*, na qual o *ClientConsole* apresenta um menu baseado em linha de comando, onde o usuário informa qual operação deseja executar. O *ClientConsole* valida os dados fornecidos e avisa ao módulo responsável por executar a operação desejada.

No trecho de Código 5.14 é apresentado o método que executa a operação de criação de arquivo. O código demonstra o fluxo que praticamente todos os métodos que executam alguma operação possuem: interação com o usuário, chamada a um método da classe *ClientDFS*, processamento e resultado.

Código 5.14: Exemplo de método da classe ClientConsole

```
private void create(Console con) throws ClassNotFoundException, IOException {
   System.out.println();
   System.out.println("Criar arquivo");
   String srcName = con.readLine("Nome ou local do arquivo:\n>");

   if(srcName.isEmpty())
      return;

   int result = c.create(srcName, null);

   if(result == ResultType.SUCCESS)
      System.out.println("Arquivo criado");
   else
      reportError(result);
}
```

A classe **Option** é apenas uma enumeração contendo os tipos de operações suportadas pelo sistema, as quais foram apresentadas e discutidas previamente nesse trabalho.

ClientServerSocket possibilita e gerencia as conexões *Socket* realizadas entre o cliente e o Serviço de Armazenamento. Tais conexões são necessárias para a execução correta de operações entre o cliente e os servidores de armazenamento. As conexões entre o cliente e Serviço de Metadados ocorrem graças as facilidades providas pela biblioteca *BFT-SMaRt*. O método *open* mostrado no trecho de Código 5.15 é invocado para receber os blocos de dados, enviados pelo Serviço de Armazenamento.

O método *send* é privado para que pudesse ser chamado por métodos públicos, tais como os métodos *create* e *delete*. O primeiro utiliza o método *send* para realizar o envio dos blocos de arquivo, enquanto o segundo o utiliza para enviar uma requisição de deletar um bloco. O trecho de Código 5.16 exemplifica essas relação entre os três métodos citados.

Código 5.15: Exemplo de método da classe ClientServerSocket

```
public byte[] open(Block block) throws ConnectException {
   int triedCount = 0;

   while(true) {
      try {
         clientSocket = new Socket(blockInfo.getHostName(), blockInfo.getPort());
         System.out.println("O cliente se conectou ao servidor na porta "
         + blockInfo.getPort());

         ByteArrayOutputStream bos = new ByteArrayOutputStream();
         ObjectOutputStream  oos = new ObjectOutputStream(bos);

         oos.writeInt(RequestType.OPEN);
         oos.writeObject(block);
         oos.flush();

         OutputStream out = clientSocket.getOutputStream();

         out.write(bos.toByteArray());

         InputStream       is = clientSocket.getInputStream();
         BufferedInputStream in = new BufferedInputStream(is);

         while(is.available() == 0);

         byte[] buffer = new byte[BUFFER_SIZE];
         int length = in.read(buffer);
         System.out.println(length);

         return Arrays.copyOfRange(buffer, 0, length);
      } catch(ConnectException | UnknownHostException e) {
         try {
            Thread.sleep(10*1000);
         } catch (InterruptedException e1) {
            e1.printStackTrace();
            System.exit(-1);
         }

         triedCount++;
         if(triedCount>3) {
            System.out.println("O cliente nao conseguiu conectar no servidor");
            throw new ConnectException();
         }
      } catch (IOException e) {
         ...
      }
   }
}
```

A classe **ClientDFS** pode ser considerada como a classe principal do lado cliente do sistema, visto que é nela onde estão concentrados os métodos para realização de todas as operações listadas na enumeração da classe *Option*.

Código 5.16: Exemplo de métodos da classe ClientServerSocket

```
public void create(Block block) throws ConnectException {
   send(RequestType.CREATE, block);
}

public void delete(Block block) throws ConnectException {
   send(RequestType.DELETE, block);
}

        ...

private void send(int ReqType, Block block) throws ConnectException {
   int triedCount = 0;

   while(true) {
      try {
         clientSocket = new Socket(blockInfo.getHostName(), blockInfo.getPort());
         System.out.println("O cliente se conectou ao servidor na porta "
         + blockInfo.getPort());

         ByteArrayOutputStream bos = new ByteArrayOutputStream();
         ObjectOutputStream  oos = new ObjectOutputStream(bos);

         oos.writeInt(ReqType);
         oos.writeObject(block);
         oos.flush();

         OutputStream out = clientSocket.getOutputStream();

         out.write(bos.toByteArray());

         clientSocket.close();

         return;
      } catch(ConnectException | UnknownHostException e) {

            ...

      }
   }
}
```

Quando um método não possui relação com qualquer operação que envolva arquivos, o cliente comunica-se apenas com o Serviço de Metadados.

O trecho de Código 5.17 representa o método responsável pela operação de criar um novo diretório. Tal operação não envolve nenhum arquivo, logo, o cliente não precisa interagir com o Serviço de Armazenamento. Como demonstrado na Figura 5.1, a conexão entre o cliente e o Serviço de Metadados é feita utilizando-se as facilidades da biblioteca *BFT-SMaRt*, através dos métodos *invokeOrdered* e *invokeUnordered*.

O método *invokeOrdered* é chamado para requisitar operações em que a ordem de execução é relevante. Entretanto, caso as requisições possam ser executadas em qualquer ordem, o método *invokeUnordered* é quem deve ser invocado.

Código 5.17: Exemplo de método da classe ClientDFS

```
public int createDir(String tgtName) throws ClassNotFoundException, IOException {
   Metadata metadata = new Metadata(System.currentTimeMillis());

   ByteArrayOutputStream out = new ByteArrayOutputStream();
   ObjectOutputStream  oos = new ObjectOutputStream(out);

   oos.writeInt(RequestType.CREATEDIR);
   oos.writeObject(currPath.toString());
   oos.writeObject(tgtName);
   oos.writeObject(metadata);
   oos.writeLong(System.currentTimeMillis());
   oos.flush();

   byte[] bytes  = this.proxy.invokeOrdered(out.toByteArray());

   ByteArrayInputStream in = new ByteArrayInputStream(bytes);
   ObjectInputStream  ois = new ObjectInputStream(in);

   int result = ois.readInt();
   currDir   = (DirEntries)ois.readObject();

   if(result == ResultType.FAILURE) {
      currPath = currDir.getPath();
   }

   return result;
}
```

Quado um método envolve alguma operação relacionada com arquivos, é necessário que o Cliente comunique-se tanto com o Serviço de Metadados quanto com o de Armazenamento. O método representado pelo trecho de Código 5.18 é responsável operação de criação de um arquivo, por isso ele é chamado de *create*.

Como dito anteriormente, a comunicação entre o cliente e o Serviço de Metadado é realizada utilizando a biblioteca *BFT-SMaRt*. Contudo, a comunicação entre o cliente e o Serviço de Metadados é feita utilizando uma conexão *Java Socket*, implementada pela classe *ClientServerSocket*.

Código 5.18: Exemplo de método da classe ClientDFS para operação sobre arquivos

```
public int create(String fileName, String tgtName) throws ClassNotFoundException,
    IOException {
   int result = ResultType.FAILURE;
   try {
        ...

      byte[] bytes  = this.proxy.invokeOrdered(out.toByteArray());

        ...

      ClientServerSocket[] css = new ClientServerSocket[nServers];
      byte [] buffer = new byte[blockSize];

      switch(raidType) {
         case(RaidType.RAID0):
         for(int i=0; i<nServers; i++) {
            Arrays.fill(buffer, (byte) 0);

            bInfo = bList.get(i);

            bis.read(buffer, 0, blockSize);

            Block block = new Block(bInfo.getID(),buffer);


            css[i] = new ClientServerSocket(bInfo, block, Option.CREATE, verbose);
            css[i].start();
         }
         for(int i=0; i<nServers; i++) {
            css[i].join();
         }
         for(int i=0; i<nServers; i++) {
            if(css[i].failure()) {
               failure(bList.get(i));
               result = ResultType.FAILURE;
               break;
            }
         }
         break;

         case(RaidType.RAID1):

      ...

         case(RaidType.RAID5):

      ...

      }

   ...

}
```

5.3.5 Executando o Sistema

O código-fonte de todo o projeto pode ser encontrado no repositório do *GitHub*, no seguinte endereço: https://github.com/diogoAF/tccRAID. Vale ressaltar que todas as bibliotecas necessárias, incluindo o *BFT-SMaRt*, já estão inclusas no projeto. Para a execução do sistema, são necessárias ao menos oito máquinas remotas, sendo três delas para o Serviço de Metadados, quatro para o Serviço de Armazenamento (no caso do RAID 0 ou 1, devido a sua natureza, é possível utilizar apenas duas máquinas), e ao menos uma máquina executando o Cliente.

O primeiro passo é criar um arquivo denotado por **host.config** dentro de um diretório chamado **config**. Localizado no diretório **bin** existe um modelo de construção deste arquivo. Ele é necessário, pois é utilizado pelo *BFT-SMaRt* para determinar o IP e porta de cada réplica. Ao final do arquivo, deve ser inserir uma nova linha contendo o ID do servidor, endereço IP e porta pela qual ele vai escutar. Por exemplo, 7001 10.1.1.9 11100.

O segundo passo é inicializar o Serviço de Metadados. Para tal deve-se chamar a classe *server.meta.ServerConsole*, a qual deve receber quatro parâmetros, os quais serão listados a seguir:

- identificador único da réplica;
- tipo do RAID que será utilizado;
- número de servidores de armazenamento;
- "v", caso deseje mostrar na tela as informações da execução.
- "n", caso deseje não executar em modo de teste, um modo que grava os dados em um único arquivo do disco.

Porém, antes de executar o comando de inicialização, recomenda-se criar um *script* que inicialize o *BFT-SMaRt* e as outras bibliotecas. Para tal, basta seguir o formato apresentado logo abaixo. Por conveniência, doravante vamos supor que o *script* foi criado e chama-se *scriptBftSmart.sh*.

```
java -cp .:lib/BFT-SMaRt.jar:lib/slf4j-api-1.5.8.jar:lib/slf4j-jdk14-1.5.8.jar:
lib/netty-3.1.1.GA.jar:lib/commons-codec-1.5.jar $1 $2 $3 $4 $5 $6 $7 $8 $9
```

Com o *script* criado, basta executar o seguinte comando (cada um em cada máquina). Repare que cada máquina irá receber um identificador único, entretanto, o restante dos parâmetros são inalterados. Repare que é na inicialização do Serviço de Metadados que o nível de RAID é determinado e ele não pode ser alterado em tempo de execução. Neste exemplo o serviço está sendo iniciado para tratar o RAID 0 com quatro servidores de armazenamento em modo normal (ou seja, não é o modo de teste), e apresentando as informações da execução.

```
sh scriptBftSmart.sh server.meta.ServerConsole 0 0 4 vn
sh scriptBftSmart.sh server.meta.ServerConsole 1 0 4 vn
sh scriptBftSmart.sh server.meta.ServerConsole 2 0 4 vn
```

Com os servidores de metadados devidamente inicializados, deve ser iniciado o Serviço de Armazenamento. A classe que deve ser invocada é a *server.data.ServerConsole*, a qual deve receber dois parâmetros, os quais serão listado a seguir:

- identificador único da réplica;
- "v", caso deseje mostrar na tela as informações da execução.

No exemplo a seguir, o serviço será instanciado com quatro servidores apresentando as informações de execução na tela.

```
sh scriptBftSmart.sh server.data.ServerConsole 1001 v
sh scriptBftSmart.sh server.data.ServerConsole 1002 v
sh scriptBftSmart.sh server.data.ServerConsole 1003 v
sh scriptBftSmart.sh server.data.ServerConsole 1004 v
```

Nesse ponto os servidores de dados já devem ter estabelecido conexão com o Serviço de Metadados. Desta forma, resta apenas ativar o cliente. Para tal, é possível ativá-lo de dois modos, o real e o de testes. No modo real será apresentado uma interface de terminal onde o usuário poderá interagir informando qual operação ele deseja que o sistema execute. No modo de teste é passado via linha de comando qual operação deve ser executada (r para leitura ou w para escrita), o tamanho do arquivo de teste (1,100,1000 ou 10000 todos em *kilobytes*), a quantidade de *threads* clientes que serão instanciadas e a quantidade de operações. Os arquivos que são utilizados no modo de teste também podem ser encontrados na pasta **bin**.

Para executar um cliente em modo real, é necessário chamar a classe *client.ClientConsole*, a qual recebe apenas o identificador do cliente como parâmetro. No exemplo a seguir, é inicializado um cliente de id 7001.

```
sh scriptBftSmart.sh client.ClientConsole 7001
```

No caso do cliente em modo de teste, deve-se chamar a classe *client.ClientTest*. A qual possui parâmetros de entrada idênticos aos da classe *client.ClientConsole*. Desta forma, no exemplo a seguir, o cliente teste instancia 10 *threads* clientes, onde cada uma irá executar 500 operações de escrita para um arquivo de 1 KB.

```
sh scriptBftSmart.sh client.ClientTest 7001 w 1 10 500
```

5.4 Conclusões do Capítulo

Nesse capítulo foi apresentado toda a modelagem do sistema proposto, abordando a arquitetura geral e partindo para pontos mais específicos como o Serviço de Metadados, Serviço de Armazenamento e o *software* do lado do cliente. Também foram apresentados em detalhes as operações que o sistema de arquivos distribuídos proposto implementa, informando quais são as operações e suas funções. Em seguida, a implementação em si do sistema foi abordada, contendo informações técnicas de quais classes Java foram desenvolvidas e detalhes de como o desenvolvimento ocorreu, dividido em subseções para cada módulo do sistema. Por fim, foi apresentado o passo-a-passo para quem deseja conseguir o código-fonte de todo o sistema e como proceder para executá-lo no modo real ou em modo de teste.

Capítulo 6

Experimentos

Neste capítulo são detalhados os experimentos que foram realizados sobre o sistema. Explanando como e onde os experimentos foram realizados, apresentando e detalhando os dados colhidos para enfim, analisa-los para no final do capítulo apresentar a conclusão.

O objetivo dos experimentos é avaliar o desempenho do sistema sobre um ambiente de alta demanda, tanto com um único cliente quanto com múltiplas solicitações concorrentes. Para alcançar o primeiro caso, foram realizados **testes de latência** e para o último, **testes de *throughput***. Ambos os testes foram focados em apenas duas operações, leitura e escrita, sendo cada uma repetida para arquivos de quatro tamanhos distintos, 1KB, 100KB, 1MB e 10MB. Nas próximas seções estes experimentos serão descritos em maiores detalhes, porém, antes será apresentada uma breve descrição do ambiente de teste.

6.1 Ambiente de Teste

Os testes foram todos executados utilizando-se as facilidades da plataforma **Emulab-Net**, mais informações sobre a plataforma podem ser encontradas na página oficinal no endereço https://www.emulab.net/. Para este trabalho, basta saber que o Emulab-Net é uma ferramenta complexa para testes de rede com mais de 900 computadores (denominados como *nodes*) separados em diferentes categorias que possibilitam o desenvolvimento de experimentos sofisticados nas áreas de rede de computadores e computação distribuída.

Tabela 6.1: Especificações Técnicas das Máquinas

Descrição	**Valor**
Tipo	d430
Classe	PC
Sistema Operacional	Ubuntu (64bits)
Disco Rígido	200GB
Memória RAM	4GB
Nº de *Cores*	8
Velocidade do CPU	2.4GHz

Antes de se iniciar um experimento é necessário informar quantas e quais tipos de máquinas serão utilizados, além de modelar a topologia que deve ser utilizada entre os dispositivos na rede.

Para a realização dos experimentos foram solicitadas oito máquinas, onde três seriam utilizadas para execução do Serviço de Metadados, quatro para o Serviço de Armazenamento e a última para executar o lado do cliente. As especificações técnicas de cada uma das oito máquinas encontram-se registradas na Tabela 6.1. Enquanto que a Figura 6.1 representa a topologia utilizada nos experimentos descritos nesse capítulo.

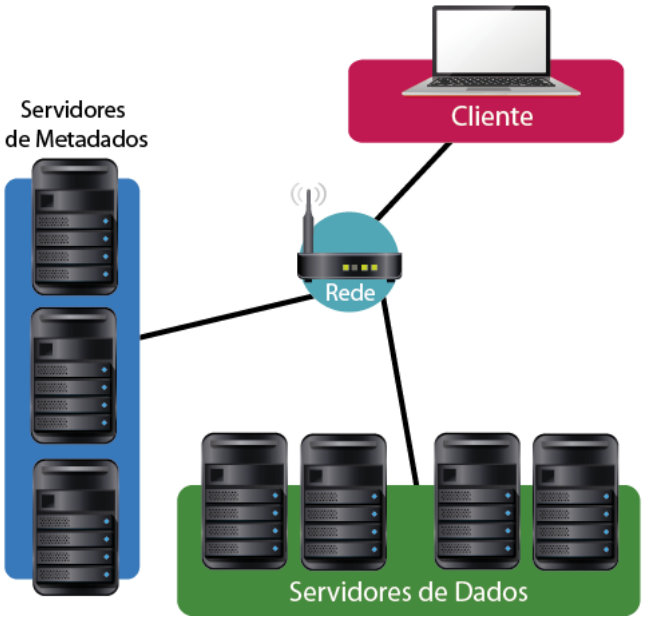

Figura 6.1: Topologia da rede nos experimentos.

Teste de latência

O objetivo do teste de latência é mensurar o tempo total que o sistema leva para executar uma operação. Nesse experimento foram testadas apenas as operações de leitura e escrita de arquivos. Cada operação foi repetida 1000 vezes de forma consecutiva utilizando-se um arquivo fixo, sendo que esse procedimento foi executado com arquivos de tamanho de 1KB, 100KB, 1MB e 10MB. Todo esse procedimento foi repetido uma única vez para cada nível de RAID suportado pelo nosso sistema, ou seja, RAID 0, RAID 1 e RAID 5. Ao fim de cada execução desse procedimento os dados eram coletados e armazenados.

Teste de *throughput*

O objetivo do teste de *throughput* é mensurar quantas operações o sistema consegue executar por segundo. Para tal, ele é realizado de forma extremamente simular ao teste de latência, a única diferença é o fato de não ser apenas um único cliente solicitando as operações, mas sim vários clientes concorrentes. Como na modelagem do experimento existe apenas uma máquina para o cliente, foram utilizadas várias *threads* onde cada uma funciona como se fosse um cliente distinto. Ao fim de cada execução do procedimento os dados eram coletados e armazenados.

O processo supracitado foi inicialmente realizado com 10 *threads* e os dados coletados, em seguida repetiu-se o mesmo processo com 20 *threads* e os dados coletados comparados com o caso anterior de 10 *threads*. Essa operação foi repetida até chegar ao ponto máximo de *threads* concorrentes em que o sistema respondia aumentando a vazão. O ponto máximo encontrado foi de 50 *threads* concorrentes.

6.2 Resultados

Esta sessão apresenta as analises quantitativas e qualitativas dos resultados obtidos na realização dos testes descritos anteriormente.

Teste de latência

Foram coletadas as média das 1000 operações, descartando-se 10% dos valores com maior desvio. A Tabela 6.2 mostra a razão entre o desvio padrão e o valor da média para os diferentes tamanhos de arquivos testados. Apenas a maior razão entre os três níveis de RAID testados estão presentas na tabela, tanto para as operações de leitura quanto de escrita. Nota-se que a variação entre as latências foi pequena, consequentemente, os valores da média simples podem ser utilizados para realizar a comparação de desempenho.

Tabela 6.2: A taxa de desvio padrão

	1KB	100KB	1MB	10MB
desvio padão/média	1,7%	6,7%	3,7%	3,1%

A Figura 6.2 representa o gráfico do teste de latência sobre a operação de leitura. Como no gráfico é difícil distinguir as linhas de cada RAID pois os valores de latência são próximos entre si, observando a Tabela 6.3 é mais fácil notar a diferença entre os valores. Neste resultado a linha apresenta formato crescente, pois conforme o tamanho dos arquivos aumentam, o tempo gasto para concluir uma operação também aumenta. Tal característica leva ao incremento da latência observada.

De acordo com Tabela 6.3 o RAID 1 apresenta o menor valor de latência entre todos os níveis de RAID para os arquivos de tamanho 1KB ou 100KB. Contudo, para arquivos

maiores o resultado inverte, o RAID 1 apresenta a maior latência. Isso pode ser explicado pelo fato dos RAID 0 e RAID 5 necessitarem da reconstrução do arquivo a partir dos bloco de dados. A reconstrução de um arquivo pequeno é mais custosa do que sua transmissão pela rede, característica essa que tende a inverter com o aumento do arquivo.

Pelos fatos acima apresentados e pelos valores da Tabela 6.3, é possível afirmar que na operação de leitura sobre arquivos de médio e grande porte a latência para cada nível de RAID obedece a relação de RAID 0 < RAID 5 < RAID 1.

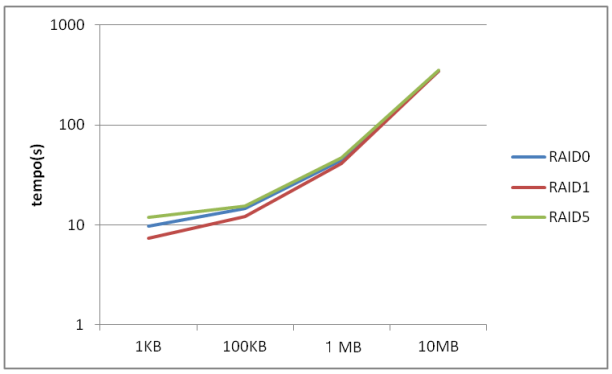

Figura 6.2: Gráfico de latência pata leitura

Tabela 6.3: Tabela de latência pata leitura(s)

	1KB	100KB	1MB	10MB
RAID 0	9.33	12.26	38.84	324.73
RAID 1	9.01	12.19	41.50	348.15
RAID 5	9.76	12.62	40.11	326.75

Diferente dos resultados obtidos nos testes de leitura, as diferenças entre os valores coletados sobre a operação de escrita para os três níveis de RAID são nítidas. Fato comprovado pela Figura 6.3 e a Tabela 6.4.

De acordo com o Gráfico 6.3 a linha de crescimento do RAID 1 é a mais intensa. A diferença entre os outros níveis de RAID tende a aumentar caso o tamanho do arquivo também aumente.

Observando apenas ao Gráfico 6.3 têm-se a errônea sensação de que o RAID 1 possui a maior latência para qualquer tamanho de arquivo. Contudo, estudando a Tabela 6.4 percebe-se que para os arquivos de 1KB e 100KB o RAID 5 é quem apresenta a maior latência. Isto pode ser explicado pela mesma razão do quê ocorreu no teste de leitura. Para arquivos pequenos o tempo para dividi-los em blocos é maior do que o de envia-los. A partir de 100KB, aproximadamente, a latência do RAID 1 supera o valor dos outros níveis.

Arquivos maiores do que 100KB mantém a mesma relação observada nos testes de leitura, RAID 0 < RAID 5 < RAID 1.

Teste de *throughput*

Foram coletados apenas o maior valor de *throughput* observado durante os testes. As linhas do gráfico possuem formato decrescente, pois o crescimento do arquivo leva ao aumento do tempo de transferência, de modo que a quantidade de operações executadas por

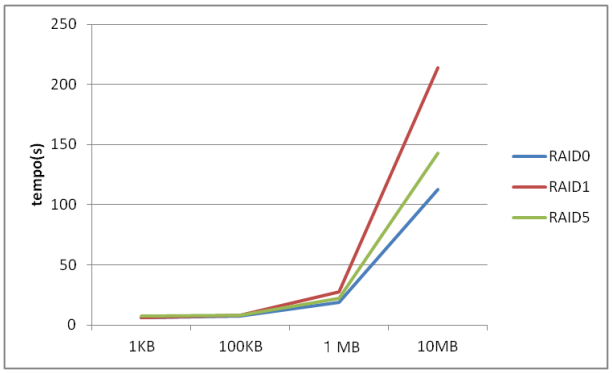

Figura 6.3: Gráfico de latência pata escrita

Tabela 6.4: Tabela de latência pata escrita(s)

	1KB	100KB	1MB	10MB
RAID 0	6.70	6.91	14.92	101.93
RAID 1	6.79	7.53	23.79	213.94
RAID 5	7.31	7.91	18.26	134.31

segundo também diminuem.

O Gráfico 6.4 e a Tabela 6.5 foram obtidos ao fim dos experimentos de *throughput* sobre as operações de leitura. Nota-se que o comportamento observado no teste de latência também ocorre no teste de *throughput*. Pois para os arquivos de tamanhos de 1KB e 100KB, o RAID 1 apresenta os maiores valores de *throughput*. Enquanto que para aquivos maiores o desempenho do RAID 1 despenca, ficando em útilmo lugar se comparado aos outros dois. A explicação para este fato pode ser a mesma do teste de latência, a relação entre os custos de reconstrução, transmissão de dados e o tamanho do arquivo.

Para os arquivos maiores do que 10MB, as taxa de *throughput* dos três níveis de RAID tendem a convergir para o mesmo valor, porém entre esse valor e 500KB o RAID 0 possui melhor desempenho.

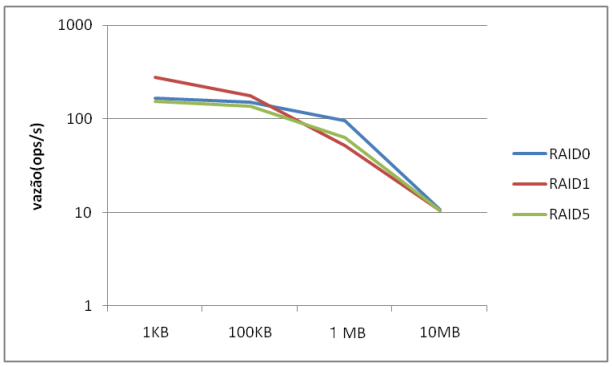

Figura 6.4: Gráfico de throughput para leitura(ops/s)

Tabela 6.5: Tabela de throughput para leitura(ops/s)

	1KB	100KB	1MB	10MB
RAID 0	167.14	151.63	95.08	10.63
RAID 1	278.86	177.43	51.92	10.52
RAID 5	152.79	135.30	63.34	10.56

O último experimento feito é o teste de *throughput* para escrita de arquivos. Diferentemente dos resultados obtidos pelos testes anteriores, em que o RAID 1 apresentava melhor desempenho apenas para os casos de arquivos pequenos, o Gráfico 6.5 mostra que o RAID 0 apresenta a melhor taxa de *throughput* desde que o arquivo não seja menor do que 1KB. Situação totalmente oposta dos testes anteriores, onde o RAID 0 apresentava bom desempenho apenas com arquivos pequenos, aqui ele apenas fraqueja lidando com

arquivos pequenos.

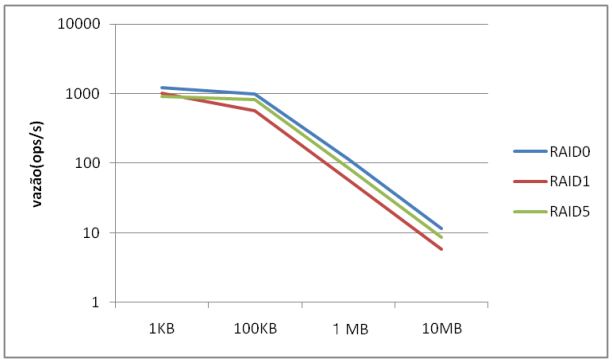

Figura 6.5: Gráfico de throughput para escrita

Tabela 6.6: Tabela de throughput para escrita(ops/s)

	1KB	100KB	1MB	10MB
RAID 0	1209.19	988.14	113.07	11.51
RAID 1	1023.54	571.43	58.01	5.76
RAID 5	914.91	830.56	84.52	8.66

6.3 Conclusões do Capítulo

Depois de obter os resultados dos testes foi possível verificar que o desempenho para leitura e escrita entre os três níveis de RAID respeita a relação de RAID 0 > RAID 5 > RAID 1 para a maioria dos casos. Deste modo é possível concluir que o RAID 5 agrega segurança e confiabilidade ao sistema de arquivos distribuídos com pouca degradação de desempenho, se comparado ao simples fracionamento de dados do RAID 0 ou espelhamento do RAId 1.

Capítulo 7

Conclusões e Trabalhos Futuros

Este capítulo apresenta as conclusões obtidas com os resultados dos experimentos realizados. Além de possuir uma breve explanação sobre quais são os rumos que podem ser tomados a fim de evoluir o sistema, quais melhorias podem ser realizadas de modo a garantir um programa mais robusto e confiável.

7.1 Conclusões

Este trabalho descreve a criação dum sistema de arquivos distribuídos tolerante à falhas, o qual mantém a confiabilidade dos arquivos armazenados usando os conceitos de RAID, e o serviço fornecido pelo sistema é protegido através da biblioteca *BFT-SMaRt*. Os resultados obtidos ao longo dos experimentos demonstram que o RAID 5 tem melhor eficiência para transferência de dados se comparado ao RAID 1, apesar do custo extra do cálculo de geração dos blocos de paridade. Por focalizar no desempenho, sem qualquer preocupação sobre a segurança dos arquivos, o RAID 0 apresenta maior taxa de transferência de dados.

Os resultados comprovam as vantagens de adotar os concentos de RAID na construção dum sistema de arquivos distribuídos,tanto em sistemas focados no desempenho quanto para os que se preocupam com a segurança dos dados armazenados. Nota-se que o RAID 5 é uma opção viável para sistemas que necessitam satisfazer condições de desempenho razoáveis, possibilitando manter a segurança dos dados armazenados.

7.2 Trabalhos Futuros

Neste trabalho foram realizados experimentos com o objetivo de realizar a comparação de desempenho para armazenamento de arquivos entre três níveis de RAID, dando ênfase nas operação de leitura e escrita. Contudo, ainda podem ser realizados outros tipos de experimentos. Verificar o que acontece quando aumentam-se os números de servidores em operação nos serviços de metado e armazenamento. Ou comparar a eficiência de recuperação de um arquivo entre os RAID 0, 1 e 5. Por fim, pode-se estender o sistema para

suportar outros níveis de RAID, como por exemplo o RAID 50.

Atualmente todo o gerenciamento realizado pelo Serviço de Metadados ocorre em tempo de execução, sendo salvo apenas em memória, pode-se modificar o código para que o mesmo possibilite que as informações gerências sejam armazenadas em disco, memória não volátil. Pois da forma realizada atualmente, toda vez que o sistema é iniciado as informações gerências estão em branco, ou seja, todos os blocos de arquivos armazenados no Serviço de Armazenamento não passam de lixo, espaço de disco desperdiçado. Quando o Serviço de Metadados estiver integrado com algum tipo de banco de dados essas informações serão preservadas mesmo quando os servidores precisarem ser desligados por qualquer motivo.

Visto os resultados obtidos nos experimentos, observou-se que o sistema comporta, sem diminuir drasticamente o tempo de resposta, cinquenta usuários simultâneos solicitando diversas operações. Pode-se estudar modos afim de aumentar essa quantidade, de modo que o sistema possa ser usado em um ambiente real. Além disso, na realização dos experimentos foram observadas ocorrências de algumas exceções Java durante a execução do programa nas máquinas remotas de servidores de armazenamento. As exceções foram de *OutOfMemoryError: Java heap space*, causada pela falta de memória disponível para a máquina virtual java (ou JVM). Outra exceção foi a *OutOfMemoryError: GC Overhead limit exceeded*, exceção que ocorre quando o coletor de lixo de java consome a maioria do poder de processamento desalocando componentes desnecessários da memória, deste modo impedindo que o programa prossiga com a sua tarefa principal.

Durante os experimentos,tais exceções foram evitadas modificando as configurações básicas da JVM. Modificações como o aumento da capacidade máxima da memória alocada pela JVM. Contudo, como não é possível assegurar que qualquer computador seja capaz de aumentar a quantidade de memória alocada para a JVM, é necessário otimizar o gerenciamento da memória pelo programa.

Referências

[1] CERN Data Centre passes 100 petabytes, 2013. http://home.web.cern.ch/about/updates/2013/02/cern-data-centre-passes-100-petabytes, acessado em 9/5/2015. 1

[2] Scaling the facebook data warehouse to 300 pb, 2014. https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb, acessado em 9/5/2015. 1

[3] What is dropbox?, 2014. https://www.dropbox.com/news/company-info, acessado em 30/5/2015. 1

[4] All aws customer stories, 2016. https://aws.amazon.com/solutions/case-studies/all/, acessado em 13/7/2016. 2

[5] Powered by apache hadoop, 2016. http://wiki.apache.org/hadoop/PoweredBy, acessado em 13/7/2016. 2

[6] Alysson Bessani, João Sousa, and Eduardo Alchieri. State machine replication for the masses with bft-smart. vi, 24, 25, 28, 33

[7] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design*. Cram 101 Textbook Outlines. Academic Internet Publishers, 2006. 6

[8] Doreen L. Galli. *Distributed Operating Systems: Concepts and Practice*. Prentice Hall, 2000. 6

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. 2003. 2

[10] Dominic Giampaolo. *Practical File System Design with the BE File System*. Morgan Kaufmann Publishers, 1999. 4

[11] Tony Hey, Stewart Tansley, and Kristin Tolle. *The Fourth Paradigm: Data-intensive Scientific Discovery*. Microsoft Research, 2009. 1

[12] Fabio Kon. Sistemas de arquivos distribuído. 1994. 6, 9, 12

[13] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22:321–374, 1990. 6

[14] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 1984. 5

[15] Dmitrey Mikhailov. Ntfs file system. 2000. 6

[16] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). *SIGMOD Rec.*, 17(3):109–116, 1988. 15

[17] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007. 8

[18] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Pearson Prentice Hall, 2007. 6