RICARDO RYOITI SUGAWARA JÚNIOR

UM SISTEMA DE ARQUIVOS COM BLOCOS
DISTRIBUÍDOS E ACESSO A INFORMAÇÕES DE
LOCALIDADE DE DADOS

São Paulo
2014

RICARDO RYOITI SUGAWARA JÚNIOR

UM SISTEMA DE ARQUIVOS COM BLOCOS
DISTRIBUÍDOS E ACESSO A INFORMAÇÕES DE
LOCALIDADE DE DADOS

Dissertação apresentada à Escola Politécnica
da Universidade de São Paulo para a obtenção
do título de Mestre em Engenharia Elétrica

São Paulo
2014

RICARDO RYOITI SUGAWARA JÚNIOR

UM SISTEMA DE ARQUIVOS COM BLOCOS
DISTRIBUÍDOS E ACESSO A INFORMAÇÕES DE
LOCALIDADE DE DADOS

Dissertação apresentada à Escola Politécnica
da Universidade de São Paulo para a obtenção
do título de Mestre em Engenharia Elétrica

Área de concentração:
Engenharia de Computação

Orientadora:
Profa. Dra. Liria Matsumoto Sato

São Paulo
2014

À minha sobrinha, Júlia Mai.

AGRADECIMENTOS

Uma dissertação de mestrado somente é possível com o apoio incondicional de um grupo de pessoas.

Agradeço à Profa. Liria Sato pela dedicação, compreensão, e sabedoria em todos os momentos em que orientou este trabalho.

Ao Dr. Osorio Meirelles, pelo incentivo e inspiração que recebo desde tenra idade. Por me apresentar à Profa. Liria, e por ser um sábio amigo de todas as horas.

À minha família, por compartilhar os momentos e me apoiar sempre. Minha mãe, Dione Marcondes, conhece profundamente o significado de todos os sentimentos que vivi ao longo do processo. Sua empatia e senso de justiça foram fundamentais para a formação dos meus valores pessoais. Minha irmã Juliana Sugawara é um exemplo de determinação e competência, a quem sou grato pelo auxílio na revisão gramatical do texto. À minha sobrinha, Júlia Mai, pela alegria que nos traz em todos os momentos, com a leveza que somente a inocência das crianças é capaz de proporcionar. Ao meu pai, Ricardo Sugawara, que, apesar da distância, permanece como um grande exemplo de bondade e honestidade.

Agradeço ao meu chefe, Sr. João Eudes, pelo apoio e compreensão que tornaram possível a obtenção dos créditos das disciplinas.

RESUMO

Muitos sistemas recentes de processamento paralelo e distribuído trabalham com acesso intensivo a dados, combinando facilidades de computação e armazenamento de dados para a construção de sistemas de grande escala, com custos reduzidos. Nesses sistemas, a interligação de um grande número de nós resulta na divisão sucessiva da capacidade de transferência, tornando a movimentação de dados um importante fator de limitação de desempenho. Ao se escalonar as tarefas computacionais em nós próximos dos dados, melhorias significativas no desempenho podem ser obtidas. Entretanto, a informação sobre a localidade dos dados não está facilmente acessível para o programador. Seu uso requer a interação com rotinas internas dos sistemas de arquivos, ou exige a adoção de um modelo de programação específico, normalmente associado a uma plataforma de execução já preparada para escalonar tarefas com aproveitamento da localidade dos dados. Este trabalho desenvolve a proposta de um mecanismo e interface para prover acesso a informações de localidade, além de permitir o controle da distribuição de novos dados. As operações de consulta e controle são realizadas por meio de arquivos e diretórios especiais, adicionados de forma transparente a um sistema de arquivos com blocos de dados distribuídos, apropriado para a execução em ambientes de processamento paralelo. O sistema é denominado *parfs* e permite obter as informações de localidade com operações de leitura e escrita em arquivos comuns, sem a necessidade do uso de bibliotecas ou modelos de programação específicos. Testes foram realizados para avaliar a proposta. Utilizando o escalonamento seletivo de operações de acesso a dados, baseado na informação de localidade, foram obtidos ganhos significativos de desempenho nessas operações.

Palavras-chave: Sistema de arquivos, Sistemas distribuídos, Princípio da localidade

ABSTRACT

Many recent data intensive parallel processing systems are built with cost effective hardware and combine compute and storage facilities. In such systems, interconnecting large numbers of nodes results in bandwidth-bisecting networks, making data movement an important performance limiting factor. By distributing jobs near data, significant performance improvements can be achieved. However, data locality information is not easily available to the programmer. It requires interaction with file system internals, or the adoption of custom programming and run-time frameworks providing embedded locality-aware job scheduling. This document develops a proposal of a mechanism and an interface to provide locality information and new data placement control. The query and control tasks are performed through special file and directories transparently added to a distributed file system, suitable for parallel processing environments. The file system is called *parfs* and allows the use of locality by read and write operations over regular files, with no need of libraries or specific programming models. Tests were conducted to assess the feasibility of the proposal. Through selective scheduling of data operations, based on locality information, significant performance gains were obtained in such operations.

Keywords: File systems, Distributed Systems, Locality Principle

LISTA DE FIGURAS

Figura 2.1	Arquivo dividido em blocos distribuídos	21
Figura 3.1	Arquitetura geral do Network Parallel File System (NPFS), adaptado de [1].	29
Figura 3.2	Arquitetura geral do Google File System (GFS) e do Hadoop Distributed File System (HDFS), adaptado de [2].	31
Figura 3.3	Política padrão de distribuição de réplicas do HDFS.	33
Figura 3.4	Componentes do Hadoop associados ao aproveitamento da localidade de dados.	34
Figura 4.1	Componentes do sistema e suas interações básicas.	39
Figura 4.2	Cálculo com números inteiros da posição da operação dentro do bloco de dados	40
Figura 4.3	Formação do espaço de nomes	41
Figura 4.4	Visão simplificada dos módulos internos do <i>parfscli</i> e suas interações com os demais componentes do sistema	43
Figura 4.5	Fluxo de dados num sistema de arquivos implementado com Filesystem in Userspace (FUSE). Adaptado de [3].	44
Figura 4.6	Arquivos e diretórios de consulta e controle formando novos ramos e folhas na árvore do espaço de nomes	48
Figura 4.7	Fluxo proposto para manutenção dos parâmetros de criação de arquivos	53
Figura 4.8	Blocos desalinhados com o tamanho dos registros do arquivo.	55
Figura 6.1	Visão simplificada do ambiente de testes	76
Figura 6.2	Número de requisições atendidas por segundo em função do número de conexões, para as operações <i>readdir</i> , <i>lookup</i> e <i>inode</i> . Cenário com saturação.	79
Figura 6.3	Latência das operações <i>readdir</i> , <i>lookup</i> e <i>inode</i> em função do número de conexões. Cenário com saturação.	80
Figura 6.4	Latência da operação <i>lookup</i> em função do número de conexões. Cenário sem saturação.	81
Figura 6.5	Desempenho para operações de leitura, releitura e leitura randômica nos casos de uso de <i>socket</i> e acesso direto aos blocos de dados.	82
Figura 6.6	Latência das operações de leitura com blocos de 4 Kb, nos casos de uso de <i>socket</i> e acesso direto aos blocos de dados.	83
Figura 6.7	Latência das operações de leitura com blocos de 128 Kb, nos casos de uso de <i>socket</i> e acesso direto aos blocos de dados.	84

Figura 6.8	Aumentos na taxa de transferência devido a leitura de regiões do arquivo com blocos armazenados localmente. Áreas sombreadas indicam as regiões locais.	85
Figura 6.9	Resultados das taxas de transferências agregadas dos testes com localidade e sem localidade	86

LISTA DE TABELAS

Tabela 5.1	Sumário das funções implementadas no <i>parfsds</i>	63
Tabela 5.2	Registros de metadados do <i>parfsmds</i>	67
Tabela 5.3	Sumário das funções implementadas no <i>parfsmds</i>	68
Tabela 6.1	Detalhes da configuração dos servidores virtuais e do arquivo de testes	77
Tabela 6.2	Quantidade de dados lidos em cada máquina e <i>speedups</i> médios como função do número de nós.	87
Tabela 6.3	Taxas médias de transferências de blocos locais e remotos obtidas com o HDFS.	88

LISTA DE ABREVIATURAS E SIGLAS

SAD	Sistema de Arquivos Distribuídos
SAP	Sistema de Arquivos Paralelo
NFS	Network File System
HDFS	Hadoop Distributed File System
GFS	Google File System
LAHPC	Laboratório de Arquitetura e Computação de Alto Desempenho
FUSE	Filesystem in Userspace
PVFS	Parallel Virtual File System
TCP	Transmission Control Protocol
SO	Sistema Operacional
VFS	Virtual File System
NPFS	Network Parallel File System
MV	Máquinas Virtuais

SUMÁRIO

1	INTRODUÇÃO	15
1.1	Motivação	16
1.2	Objetivo	17
1.3	Metodologia	17
1.4	Organização do texto	18
2	SISTEMAS DE ARQUIVOS DISTRIBUÍDOS E PARALELOS	19
2.1	Armazenamento persistente	19
2.2	Arquivos e diretórios	19
2.3	Gerenciamento do espaço	20
2.3.1	Granularidade do espaço	20
2.3.2	Distribuição de blocos ou arquivos	20
2.4	Transparência	21
2.5	Caches	22
2.6	Consistência e concorrência	23
2.7	Replicação	23
2.7.1	Controle da alocação de blocos ou réplicas	24
2.8	Princípio da localidade	24
2.8.1	Distância e custo	25
2.9	Metadados	25
2.9.1	Localidade das informações	26
2.10	Comentários sobre as diferenças entre sistemas de arquivos distribuídos e paralelos	26
2.11	Sumário	26
3	TRABALHOS RELACIONADOS	28
3.1	Alguns trabalhos que abordam a localidade de dados	28
3.1.1	Network Parallel File System (NPFS)	28
3.1.2	Google MapReduce e Apache Hadoop	30
3.2	GFS e HDFS	30
3.2.1	Modelos de consistência	31
3.2.2	Aderência do modelo de consistência à aplicação	32
3.2.3	Replicação de blocos	32
3.2.4	Uso da localidade de dados pelo Hadoop	33
3.2.5	Camadas de abstração para substituir o HDFS no Hadoop	34
3.3	Relevância futura da técnica	34
3.3.1	Uso da localidade de dados para novas estratégias de alocação	35

3.4	Considerações sobre os trabalhos relacionados	36
4	UMA PROPOSTA PARA O ACESSO À LOCALIDADE DOS DADOS	37
4.1	Arquitetura geral do <i>parfs</i>	38
4.2	Servidor de dados <i>parfsds</i>	39
4.2.1	Cálculo das posições das operações	40
4.3	Servidor de metadados <i>parfsmds</i>	41
4.3.1	Identificação de nomes de arquivos e diretórios	41
4.3.2	Atributos básicos do conjunto de metadados	42
4.4	Cliente <i>parfscli</i>	42
4.4.1	Interface com o sistema operacional	43
4.4.2	Comunicação com os servidores de dados e metadados	44
4.5	Solução proposta para o acesso à localidade dos dados	45
4.5.1	Identificador reservado	45
4.6	Conteúdo do arquivo de consulta de localidade	46
4.7	Efeito da solução proposta sobre o espaço de nomes	47
4.8	Configuração da alocação de dados para novos arquivos	48
4.9	Técnica para disponibilizar os arquivos e diretórios de consulta e controle	49
4.9.1	Arquivos e diretórios de consulta e controle	49
4.9.2	Mascaramento no inode	50
4.9.3	Operações em arquivos e diretórios com máscaras	51
4.10	Criação de novos arquivos	52
4.10.1	Configuração para o valor padrão	54
4.11	Extensibilidade da proposta	54
4.11.1	Controle da alocação dos blocos de dados	54
4.11.2	Blocos desalinhados com a estrutura	55
4.11.3	Arquivos com registro de tamanho fixo	56
4.11.4	Registros de tamanhos variáveis: arquivos de texto	56
4.11.5	Registros de tamanhos variáveis: arquivos delimitados por sequência de caracteres	57
4.12	Considerações sobre a proposta	58
5	DETALHAMENTO DA IMPLEMENTAÇÃO	60
5.1	Algumas decisões sobre as características da implementação	60
5.1.1	Serviço de metadados centralizado	60
5.1.2	Modelo de consistência	61
5.1.3	Replicação de dados e metadados	61
5.2	Servidor de dados <i>parfsds</i>	62
5.2.1	Operações do servidor de dados	62
5.2.2	Parâmetros de configuração e armazenamento dos blocos	64

5.2.3	Outros aspectos da implementação do parfsds	64
5.3	Servidor de metadados <i>parfsmds</i>	65
5.3.1	Atributos e mapeamento de blocos	65
5.3.2	Operações com arquivos e diretórios	66
5.3.3	Comentários sobre a implementação do parfsmds	68
5.4	Cliente <i>parfscli</i>	69
5.4.1	Parâmetros de configuração	70
5.4.2	Tratamento das requisições do sistema operacional	71
5.4.3	Operações do componente cliente	71
5.4.4	Gerenciamento da tabela de parâmetros	73
5.4.5	Acesso direto a dados locais	73
5.5	Considerações sobre os aspectos de implementação	74
6	TESTES REALIZADOS	75
6.1	Descrição do ambiente de testes	75
6.1.1	Linha de base de desempenho	76
6.2	Caracterização do desempenho do servidor de metadados	77
6.2.1	parfsmds: cenário com saturação	78
6.2.2	Capacidade de atendimento	78
6.2.3	Latência das operações	80
6.2.4	parfsmds: cenário sem saturação	80
6.2.5	Comentários sobre a caracterização do parfsmds	81
6.3	Estratégias para movimentação dos dados no componente cliente	81
6.4	Avaliação da proposta: blocos de dados locais e remotos	83
6.5	Testes com os arquivos de consulta de localidade	84
6.5.1	Resultados e discussão	86
6.6	Comparação de desempenho com HDFS	87
7	CONCLUSÕES E TRABALHOS FUTUROS	89
	REFERÊNCIAS	90
A	USO DOS ARQUIVOS DE LOCALIDADE	93
A.1	Obtendo regiões de dados locais	93
A.2	Emprego do programa <i>iozone</i>	94
B	CONFIGURAÇÕES DO HDFS	96
B.1	Localidade dos blocos no HDFS	96
B.2	Arquivos de configuração	96

1 INTRODUÇÃO

Sistemas que processam quantidades massivas de dados, da ordem de múltiplos *terabytes*, requerem interação entre o ambiente de processamento paralelo e distribuído e uma infraestrutura para o armazenamento de seus arquivos. O emprego de um sistema de arquivos distribuídos (SAD) oferece um ponto de acesso comum a arquivos para toda a hierarquia de hardware e software [4, 2].

Além de um ponto único de acesso, um sistema de arquivos distribuídos oferece uma visão unificada para um conjunto de arquivos armazenados de modo distribuído. O acesso se dá por meio de um caminho que identifica unicamente o arquivo em toda a rede, e ocorre por meio de operações comuns de leitura e escrita, independente do nó utilizado. Conforme a implementação, é estabelecida uma transparência no acesso a dados locais e remotos.

Tradicionalmente, os sistemas de arquivos distribuídos enviavam arquivos inteiros para o conjunto dos nós. Implementações recentes [5, 6, 2, 7] dividem os arquivos em blocos, permitindo que operações de leitura e escrita sejam realizadas em paralelo, trazendo desempenho agregado a essas operações. A alta disponibilidade dos dados pode ser obtida por meio da criação de réplicas dos blocos em múltiplos nós.

Quando os dados são acessados a partir de nós remotos, a comunicação entre os nós de armazenamento e de processamento pode ocupar parte significativa do tempo total de execução [8], reduzindo o aproveitamento dos recursos e o desempenho. Em sistemas onde a capacidade de transferência da interconexão não é o problema, o fator com maior peso passa a ser a latência devido à distância entre o processador e os nós de dados, especialmente no caso de sistemas geograficamente esparsos. Assim, para a obtenção de alto desempenho, é desejável reduzir a troca de dados entre os nós do *cluster* sempre que possível, seja para reduzir o uso dos canais de comunicação, ou a latência.

Uma abordagem imediata para este problema é o emprego dos nós de armazenamento também para as tarefas computacionais. Com a computação e o armazenamento coexistindo nos nós, é possível escalonar as tarefas para que sejam realizadas nas máquinas que contêm os dados de interesse em seus discos locais [9, 10, 1].

Desse modo, é possível criar ambientes de processamento de alto desempenho que estendam os modelos tradicionais de sistemas de arquivos distribuídos ao se explorar a localidade de dados.

Entretanto, essas soluções exigem aderência estrita a uma interface ou modelo de programação específicos, o que dificulta a adaptação de diversas classes de programas para o aproveitamento dessas otimizações. Em outros sistemas, as informações sobre a distribuição dos dados podem não estar disponíveis de modo simples para o programador.

Este trabalho apresenta uma proposta de um sistema de arquivos distribuídos simplificado, que oferece um mecanismo e interface para que as aplicações obtenham informações e controlem a localidade de seus dados de interesse. O acesso se dá por meio de operações comuns em arquivos e diretórios, localizados no âmbito da própria árvore do sistema, sem a necessidade do uso de bibliotecas específicas, ou de interações de baixo nível com o ambiente de armazenamento.

1.1 MOTIVAÇÃO

O surgimento de aplicações de larga escala com uso massivo de dados demandou um aumento sustentado do poder computacional associado a uma capacidade de armazenamento cada vez maior. Esses requisitos fomentaram o recente desenvolvimento de novos modelos de programação e plataformas de computação paralela e distribuída, que agregam serviços de armazenamento e processamento, como o Hadoop [10]. Ao se explorar a capacidade de processamento aplicado a arquivos armazenados localmente, a movimentação dos dados, que normalmente representa um limitador significativo de desempenho, é evitada [8]. Assim, é possível a construção de ambientes de alto desempenho com custo reduzido.

Entretanto, esses modelos de computação paralela e distribuída sobre plataformas de armazenamento [10, 9] requerem a adoção de ferramentas, bibliotecas e modelos de programação específicos. Algoritmos e aplicações que não apresentarem aderência a esses ambientes não conseguirão aproveitar a localidade dos dados de modo simples. Este fato pode restringir a utilização dessas aplicações em *clusters* menores ou exigir grande capacidade de banda para que o desempenho seja satisfatório.

Mesmo que uma aplicação permita o escalonamento seletivo de suas tarefas, não existe vantagem se não for possível combiná-lo com as informações de localidade de dados. Em casos como esse, apenas acessando estruturas internas dos sistemas de arquivos é que o programador seria capaz de obtê-la. Isso pode significar o uso bibliotecas proprietárias ou até mesmo o uso de chamadas de sistema. Esta situação constitui a principal motivação para o presente trabalho.

Adicionalmente, a informação sobre a localidade dos dados, armazenada nas estruturas internas do sistema de arquivos distribuídos, normalmente se baseia unicamente na distribuição dos blocos de dados e diz respeito aos conjuntos de *bytes* ao longo dos nós. Nem sempre esta informação será suficiente para otimizar totalmente o acesso aos arquivos, pois sua estrutura pode exigir a inspeção do seu conteúdo. Um exemplo é o processamento de arquivos de texto com linhas de comprimento variável, que se beneficiaria mais da localização de cada linha de texto ao longo dos nós do que da distribuição dos bytes do arquivo completo.

O meio de acesso às informações de localidade de dados precisa oferecer facilidades para ser instruído sobre o tipo de dado que é gravado e, ao processá-lo durante a escrita, enriquecer seus registros no sistema. Isso aumentaria a abrangência da solução para os mais diversos tipos de aplicações que trabalham com arquivos estruturados.

1.2 OBJETIVO

O principal objetivo deste trabalho é desenvolver um sistema de arquivos com blocos distribuídos que ofereça facilidades para o aproveitamento da localidade de dados, buscando o alto desempenho das aplicações. Para tanto, são propostas estratégias para melhorar o desempenho no acesso a dados locais, permitindo consultar informações de localidade e controlar a distribuição de novos dados, o que pode ser utilizado no escalonamento seletivo das tarefas no conjunto de nós. A escolha desta solução está baseada na simplicidade da interface, uma vez que não é necessário o uso de bibliotecas ou interfaces de programação específicas: o acesso ocorre por meio de operações regulares de leitura e escrita em arquivos.

A finalidade dessa solução é favorecer o emprego desse sistema em aplicações de processamento paralelo e distribuído que requerem acesso intensivo a dados, evitando situações em que o desenvolvedor precise adotar bibliotecas ou desenvolver códigos complexos para interagir com a camada de armazenamento para obter alto desempenho.

1.3 METODOLOGIA

Existem diversos conceitos e decisões de projeto associados a um sistema de arquivos com blocos distribuídos. Esses conceitos são fundamentais para estabelecer uma arquitetura que atenda às necessidades do objetivo do trabalho. Uma revisão bibliográfica foi o ponto de partida para defini-los.

Em seguida, um estudo sobre alguns trabalhos relacionados permitiu a avaliação das decisões de projeto, levando em consideração as necessidades do trabalho. Alguns sistemas relacionados [2, 6] serviram como modelo para os programas implementados, por esse fato, foram analisados com maior detalhamento.

Foi desenvolvida uma proposta para permitir o acesso às informações de localidade e controle da alocação de novos dados, sem o uso de bibliotecas. Essa proposta foi detalhada em um conjunto de técnicas e mecanismos. O resultado foi implementado junto ao sistema de arquivos, para que pudesse ser avaliado em testes.

A viabilidade da solução foi avaliada em um ambiente com um conjunto de máquinas virtuais com discos muito mais rápidos que suas interfaces de rede. Neste cenário, o uso da

localidade de dados é relevante, e o mecanismo e a interface propostos permitiram obter ganhos significativos de desempenho nos acessos a dados.

1.4 ORGANIZAÇÃO DO TEXTO

O presente texto se encontra organizado do seguinte modo:

No capítulo 2 são apresentados os principais conceitos acerca dos sistema de arquivos distribuídos, e uma breve introdução ao princípio da localidade de dados.

Em seguida, o capítulo 3 trata da relevância do uso da localidade dos dados para obtenção de alto desempenho nas aplicações. São apresentados alguns trabalhos relacionados e é discutida a relevância futura do princípio. As características fundamentais de algumas implementações modernas, e as principais decisões de arquitetura, discutidas.

O capítulo 4 propõe o desenvolvimento de um sistema de arquivos distribuídos associado a um mecanismo e interface com arquivos e diretórios, com os quais é possível interagir para obter localidade de dados e controlar a alocação de novos dados. Após apresentar uma arquitetura para a solução, é explicado de que modo as informações relevantes para o escalonamento podem ser acessadas e de que modo o sistema deve ser desenvolvido para expor essas informações e controles como se fossem arquivos comuns.

A implementação da solução é apresentada no capítulo 5. São discutidos as principais decisões de projeto, como cada componente opera, e a interação dos mesmos entre si e com o sistema operacional.

No capítulo 6 são apresentados os resultados de testes realizados na implementação para caracterizar o seu desempenho. Em seguida, para validar os efeitos do uso da localidade sobre o desempenho, são realizados testes utilizando as informações fornecidas pelo mecanismo e interface propostos.

Finalmente, as conclusões do trabalho são apresentadas no capítulo 7, onde também são indicadas propostas para trabalhos futuros.

2 SISTEMAS DE ARQUIVOS DISTRIBUÍDOS E PARALELOS

Neste capítulo são apresentados os principais conceitos associados à construção dos sistemas de arquivos distribuídos e paralelos. Sistemas de arquivos são a base da organização estrutural de dados nos sistemas operacionais, e esses sistemas provêm o acesso do usuário a essas estruturas, distribuídas em um ou mais nós de uma rede, oferecendo diferentes níveis de transparência e controle. Quando o sistema de arquivos é desenvolvido para oferecer alto desempenho e paralelismo, é muitas vezes denominado sistema de arquivos paralelo (SAP).

Muitos dos conceitos ora estabelecidos se acham fortemente ligados a decisões de projeto que determinam suas características operacionais. Essas características serão discutidas no âmbito da solução desenvolvida, conforme forem apresentadas.

2.1 ARMAZENAMENTO PERSISTENTE

A construção de sistemas de arquivos, sejam locais ou distribuídos, requer a disponibilidade de armazenamento persistente. Trata-se de uma abstração de caráter fundamental que garante que um conjunto de dados, após sua criação ou atualização, permaneça disponível até que seja explicitamente destruído, mesmo que ocorram falhas temporárias no sistema.

Esta propriedade não deve ser confundida com tolerância a falhas permanentes na camada de armazenamento, que devem ser resolvidas com estratégias de replicação, correção de erros, ou ambas, que são definidas a seguir. Por essa razão, a disponibilidade de armazenamento persistente é um requisito comum para os mais diversos projetos de sistemas de arquivo.

2.2 ARQUIVOS E DIRETÓRIOS

Um refinamento imediato para o conceito de armazenamento persistente é identificar os conjuntos de dados por meio de nomes. Os arquivos são estruturas nomeadas que representam uma sequência de dados armazenados de modo persistente. Os arquivos podem ser organizados em uma estrutura hierárquica em formato de árvore. Seus nós são denominados diretórios, podendo abrigar um conjunto de arquivos ou novos diretórios. Assim, cabe identificar os arquivos como uma abstração de alto nível para o armazenamento persistente, e os diretórios como agrupamentos de dados.

Um caminho composto por diretórios e nomes de arquivos identifica unicamente cada um dos objetos, e constituem a estrutura de nomes, que é dependente do sistema ope-

racional (SO) em questão. Um exemplo em um sistema Unix é um arquivo no caminho “/home/usuario/texto.txt”, onde “home” e “usuario” são diretórios, separados por barras, e “texto.txt” é o nome do arquivo. No Unix, outro conceito associado a sistemas de arquivos é o *ponto de montagem*. No exemplo, “/home” pode identificar um sistema de arquivos individual, montado após o denominado *root-filesystem* “/”, que é o *diretório raiz* a partir do qual todos os arquivos são encontrados.

2.3 GERENCIAMENTO DO ESPAÇO

A utilização do armazenamento persistente requer o gerenciamento do espaço nas mídias ou memórias em que os dados são gravados. Uma vez que os dados podem ser alocados em múltiplos nós, e em uma ou mais unidades de armazenamento em cada um desses, o rastreio e o controle dessa alocação se tornam fundamentais para que seja possível identificar quais dados pertencem a quais arquivos, onde e em que quantidade o armazenamento persistente se acha disponível para receber novos arquivos.

2.3.1 Granularidade do espaço

Em sistemas de arquivos locais, normalmente os arquivos são divididos em fragmentos da ordem de alguns *Kilobytes* de tamanho, armazenados em discos rígidos. No caso de sistema de arquivos distribuídos, algumas variações são comuns em projetos existentes. Em particular, observa-se que a granularidade do gerenciamento do espaço pode ser consideravelmente maior, chegando até ao arquivo completo como unidade básica de distribuição de dados. Para viabilizar o paralelismo das operações de leitura e escrita, o sistema de arquivos paralelo sempre fragmenta os arquivos em blocos distribuídos.

Alguns sistemas de arquivos distribuídos [11] apenas definem interfaces para o acesso a arquivos remotos, delegando a responsabilidade do gerenciamento do espaço para um sistema de arquivos local no servidor. Neste caso, o sistema meramente reflete este gerenciamento nas operações de dados dos clientes remotos. Assim, de acordo com a implementação em questão, o gerenciamento do espaço pode ser entendido como uma operação de controle e rastreamento sobre um conjunto de discos ou blocos remotos, ou apenas o acesso a arquivos completos, distribuídos em nós remotos, e agrupados em um único espaço de nomes.

2.3.2 Distribuição de blocos ou arquivos

O gerenciamento do espaço baseado em blocos facilita a obtenção de paralelismo, uma vez que os mesmos podem ser distribuídos ao longo de múltiplos nós para balancear a carga

Figura 2.1: Arquivo dividido em blocos distribuídos

das operações. Também pode ser dinâmico, permitindo aumento de espaço e desempenho por meio da adição de novos nós na rede. A figura 2.1 ilustra esta configuração. Em contrapartida, o gerenciamento do espaço baseado em arquivos completos pode limitar a escalabilidade do sistema de arquivos do ponto de vista de desempenho, porque não permite que múltiplos nós cooperem para atender a uma operação de leitura ou escrita em um arquivo. É, entretanto, mais simples de ser implementado.

Essa característica é muitas vezes empregada para diferenciar um sistema de arquivos distribuídos de um sistema de arquivos paralelo. O armazenamento de arquivos inteiros como unidade do espaço é típico de sistema de arquivos distribuídos (SAD), enquanto que um sistema de arquivos paralelo (SAP) procura quebrar os arquivos em blocos de menor granularidade para favorecer o paralelismo. Outros aspectos que diferenciam esses sistemas são apresentados na seção 2.10.

Entretanto, cabe notar que não há uma regra rígida neste aspecto e que versões recentes de sistemas classificados como SAD têm oferecido extensões para acesso paralelo a dados [7]. Adicionalmente, os blocos de dados podem efetivamente ser armazenados como arquivos comuns em um sistema de arquivo local do nó. Em um caso como este, arquivos com blocos de dados podem coexistir com arquivos não relacionados. A variação no espaço disponível devido aos arquivos externos precisa ser monitorada para que a informação do espaço apresentada ao usuário seja correta.

2.4 TRANSPARÊNCIA

Muitos aspectos da operação de um sistema de arquivos distribuídos podem ser implementados de modo a oferecer transparência para o usuário. O nível de transparência estabelece um critério de qualidade da implementação e determina a aderência da mesma a cada tipo de aplicação. Alguns dos principais aspectos de transparência são os seguintes:

- *Rede*: os primeiros meios para acesso remoto a arquivos requeriam que um processo de transferência remota de arquivos fosse comandada pelos usuários do sistema. O reconhecimento de que as operações em arquivos remotos poderiam se assemelhar às aquelas com arquivos locais possibilitou o desenvolvimento de sistemas de arquivos

distribuídos. Esta propriedade permite que arquivos locais e remotos sejam tratados pelo usuário com as mesmas operações, e é denominada *transparência de rede* [12].

- *Localidade*: de modo análogo, se a estrutura de nomes do sistema for comum a todos os usuários, então o caminho completo do arquivo não exige a identificação das máquinas na rede onde os arquivos ou seus blocos de dados se encontram armazenados. O sistema se encarrega de localizar e acessar remotamente os recursos associados ao arquivo. Neste caso, diz-se que o sistema provê *transparência de localidade*. Esta propriedade é desejável para qualquer sistema de arquivos distribuídos, mas a capacidade de se obter as informações acerca da localidade, ou múltiplas localidades, é um fator imperativo para que se possa levar a computação aos dados. O desenvolvimento de uma interface simples que quebre esta transparência, quando desejável, faz parte do escopo do presente trabalho.
- *Replicação*: em sistemas tolerantes a falhas, a manutenção da disponibilidade por meio da replicação de blocos ou arquivos pode ser gerenciada automaticamente. O acesso a uma ou outra réplica também pode ser decisão do sistema, com o intuito de reduzir a latência ou aumentar a taxa de leitura ou escrita.
- *Concorrência*: as semânticas de concorrência e acesso paralelo podem ser tratadas sem interação com o usuário, inclusive na resolução de conflitos para a manutenção da consistência. Conforme será visto a seguir, o compromisso entre a manutenção da consistência e o nível de concorrência permitido também é um importante fator de decisão de projeto, particularmente no caso de operações de escrita em paralelo.

2.5 CACHES

Um importante fator no desempenho de uma aplicação executando sobre um sistema distribuído é que o grande aumento na velocidade dos processadores exige que uma estratégia de *cache* seja implementada. O cache de dados pode ser realizado tanto do lado do cliente quanto nos servidores do sistema. Quando realizado no cliente, o resultado é a redução da latência e do tráfego já que ocorrem menos acessos aos servidores de rede. No servidor, o principal efeito é a redução na quantidade de acessos aos discos físicos, uma vez que a parte das operações de leitura são atendidas diretamente da memória.

Para evitar o corrompimento de dados, as operações de escrita são completadas somente depois que as informações são gravadas em um armazenamento persistente. Assim, predomina o gargalo da escrita na maioria das implementações que garantem a integridade dos blocos de dados [4]. Evidentemente, o armazenamento persistente também pode apresentar sua camada de cache de escrita com memórias voláteis, sustentadas por baterias.

Com isso, múltiplas camadas de cache podem ser encontradas dependendo do projeto do sistema, incluindo o *hardware* empregado.

A adoção de *caches* implica em múltiplas instâncias do mesmo dado coexistindo de modo distribuído no conjunto de nós. Isso requer uma abordagem para o controle da consistência e concorrência das operações, conforme será tratado a seguir.

2.6 CONSISTÊNCIA E CONCORRÊNCIA

No caso de dados imutáveis, isto é, dados que não podem ser alterados depois de gravados, altos níveis de desempenho e concorrência para sua leitura podem ser obtidos ao se dividir os blocos entre múltiplos nós e, como será tratado na próxima seção, criando múltiplas réplicas dos blocos para agregar desempenho às operações. Com o aproveitamento de caches nos clientes, sistemas altamente escaláveis podem ser construídos.

No entanto, em qualquer sistema distribuído, ou mesmo multiprocessado, onde os dados não são imutáveis, o emprego de caches e réplicas traz dificuldades para o provimento de uma visão consistente dos dados para todos os clientes ou processos, pois atualizações precisam ser propagadas para todas as réplicas e para todos os nós ou processadores que abrigam esses dados no cache.

A menos que exista algum mecanismo de coordenação de acesso, a latência na propagação dessas informações impede que a visão global seja sempre consistente. Diferentes abordagens podem ser adotadas para este problema, e caracterizam uma decisão de projeto crucial: em qualquer sistema em que exista algum tipo de dado ou computação distribuídos, é necessário estabelecer um compromisso de escolha entre a disponibilidade de concorrência ou paralelismo, e a consistência da informação apresentada. No capítulo seguinte é discutido de que modo algumas implementações relevantes tratam essa questão.

2.7 REPLICAÇÃO

A replicação no contexto de um sistema de arquivos distribuídos pode atender a múltiplos propósitos e é natural que a replicação siga a mesma granularidade da divisão física dos dados, seja em arquivos completos, ou em blocos.

Para o caso de sistemas de alto desempenho, a existência de múltiplas réplicas possibilita a divisão da carga de leitura em paralelo em um conjunto de nós. Os critérios para a criação de réplicas, bem como a escolha de qual será lida, pode levar em conta a topologia do sistema, a fim de reduzir comunicações desnecessárias entre os nós.

A existência de réplicas em nós distintos, em diferentes *racks* onde são armazenados os servidores, ou mesmo em múltiplos centros de dados, possibilita a construção de sistemas robustos que permanecem em operação mesmo quando múltiplos nós ficam indisponíveis.

Desse modo, a estratégia de alocação de réplicas é relevante para atender aos requisitos de alto desempenho e/ou tolerância a falhas, conforme desejado. Essas estratégias se associam a políticas dinâmicas ou fixas, dependendo da implementação. A existência de réplicas também exige uma abordagem para a questão da consistência, uma vez que introduzem versões distribuídas de um mesmo dado.

2.7.1 *Controle da alocação de blocos ou réplicas*

As políticas de alocação de réplicas podem considerar a topologia do ambiente, designando réplicas para nós distintos em um mesmo *rack*, criando réplicas adicionais em *racks* separados ou mesmo em centros de dados geograficamente distantes. Estas estratégias de alocação oferecem níveis elevados de proteção, pois garantem a disponibilidade dos dados contra múltiplos tipos de ocorrências que geram indisponibilidades.

Mesmo em sistemas que não trabalhem com replicação de arquivos ou blocos, a possibilidade do usuário controlar a alocação dos dados permanece relevante. No caso de um arquivo temporário a ser reutilizado pelo mesmo nó de processamento que o gera, é conveniente mantê-lo localmente. Se o dado que está sendo gravado será utilizado em seguida por um outro nó previamente conhecido, os dados podem já ser enviados neste nó para maior desempenho na próxima etapa de processamento. Se os dados serão lidos por muitos outros nós, pode ser interessante espalhar os blocos para obter distribuição de carga.

2.8 PRINCÍPIO DA LOCALIDADE

O *princípio da localidade*, também denominado *localidade de referência*, consiste na observação de que, em sistemas reais, os mesmos valores ou seus endereços de memória associados são frequentemente acessados com maior probabilidade em comparação a outros conjuntos de dados. Existem dois aspectos básicos associados ao princípio, a saber:

- *Localidade temporal*: se refere à utilização sucessiva do mesmo dado numa janela de tempo reduzida. Significa que um dado recentemente acessado possui alta probabilidade de ser requisitado novamente em uma janela de tempo próxima.
- *Localidade espacial*: indica que dados armazenados relativamente próximos a regiões já demandadas tendem a ser acessados com maior probabilidade. Esta propriedade é de fácil identificação quando se observa um programa varrendo uma sequência de dados alinhados, como um vetor na memória ou um arquivo sequencial.

Esses dois aspectos se traduzem em comportamentos preditivos bastante úteis para a otimização do acesso a dados. O emprego de caches e de algoritmos de busca antecipada

são exemplos de técnicas amplamente empregadas com o intuito de obter alto desempenho por meio da localidade.

O artigo de [13] apresenta a história do princípio e sua relevância em diversos campos de aplicação, justificando que a observação do princípio se deve ao uso de *laços* nos programas, à execução de módulos ou funções com dados privativos, e do fato das estruturas de dados serem agrupadas como vetores, matrizes, e outras estruturas de dados compactadas.

Essas características se devem ao fato de que os programas tendem a refletir o aspecto humano de dividir problemas complexos em pequenos pedaços e tratá-los individualmente [13]. Desse modo, em essência, qualquer sistema desenvolvido seguindo este fluxo tende a apresentar algum aspecto do princípio da localidade.

2.8.1 *Distância e custo*

A noção de *distância* de um recurso a um processador, ou nó de processamento, se associa a uma função que define algum tipo de *custo*. O custo pode ser entendido por um valor não decrescente e dependente do tempo, associado a um objetivo do projeto do sistema, tais como latência, tempo de transmissão, etc.

É possível identificar um número de empregos para o termo *distância*: a *distância temporal* pode ser entendida como o tempo desde a última referência, o tempo até a próxima referência, ou mesmo o tempo para acesso a um recurso armazenado em um nó remoto. A *distância espacial* pode ser representada pelo número de saltos na rede até o recurso remoto, ou o número de endereços entre uma sequência de referências [13].

O bom uso do princípio da localidade consiste, portanto, no desenvolvimento de técnicas que possibilitem a redução dessas ou de outras *distâncias* do sistema.

2.9 METADADOS

Além dos conceitos que foram apresentados, em grande parte associados aos dados que o sistema abriga, existem propriedades adicionais que também são fundamentais para a operação de um sistema de arquivos, tais como a data de criação e modificação do arquivo, seu tamanho, e atributos de segurança tais como permissões, listas de controles de acesso, etc. O agregado dessas informações forma o conjunto de *metadados*. Muitas implementações [6, 2] utilizam um serviço centralizado de metadados para coordená-las junto aos clientes, e acessam o conteúdo do arquivo por meio de um segundo serviço de distribuição de blocos de dados.

2.9.1 *Localidade das informações*

É também por meio do conjunto de metadados que se encontra o mapeamento dos arquivos para a lista de seus blocos, bem como a localização dos mesmos nos nós da rede. Se o sistema for capaz de expor tais informações para o usuário, este terá a possibilidade de decidir quais os nós que oferecerão menor distância até os dados durante uma operação em determinada região do arquivo.

Por essa razão, um meio para se obter informações para localidade de dados será essencialmente uma operação de consulta aos registros de metadados.

2.10 COMENTÁRIOS SOBRE AS DIFERENÇAS ENTRE SISTEMAS DE ARQUIVOS DISTRIBUÍDOS E PARALELOS

Embora não exista uma regra rígida para diferenciar um SAD de um SAP, alguns critérios podem ser observados. A distribuição de blocos é um dos principais aspectos associados aos SAP: conforme comentado na seção 2.3.2, tradicionalmente os SAD guardavam arquivos completos em um único nó de armazenamento, enquanto os SAP faziam a distribuição dos blocos. Entretanto, implementações modernas [5, 6, 2] classificadas como SAD fazem a distribuição dos blocos de dados. O tratamento acerca da tolerância a falhas também é um critério. Uma vez que os SAP são destinados ao uso em aplicações científicas de alto desempenho, muitas vezes são usados equipamentos dedicados ao armazenamento, de alto custo, que oferecem tolerância a falhas. Isso pressupõe que falhas na camada de armazenamento sejam tratadas como exceção, enquanto múltiplas implementações de SAD assumem a falha como norma, favorecendo o emprego de *hardware* econômico [4]. Em SAD, normalmente ocorre uma simetria e os nós de armazenamento também são empregados para processamento. Em SAP tradicionais, existe uma separação entre essas camadas. Este critério, entretanto, também não é rígido.

Existe uma clara sobreposição nessas definições. Por simplicidade, os sistemas de arquivos discutidos no presente trabalho serão denominados SAD. Conforme for relevante para a discussão, os aspectos de cada sistema que favorecem seu uso em ambientes paralelos serão comentados.

2.11 SUMÁRIO

Neste capítulo foram apresentados os principais conceitos associados à implementação de um sistema de arquivos distribuídos. Alguns dos aspectos se acham associados a compromissos entre as propriedades de transparência, consistência e desempenho. O de-

envolvimento ou a escolha de um sistema deve levar em consideração esses compromissos e decidir quais as características mais relevantes para a sua aplicação.

Também foi introduzido o princípio de localidade de dados e discutido de que modo esse é relevante para obter alto desempenho em uma plataforma de computação distribuída que realize uso intensivo de dados.

3 TRABALHOS RELACIONADOS

Uma vez que a transparência de rede é um dos principais atributos de um sistema de arquivos distribuídos bem implementado, a estrutura de arquivos e diretórios não deve permitir a distinção entre dados locais e remotos. Isso faz com que a transparência de rede se torne um fator de dificuldade no aproveitamento da localidade dos dados, nesses sistemas. Por essa razão, o uso do princípio de localidade normalmente é apoiado por meio de bibliotecas específicas, que substituem as funções de arquivos das bibliotecas padrão das linguagens, ou então se estabelecem plataformas dedicadas para a execução de alguns modelos de programação, combinando computação e armazenamento. Neste capítulo são apresentados alguns trabalhos e suas abordagens para o problema.

A discussão se inicia com uma solução baseada em uma biblioteca de acesso a arquivos paralelos. Os programas passam a utilizar suas funções no lugar das operações regulares com arquivos. Em seguida, uma discussão simplificada sobre como o modelo MapReduce aproveita a localidade de dados é apresentada. As características dos SAD que dão suporte ao uso do MapReduce em duas implementações conhecidas são detalhadas. Conforme apresentado no capítulo seguinte, suas arquiteturas serviram de base para a implementação da solução proposta.

Finalmente, uma vez que o emprego do princípio da localidade em diversos campos da computação não é novo [13], a relevância presente e futura da técnica é um assunto importante para apoiar a proposta. Portanto, é discutido de que modo a evolução das tecnologias de conectividade e redes de alta velocidade pode afetar o emprego da técnica. Como contrapartida, são indicados alguns fatores de caráter técnico e econômico que podem mantê-la relevante, bem como trabalhos recentes que demonstraram a importância da técnica para diferentes finalidades.

3.1 ALGUNS TRABALHOS QUE ABORDAM A LOCALIDADE DE DADOS

3.1.1 *Network Parallel File System (NPFS)*

O trabalho de [1], realizado junto ao Laboratório de Arquitetura e Computação de Alto Desempenho (LAHPC) procurou estender o conceito de sistema de arquivos paralelo ao incluir um mecanismo que permite iniciar programas em cada um dos nós que compõem o sistema e abrigam dados. O resultado é uma efetiva aproximação da computação ao dado, resultando em ganhos de desempenho significativos em relação ao acesso que exige busca de dados em nós remotos. A implementação foi denominada Network Parallel File System (NPFS) e é descrita sucintamente a seguir.

Figura 3.1: Arquitetura geral do NPFS, adaptado de [1].

O NPFS é formado por três componentes. O componente *mestre* realiza o gerenciamento dos servidores que compõem o sistema. Cada nó da rede executa um programa *servidor* que é coordenado pelo mestre e demandado pelos clientes. A finalidade do *servidor* é prover o acesso aos segmentos dos arquivos, que são armazenados nos discos locais dos nós que os executam. O *cliente* abstrai as operações para as aplicações que utilizam o sistema. É responsável por prover as funções primitivas de manipulação dos arquivos, comunicando-se com o *mestre* e com todos os *servidores* da rede. A arquitetura geral do NPFS, com seus componentes e suas interações, é ilustrada na figura 3.1.

Os arquivos somente são acessíveis por uma biblioteca, cujas operações devem ser usadas no lugar das chamadas padronizadas do sistema operacional. Cada arquivo no NPFS é identificado por um índice numérico e dividido em segmentos distribuídos para um grupo de servidores.

Em cada um dos servidores, o sistema permite comandar o disparo de um programa preexistente. O número de programas iniciados pode coincidir com o número de segmentos, ou seja, múltiplas instâncias de um mesmo programa são executadas em paralelo nos servidores, e cada uma trabalha com porções diferentes do conjunto de dados.

Para se determinar qual segmento dos arquivos deve ser aberto por cada um dos processos, o programa disparado deve chamar uma função que identifica a posição em um grupo de processos iniciados, bem como outra função que provê o número total de processos naquele grupo.

Uma das características do gerenciamento do paralelismo é a capacidade do NPFS de fornecer *argumentos* para os programas iniciados em paralelo. Com isso, uma lista poderia informar ao programa quais os segmentos do arquivo estão armazenados localmente. Entretanto, os programas iniciados não são capazes de acessar os dados dos arquivos paralelos sem se associar à biblioteca, o que quebra parcialmente a transparência deste gerenciamento.

3.1.2 Google MapReduce e Apache Hadoop

Mais recentemente, modelos e ambientes que reduzem a complexidade associada à distribuição de tarefas em sistemas de processamento paralelo emergiram. Um exemplo desses desenvolvimentos é o MapReduce [9]. O MapReduce é um modelo de programação associado a um ambiente de execução que, descrito sumariamente, aceita um grande número de pares chave/valor como entrada para uma função de processamento. Esse passo, denominado *map*, executa em paralelo em todos os nós e gera um número de pares chave/valor intermediários em cada um. Os pares intermediários são redistribuídos de acordo com uma partição efetuada sobre o campo chave, para que então esses resultados intermediários possam ser reduzidos, também em paralelo, na fase final, denominada *reduce*.

Uma vez que a função *map* depende unicamente dos pares chave/valor, a operação pode ser escalonada de qualquer maneira no conjunto de nós, desde que sejam cobertos todos os dados de origem. Isso provê ao modelo a oportunidade de aproveitar a localidade de dados cooperando com o ambiente de armazenamento de arquivos. Assim, é do interesse do escalonador das operações *map* ter conhecimento sobre a distribuição dos dados de entrada. Durante a fase de redução, as funções também podem gravar os resultados finais localmente.

A seguir são apresentados os SAD associados às implementações do Google MapReduce [9], que é proprietária, e do Apache Hadoop, um software de distribuição gratuita [10].

3.2 GFS E HDFS

O Google File System (GFS) e o Hadoop Distributed File System (HDFS) são dois exemplos de sistema de arquivos distribuídos projetados para grande escalabilidade em *clusters* com elevado (>1000) número de nós. São desenvolvidos com o propósito específico de armazenar, de forma tolerante a falha, e com alto desempenho de leitura e escrita, grandes quantidades de dados (da ordem de múltiplos *terabytes*) processadas principalmente em serviços de Internet [6, 2].

Em ambos os casos, os arquivos são divididos em blocos gerenciados e replicados em diferentes nós do *cluster*. O objetivo desta abordagem é permitir grande escalabilidade de desempenho, favorecendo operações paralelas, e assegurar que a falha de uma ou mais máquinas não paralise todo o serviço. As duas soluções também são baseadas em um serviço de metadados centralizado, denominados, respectivamente, *GFS master* e o *Namenode* (HDFS). Os blocos de dados são servidos por componentes dedicados a esta função, os *chunkservers* (GFS) e os *datanodes* (HDFS). A figura 3.2 ilustra a arquitetura dos dois sistemas com suas respectivas nomenclaturas.

Uma vez que o HDFS foi desenhado com os princípios do artigo original do GFS, suas arquiteturas são particularmente semelhantes. A aplicação está associada a uma biblioteca

Figura 3.2: Arquitetura geral do GFS e do HDFS, adaptado de [2].

de funções por meio da qual o acesso ao SAD é feito. Esta biblioteca estabelece comunicação com o serviço de metadados (*GFS Master* e *Namenode*) para obter informações sobre a hierarquia de arquivos e diretórios, bem como para informações sobre a localização dos blocos de dados e os endereços dos servidores de dados (*Chunkserver* e *Datanode*). Os acessos aos dados são realizados por meio de um canal direto entre a biblioteca e o servidor de dados. Uma vez que pode haver um servidor de dados na própria máquina cliente, as operações com blocos locais tendem a oferecer alto desempenho. O serviço de metadados estabelece comunicação com os servidores de dados para rastreo, comando de operações de replicação, e gerenciamento do espaço de armazenamento.

Os modelos de consistência adotados nesses sistemas são discutidos a seguir.

3.2.1 Modelos de consistência

Muitas implementações de sistema de arquivos distribuídos são apropriadas para o processamento de aplicações de análise de dados imutáveis (como arquivos de históricos ou outros arquivos dados brutos, previamente coletados) e empregam modelos de consistência relaxados, que evitando acessos exclusivos ou, alternativamente, impedem a atualização de dados que gerem situações de potencial perda de consistência (por exemplo, permitindo apenas a concatenação de dados ao final dos arquivos). As condutas escolhidas pelos desenvolvedores do GFS e do HDFS foram as seguintes:

- No Hadoop Distributed File System (HDFS), depois que um arquivo é fechado após sua criação, não pode ter seu conteúdo atualizado. É permitido apenas concatenar novas informações ao final de um arquivo, num modelo de *único escritor, múltiplos leitores*: quando algum cliente abre o arquivo para concatenação, ele recebe um acesso de escrita exclusivo ao mesmo, até o final da operação [6]. O fato de existir apenas uma operação de concatenação garante a consistência para os leitores.

- No Google File System (GFS) o modelo de consistência relaxado garante a atomicidade das operações no espaço de nomes (como criação e deleção de arquivos) e em operações de concatenação ao final do arquivo, que podem ser concorrentes [2]. Para garantir a consistência, o cliente recebe a localização (*offset*) do seu registro concatenado no arquivo após a conclusão da escrita, ao invés de escolhê-lo durante a operação. Escritas comuns em regiões arbitrárias são permitidas, porém não garantem a consistência no caso em que ocorrer concorrência. No caso da leitura, a existência de réplicas possivelmente desatualizadas e de cache da localização dos blocos permite o acesso a dados inconsistentes, durante janelas de atualização dos metadados entre o cliente e o servidor.

3.2.2 Aderência do modelo de consistência à aplicação

A justificativa para a adoção desses dois modelos para o GFS e o HDFS se baseia no fato de que as aplicações para as quais se destinam são predominantemente ligadas à leitura de grandes arquivos, gravados uma vez e, muito pouco, ou nada modificados posteriormente: normalmente, no máximo são adicionados novos conteúdos ao final dos arquivos. Desse modo, é privilegiada a capacidade de alta concorrência nas operações de leitura, em detrimento da capacidade de escrita concorrente, e do relaxamento das propriedades de consistência.

Não é toda aplicação que possui essas características e, portanto, cabe ao usuário avaliar a aderência dessas decisões de projeto ao seu cenário. Efetivamente, os compromissos assumidos em relação ao modelo de consistência são um balanço entre a flexibilidade operacional do sistema, a simplicidade da implementação, e o desempenho resultante.

3.2.3 Replicação de blocos

No HDFS o usuário é capaz de controlar a alocação de réplicas (local e quantidade) por meio de uma interface de programação específica [6]. Caso o usuário abra mão desta funcionalidade, é adotada uma política preestabelecida para a distribuição de réplicas. Por padrão, são criadas 3 réplicas para cada bloco de dados, distribuídas do seguinte modo:

- A primeira réplica é gravada no nó que iniciou a gravação (réplica local);
- Uma segunda réplica é gravada em um nó localizado em outro *rack* da topologia;
- Uma terceira réplica é enviada para um nó localizado no mesmo *rack* do nó que recebeu a segunda réplica.

A figura 3.3 ilustra este esquema. Esta abordagem, também adotada pelo GFS [2], possui duas propriedades: ela reduz a taxa de comunicação de maior custo ou latência, entre

Rack C

Servidor E

2	3	4
8	9	10

Servidor F

2	3	5
8	9	11

Figura 3.3: Política padrão de distribuição de réplicas do HDFS.

racks, para acessos posteriores aos blocos, já que preferencialmente existirá uma réplica no próprio nó em que foi gerado ou em um nó adjacente; ao mesmo tempo, maximiza a disponibilidade dos dados, ao contemplar a possibilidade da falha de um *rack* completo, e não apenas de uma máquina isolada.

A política de replicação de blocos está associada às abordagens de consistência, e estão de acordo com o objetivo de prover alta disponibilidade e desempenho em um sistema baseado em componentes (máquinas) não confiáveis.

3.2.4 Uso da localidade de dados pelo Hadoop

Quando uma operação de mapeamento (*map*) do MapReduce é realizada, um componente do Hadoop denominado *JobTracker*, que é responsável pelo escalonamento das operações, obtém a localidade das réplicas a partir do serviço de metadados *Namenode*. Essa informação permite que a execução da operação, controlada pelo componente *TaskTracker*, seja iniciada na máquina que abriga os blocos de interesse, desde que a mesma disponha de recursos para iniciá-la. O diagrama da figura 3.4 ilustra os componentes associados a esse processo no HDFS para um ambiente com dois *Datanodes*.

Para o usuário que desenvolve sua aplicação no modelo MapReduce, esta operação é transparente. Caso a aplicação queira utilizar o HDFS diretamente, é necessário empregar a interface de programação provida pela biblioteca do sistema e consultar uma função denominada *getFileBlockLocations*.

Desse modo, o uso da localidade pelo Hadoop é automático se o programa for modelado como um fluxo MapReduce. Caso contrário, e embora o HDFS possa ser montado como

Figura 3.4: Componentes do Hadoop associados ao aproveitamento da localidade de dados.

um sistema de arquivo regular, somente será possível obter informações de localidade por meio da biblioteca proprietária do sistema.

3.2.5 Camadas de abstração para substituir o HDFS no Hadoop

Alguns trabalhos recentes estudaram a possibilidade de substituir o HDFS no Hadoop, utilizando outros sistemas de armazenamento distribuídos ou paralelos em seu lugar, modificando o modo com que seus dados são acessados. A principal motivação é que o HDFS é um sistema desenvolvido com um propósito bastante específico, não oferecendo semânticas completas de consistência e aderência para aplicações de caráter geral.

O trabalho de [5] demonstra que um sistema de arquivos distribuídos denominado *Gfarm*, que apresenta semânticas de consistência compatíveis com um maior número de aplicações, pode oferecer desempenho compatível com o do HDFS.

Um outro trabalho, por [14], demonstra que um sistema de arquivos paralelo tradicional, como o Parallel Virtual File System (PVFS), também pode ser utilizado com sucesso no Hadoop. Ambos os trabalhos consistiram em desenvolver uma camada de compatibilidade para fornecer as informações de localidade de dados para o Hadoop, de modo que toda a interação entre o Hadoop e o HDFS fosse reproduzida. [14] conclui que a denominação de *sistemas de arquivos para acesso intensivo a dados para serviços de internet* é apenas uma nova roupagem para os sistemas de arquivos já comumente empregados em soluções de computação de alto desempenho.

Em todos os casos, o uso da localidade não é exposto ao usuário desses sistemas, e ganhos de desempenho somente são possíveis se a aplicação for modelada como um fluxo de MapReduce e executar nessas plataformas.

3.3 RELEVÂNCIA FUTURA DA TÉCNICA

A quantidade de tráfego de rede gerada por tarefas computacionais em *clusters* pode representar um limitador de desempenho [8]. Entretanto, cabe discutir de que modo a possível adoção de tecnologias de conectividade com grande capacidade de banda, da

ordem de dezenas ou centenas de *gigabits* por segundo, pode colocar em questionamento a relevância futura do emprego da localidade de dados como estratégia para ganho de desempenho em aplicações com acesso intensivo a dados.

O artigo de [15] propõe que no futuro próximo a localidade se tornará irrelevante, embora seus princípios permaneçam válidos para a redução de contenção no acesso a discos, quando da utilização de réplicas. Também propõe que o desacoplamento entre os nós de processamento e armazenamento poderá se tornar mais comum. Entretanto, estudos recentes neste sentido, realizados por [16], indicam que permanece relevante o aproveitamento da localidade dos dados, mesmo com o desacoplamento entre computação e armazenamento.

Os trabalhos de [17, 18] mostram que a relação de banda entre os nós e as camadas imediatamente superiores da topologia de um centro de dados pode apresentar taxas entre 5 para 1, e 10 para 1, podendo chegar a 240 para 1, no caso do núcleo da rede. O uso de redes de altíssima velocidade para a interligação dos nós nas camadas mais inferiores também se encontra em oposição à tendência da criação de sistemas baseados em componentes de custo reduzido. A tolerância a falhas provida pelas estratégias de replicação dos blocos de dados permite assumir a falha dos componentes como norma, o que encoraja ainda mais o uso de servidores e componentes de interconexão econômicos.

Portanto, mesmo com redes de altíssima velocidade disponíveis, o compromisso de escolha entre a movimentação dos dados e da computação sempre cairá em uma questão essencialmente de custo, e específica para cada tipo de aplicação. A movimentação da computação parece ser mais atraente do ponto de vista do custo final [19]. Adicionalmente, uma vez que os meios de transmissão são compartilhados, a localidade deve permancer relevante para a escalabilidade de sistemas futuros. Finalmente, a degradação de desempenho trazida pela latência em sistemas geograficamente esparsos, como *grades computacionais* ou grandes nuvens computacionais, persistiria como um fator decisivo para garantir a relevância do aproveitamento da localidade temporal também nesses sistemas.

3.3.1 *Uso da localidade de dados para novas estratégias de alocação*

Além dos fatos discutidos no item anterior, o princípio de localidade se presta à aplicação de outras estratégias. O trabalho de [16] explora a possibilidade de desacoplamento entre os nós de processamento e de armazenamento no HDFS. Neste trabalho, os nós de armazenamento possuem números maiores de discos e apenas armazenam dados, sem processá-los. O resultado é que o gerenciamento centralizado dos recursos de discos é capaz de oferecer ganhos de desempenho em comparação às instalações tradicionais, sem abrir mão totalmente da localidade de dados. Outras linhas de pesquisa, onde a localidade de dados é relevante, incluem o emprego de escalonamento tardio para aumentar o uso da localidade de dados, e o particionamento do cluster em regiões com alta e baixa ativi-

dade, com o intuito de reduzir o consumo de energia [20, 21]. O escalonamento eficiente de tarefas em ambientes heterogêneos também requer considerações sobre a localidade de dados, conforme os trabalhos [22] e [23].

3.4 CONSIDERAÇÕES SOBRE OS TRABALHOS RELACIONADOS

Os trabalhos citados no presente capítulo demonstram que a relevância presente e futura do emprego da localidade de dados se dá por diversos aspectos. Exemplos para apoiar a relevância futura do princípio de localidade como estratégia incluem a tendência do uso de componentes de baixo custo para construir *clusters*, o emprego de redes com arquiteturas de bisseção de banda, latência, e o aproveitamento mais eficiente de recursos.

Nos trabalhos já apresentados, ao mesmo tempo em que a localidade se mostra relevante, o seu uso requer a adoção de bibliotecas ou modelos de programação específicos. Programas escritos para o acesso a sistemas de arquivos comuns não se beneficiam das facilidades de localidade de dados providas, a menos que sejam substancialmente modificados, trocando-se toda a camada de acesso a arquivos por novas funções ou, então, se forem reescritos em um ambiente de programação dedicado como o Hadoop.

Já os trabalhos [5], [14], e [16] acoplaram sistemas de arquivos novos ou existentes e criaram conexões para prover ao Hadoop as informações de localidade. Se o programador não puder adotar o modelo do MapReduce, não haverá benefício imediato da localidade de dados, embora a infraestrutura de armazenamento ofereça suporte. Efetivamente, o *Gfarm* pode ser montado como um sistema de arquivos regular, porém as informações de localidade não são providas ao usuário. O mesmo ocorre no caso do HDFS e de outros sistemas estudados, como o PVFS [5], [14].

O presente trabalho optou por uma abordagem diferente na tentativa de prover ao usuário um meio de interagir com as informações de metadados, enquanto permite o uso de um sistema de arquivos montado como um sistema de arquivos distribuídos comum. Uma solução é proposta no próximo capítulo, no qual seus detalhes funcionais e sua arquitetura são também apresentados.

4 UMA PROPOSTA PARA O ACESSO À LOCALIDADE DOS DADOS

As discussões apresentadas no capítulo anterior mostram que o emprego da localidade dos dados é relevante, e que os trabalhos neste sentido desenvolveram diferentes abordagens para o seu aproveitamento. Esses trabalhos consistem em prover bibliotecas de acesso e programas de gerenciamento de execuções distribuídas, tal como o NPFS [1], ou então se baseiam na criação de um sistema de arquivos distribuídos completo, para um propósito específico, como o caso do HDFS, que apoia uma plataforma como o Hadoop. Neste caso, são executados programas escritos de acordo com um modelo de programação, como o MapReduce.

Em todos os casos, o programador precisa adequar suas aplicações, novas ou existentes, às bibliotecas e padrões de desenvolvimento de cada solução. A proposta do presente trabalho busca uma solução para prover acesso às informações de localidade às aplicações, obtendo alto desempenho no acesso a dados no nó local, porém, sem a necessidade do uso de bibliotecas específicas, ou de chamadas de sistema.

Para tanto, o presente capítulo apresenta uma proposta organizada em três partes.

Primeiramente, é apresentada a **arquitetura** de um sistema de arquivos distribuídos, desenvolvido com propriedades para ambientes de processamento paralelo, que oferece uma interface através da geração de arquivos de texto para o acesso às informações de localidade, e permitindo também o controle sobre a alocação de novos dados. O sistema é denominado *parfs* (**parallel filesystem**). A apresentação inicia com uma introdução aos seus componentes individuais, suas principais funções e como ocorre a interação entre os mesmos.

Em seguida, é apresentada a proposta para a **solução de acesso à localidade dos dados**. O principal ponto é que as interações sejam realizadas por meio de arquivos especiais, introduzidos de modo transparente para o usuário, no espaço de nomes do sistema. Após uma explicação sobre o funcionamento desses arquivos, é apresentada a proposta técnica sobre como são inseridos no espaço de nomes, e como relacioná-los com os arquivos de dados gravados do usuário. Um esquema para o gerenciamento dos parâmetros operacionais, associados às operações de controle de alocação de dados, é necessário para dar apoio à solução.

Finalmente, a **extensibilidade da proposta** é discutida, com exemplos de possíveis novas funcionalidades, capazes de serem desenvolvidas por meio da solução sugerida.

4.1 ARQUITETURA GERAL DO *parfs*

O *parfs* é um conjunto de programas que coordenam a utilização de uma rede de computadores que compartilham seus discos locais, provendo a visão global de arquivos e diretórios de um SAD. Os arquivos são quebrados em blocos de dados distribuídos no conjunto de nós. Além de distribuir a carga nos casos de acesso remoto, permite também que operações ocorram em paralelo, em nós distintos. Isso dá ao *parfs* algumas características paralelas de um SAP. Se as tarefas computacionais coexistirem com os nós que abrigam os dados, a localidade pode ser aproveitada por aplicações distribuídas para ganho de desempenho no acesso a dados.

O aspecto mais significativo da arquitetura trata da escolha dos componentes, e o escopo de suas funções. A abordagem encontrada no GFS e no HDFS, que os separa em papéis bem delimitados (cliente, servidores de dados e metadados), é uma escolha natural. Uma vez que se trata de um sistema distribuído, a redução do acoplamento entre os componentes, por meio de canais de comunicação que ofereçam transparência de rede entre os mesmos, é desejável. Isso é possível quando os componentes se comunicam por meio de *sockets*, interagindo de mesmo modo, independentemente da localização de cada um na rede.

Assim, a arquitetura do *parfs* foi concebida com base nas experiências do HDFS e GFS. Os três módulos e suas interações básicas são descritas de modo sucinto a seguir:

- *parfsds*: é o servidor de dados, responsável por realizar as operações de leitura e gravação nos blocos solicitados pelos clientes. Um cliente do sistema também pode executar o *parfsds* e isto possibilita o acesso mais rápido aos blocos que estiverem armazenados localmente;
- *parfsmds*: responde às requisições dos clientes acerca de todo o conjunto de metadados do sistema, incluindo o espaço de nomes da estrutura de arquivos e diretórios, tamanho e permissões dos arquivos, bem como as informações de localidade dos blocos de cada um dos arquivos. A consistência global do sistema de arquivos distribuídos é coordenada por meio deste componente, central e único na arquitetura;
- *parfscli*: o cliente é o componente básico que deve ser executado em todas as máquinas em que se deseja ao acesso ao sistema. É o componente mais complexo da arquitetura, pois interage com todos os demais componentes e com o sistema operacional, recebendo e devolvendo os resultados para as requisições de operação com arquivos e diretórios.

A figura 4.1 apresenta esses componentes e suas interações básicas, que são detalhadas nas seções seguintes. Um nó regular do sistema deve executar o programa cliente, *parfscli* e o servidor de dados *parfsds*. O acesso aos dados se dá a partir de uma aplicação que utiliza

Figura 4.1: Componentes do sistema e suas interações básicas.

o sistema de arquivos em algum ponto de montagem. A requisição é encaminhada pelo núcleo do sistema operacional até o componente cliente, que se comunica com o servidor de dados e metadados para tratá-la.

O servidor de metadados, *parfsmds*, pode ser instalado em qualquer nó da rede, ou receber uma máquina dedicada (executando apenas este serviço) para abrigá-lo. Como se trata de um componente central, deve ser escolhido um nó com boa capacidade de memória e processador. Preferencialmente, o conjunto de metadados deve caber na memória, para que a latência nas operações seja a menor possível.

4.2 SERVIDOR DE DADOS *parfsds*

O componente de nível mais baixo é o servidor de blocos de dados denominado *parfsds*. Um arquivo no *parfs* é fragmentado em múltiplos blocos de mesmo tamanho, de 64 MB por padrão. Este tamanho de bloco é adotado no HDFS e no GFS e foi escolhido para

Figura 4.2: Cálculo com números inteiros da posição da operação dentro do bloco de dados

permitir a otimização de leituras e escritas sequenciais, além de reduzir a quantidade de metadados associada à localização dos blocos [2]. No *parfs* esse valor é configurável pela aplicação, podendo ser alterado por meio da solução de controle, que será apresentada na seção 4.8.

Por simplicidade, os blocos são armazenados como arquivos individuais em um sistema de arquivo local, e são identificados por um número inteiro, monotonicamente crescente. O servidor de metadados define o número associado a cada bloco. O uso de um sistema de arquivo local permite aproveitar o cache oferecido pelo sistema operacional para esses blocos de dados.

O *parfsds* também possui um número identificador sequencial para o seu nó. Este número é útil para determinar qual nó abriga as regiões de dados obtidas com a consulta de localidade, conforme será visto adiante, na seção 4.6. O servidor de dados aceita conexões dos clientes por meio de um *socket* Transmission Control Protocol (TCP) para prover acesso remoto aos seus dados. O cliente pode estabelecer múltiplos *sockets* para que transferências possam ocorrer em paralelo. Cada operação de dados é tratada por um processo individual e, desse modo, múltiplos processadores atendem às requisições em paralelo.

4.2.1 Cálculo das posições das operações

Quase sempre, o primeiro *byte* de um bloco de dados no *parfsds* corresponde a uma posição intermediária no arquivo visto pelo usuário, armazenado no sistema de arquivos distribuídos. Apenas no primeiro bloco as posições coincidem. Por essa razão, os endereços em que o *parfsds* opera precisam ser automaticamente ajustados pelo componente cliente. A figura 4.2 ilustra como uma posição p de uma operação em um arquivo deve ser convertida para uma posição p_3 no bloco do *parfsds*. Os cálculos devem ser realizados com números inteiros. Em particular, a divisão inteira p/t_b identifica o número do bloco em que ocorrerá a operação.

Figura 4.3: Formação do espaço de nomes

4.3 SERVIDOR DE METADADOS *parfsmds*

O componente *parfsmds* é essencialmente uma interface para um banco de dados que expõe ao cliente os metadados necessários para a operação do sistema. Foram organizados em três grupos, a saber: a hierarquia do espaço de nomes, o controle de atributos de arquivos e diretórios, e o mapeamento de arquivos para blocos de dados. Quando o usuário entra na árvore de diretórios do *parfs*, o componente cliente consulta o servidor de metadados para obter a listagem de arquivos e diretórios. Existe um primeiro diretório denominado *raiz* (ou *root*), a partir do qual toda a estrutura se estabelece.

4.3.1 Identificação de nomes de arquivos e diretórios

Cada arquivo ou diretório é identificado no *parfs* por um número único denominado *inode*. Este número está associado a uma estrutura de metadados que abriga os atributos de permissão, nome do arquivo ou diretório, além de mapear seus blocos de dados, no caso de arquivos, e sua posição na hierarquia. Todos os arquivos e diretórios, exceto o diretório raiz, possuem um *inode pai*. Este *inode pai* é uma referência para outro *inode* que representa o diretório onde está armazenado. Essa referência estabelece uma hierarquia em árvore, iniciando do diretório *raiz*, que recebe o *inode* de número um. Os demais são numerados sequencialmente, conforme vão sendo criados.

Além do número do *inode*, há um texto associado ao nome do diretório ou arquivo. Com esse campo, mais o número *inode*, e mais um campo que identifique o *inode pai*, o espaço de nomes em formato de árvore fica completo. A figura 4.3 ilustra um exemplo de formação de um espaço de nomes contendo quatro arquivos e um diretório.

O *inode* é, portanto, a chave que permite o acesso ao nome e a todos os demais atributos de arquivos e diretórios. O mapeamento dos arquivos para a sua distribuição dos blocos nos servidores *parfsds* também está associado ao *inode*, bem como os atributos de permissão, datas de criação, modificação, etc. Esses atributos são armazenados em tabelas de controle, gerenciadas pelo servidor de metadados.

4.3.2 Atributos básicos do conjunto de metadados

Além do espaço de nomes, existem atributos armazenados no conjunto de metadados que são fundamentais para que o sistema operacional possa coordenar o uso dos arquivos e diretórios. Alguns exemplos incluem:

- Permissões de acesso;
- Usuário e grupo donos do arquivo ou diretório;
- Marcação de tempos de criação, acesso e modificação;
- Tamanho do arquivo.

O conjunto de metadados é complementado pelo mapeamento de blocos que constituem o conteúdo do arquivo. Esse mapeamento inclui as regiões de dados que cada bloco contém, e em que nó se acha armazenado.

4.4 CLIENTE *parfscli*

O *parfscli* é o programa que permite ao cliente obter acesso ao sistema de arquivos distribuídos. As operações são recebidas e respondidas por meio de uma interface com o sistema operacional. Ele se comunica com os servidores de dados por meio da rede para efetuar operações de dados em blocos remotos. Um conjunto de conexões TCP é mantido para todos os servidores *parfsds* do sistema. Entretanto, acessos diretos aos blocos locais são possíveis no caso em que os dados solicitados estiverem armazenados nos mesmos. Ao mesmo tempo, o *parfscli* também necessita das informações de metadados relativos à distribuição dos blocos, localidade dos dados, permissões, etc. Para este fim, uma conexão TCP é também mantida para o servidor *parfsmds*.

Além de interagir com todos os demais componentes da arquitetura, é no *parfscli* que a solução para prover acesso às informações de localidade é implementada. Essa solução requer uma estratégia para o gerenciamento do estado, que é um conjunto de parâmetros operacionais das funções usadas para o controle de alocação de novos dados. O programa consiste em um laço principal que recebe notificações assíncronas de seus principais módulos internos, que são responsáveis basicamente por acesso a **dados**, **metadados**, gerenciamento de **parâmetros**, e atendimento às **requisições**.

Uma visão simplificada dessa organização é ilustrada na figura 4.4. Nas seções seguintes são detalhados os aspectos de cada um dos módulos, que são apoiados por filas de operações para tratar requisições e operações de dados e metadados, e tabelas para controle de parâmetros operacionais.

Figura 4.4: Visão simplificada dos módulos internos do *parfscli* e suas interações com os demais componentes do sistema

4.4.1 Interface com o sistema operacional

É uma abstração interessante em um SO permitir-se que as operações comuns com arquivos e diretórios sejam realizadas independentemente do sistema de arquivos demandado. Isso garante uma uniformidade nas operações do cliente, mesmo que se substituam as camadas mais inferiores do armazenamento. O componente nos sistemas Unix e Linux que provém essa abstração é denominado Sistema de Arquivos Virtual, ou Virtual File System (VFS) [24]. Uma operação de escrita, por exemplo, é tratada de modo transparente para o usuário pelo VFS, que então envia a operação para o código do sistema de arquivos que o armazenará.

Mesmo com o VFS, sistemas de arquivos permanecem intrinsecamente ligados ao núcleo do SO. O FUSE [3] é uma biblioteca e um módulo de *kernel* que permite que programas implementem sistemas de arquivo executando totalmente no espaço de usuário. A comunicação entre o *kernel* do sistema operacional e o programa ocorre por meio de um *socket*, e a comunicação é realizada por meio de uma biblioteca com a qual o programa é ligado [25].

O fluxo de uma operação no *parfs*, partindo de uma aplicação, é ilustrado na figura 4.5. As operações ocorrem de modo transparente para o usuário. A camada de VFS recebe a requisição e identifica tratar-se de um sistema de arquivo do FUSE. Seu módulo envia a requisição para a biblioteca *libfuse*, que se acha ligada à implementação do SAD. As operações são recebidas a partir de um *socket* entre o *kernel* e a biblioteca, e são então tratadas e devolvidas para o usuário. Sistemas de arquivo escritos com FUSE coexistem com os demais no SO.

Quando um dado é recebido no *socket*, é criada uma estrutura com dados da requisição, que possui um identificador único para a operação. Uma função interna, correspondente à operação solicitada pelo sistema operacional (como leitura, escrita, listagem de diretório, etc), deve ser, então, ativada.

Em seguida, o componente *parfscli* responde às requisições conforme as mesmas vão sendo completadas. Não há necessidade de respondê-las em ordem, mesmo em operações

Figura 4.5: Fluxo de dados num sistema de arquivos implementado com FUSE. Adaptado de [3].

relacionadas (como duas leituras sequenciais em um arquivo). O FUSE se encarrega de ordenar os dados recebidos, conforme necessário. Por essa razão, todas as operações que demandam acessos remotos são colocadas em uma fila para processamento assíncrono, e retornam assim que finalizam.

Com esta abordagem, as operações remotas se tornam não bloqueantes e não impedem o tratamento de operações rápidas em blocos locais, assim como o despacho de outras operações concorrentes.

4.4.2 Comunicação com os servidores de dados e metadados

Quando a fila de operações recebe alguma operação demandada pelo sistema operacional, uma rotina procura conexões disponíveis para o servidor de dados ou metadados, de acordo com a necessidade da operação específica. Quando um *socket* disponível é encontrado, a requisição é despachada, sendo os dados da solicitação enviados para um *buffer* de transmissão. Se não forem encontradas conexões disponíveis, a operação é mantida na fila até o próximo evento de liberação. Múltiplas conexões para um mesmo servidor de dados permitem operações concorrentes.

Quando algum dado chega a partir dos servidores de dados ou metadados, a conexão em questão possui um estado associado à operação em execução. O estado armazena, entre outros parâmetros, a quantidade de dados pendentes na operação. O evento em um *socket* pode significar a chegada de parte dos dados de uma operação de leitura. Até que a operação finalize, o laço principal continua tratando as notificações de eventos e atualizando as informações de estado.

Um evento que sinalize o fim de uma operação de dados ou metadados ativa uma função da biblioteca FUSE para retornar a resposta para o sistema operacional.

Quando uma operação de dados ocorre em blocos armazenados localmente, as conexões TCP do cliente para o *parfsds* local não são usadas. O *parfscli* abre o arquivo de bloco de dados e processa a região demandada diretamente, o que reduz significativamente a latência da operação e, conseqüentemente, aumenta o desempenho da aplicação.

4.5 SOLUÇÃO PROPOSTA PARA O ACESSO À LOCALIDADE DOS DADOS

Existem dois aspectos no aproveitamento da localidade dos dados nas operações computacionais: uma tarefa de processamento requer a leitura de dados previamente armazenados e, opcionalmente, a gravação de novos dados como resultados finais ou como entradas para etapas de processamento que seguem. No primeiro caso, procura-se levar a computação para mais perto dos dados, escalonando as tarefas em função da localidade. O segundo aspecto consiste em controlar o local da gravação de novos arquivos, também com esse propósito.

Com a premissa de se manter a transparência de rede de um SAD comum, o presente trabalho busca atender aos dois aspectos por meio da solução descrita a seguir. O ponto de acesso fundamental para a solução são arquivos e diretórios com nomenclaturas especiais que fazem parte da própria hierarquia do espaço de nomes. Esses pontos de acesso não constituem arquivos gravados pelo usuário do SAD, mas são introduzidos automaticamente para expor as funções implementadas para interação com a localidade dos dados. Tais arquivos e diretórios serão denominados **arquivos e diretórios de consulta e controle**.

Nas seções que seguem, que apresentam o funcionamento da solução, será assumido que a estrutura de pastas segue o padrão dos sistemas Unix, de acordo com a seção 2.2.

4.5.1 *Identificador reservado*

O ponto de entrada para interagir com as funções de consulta e controle é um diretório denominado **“:i”**. O *“ponto”* (“.”) que precede o nome *“i”* significa, em sistemas baseados em Unix, que se trata de um arquivo ou diretório oculto, não exibido em listagens regulares.

Esse não é um diretório real no sentido de que faz parte do conjunto de metadados. O componente cliente, que dá acesso ao sistema e é descrito na seção 5.4, o insere em cada diretório do espaço de nomes. Por essa razão, sua existência é implícita em todos os diretórios e é o único nome reservado, que não pode ser usado para a criação de um arquivo ou diretório. Todas as operações de consulta ou controle ocorrem por meio deste diretório **“:i”**.

Por este fato, será empregada a nomenclatura **arquivo ou diretório real** para designar arquivos e diretórios gravados pelo usuário no SAD, e **arquivo/diretório de consulta**,

Listagem 4.1: Obtendo informações sobre a localidade de um arquivo de nome “teste”. Apenas os 6 primeiros blocos da saída são exibidos na listagem abaixo.

```

1 $ ls -l teste
2 -rwxr-xr-x 1 root root 504782848 Jan 13 2006 teste
3 $ cat .i/local/teste
4 #size: 504782848
5 #blocks: 8
6 #block size: 67108864
7 #nodes: 5
8 #listing: node start end
9 0 0 67108863
10 1 67108864 134217727
11 2 134217728 201326591
12 3 201326592 268435455
13 4 268435456 335544319
14 0 335544320 402653183

```

ou **arquivo/diretório de controle**, para tratar dos que são inseridos para possibilitar a interação com as funções implementadas, a partir do diretório “.i”. O conteúdo desses arquivos e diretórios de consulta e controle não ficam gravados em disco. Eles são gerados dinamicamente pelo componente *parfscli*, e são apresentados a seguir.

4.6 CONTEÚDO DO ARQUIVO DE CONSULTA DE LOCALIDADE

Para obter a localidade dos dados de um arquivo em um diretório, o usuário realiza uma operação de leitura no caminho “.i/local/arquivo”, onde a palavra “arquivo” especifica o nome do arquivo real do qual se deseja consultar a localidade. O conteúdo desse arquivo de consulta é um texto contendo as informações de localidade relativas ao arquivo real. A listagem 4.1 apresenta um exemplo de consulta realizada em um ambiente *shell* padrão Unix. Nas primeira linhas, é exibida a listagem de um arquivo denominado “teste”. Por meio do comando *cat*, é obtido o conteúdo do arquivo de consulta correspondente, que se acha, neste caso, no caminho “.i/local/teste”. O resultado é a interface para obter a localidade. Cada uma das linhas correspondentes ao conteúdo são indicadas a seguir:

1. Tamanho do arquivo em bytes.
2. Informa o número de blocos que compõem o arquivo.
3. O tamanho em bytes dos blocos em que o arquivo é dividido.
4. Exibe o número de nós do SAD que abrigam esses blocos.
5. Apresenta um cabeçalho onde é especificado o formato da listagem das linhas posteriores. Cada linha seguinte é uma entrada de bloco de dados. O texto é **#listing:**

start end, indicando que o formato especifica o nó que abriga a localidade, o que corresponde a um identificador numérico único para cada servidor membro do sistema, início do bloco, e fim do bloco.

6. As linhas se repetem com os quatro números inteiros especificados no item acima.

A listagem 4.1 mostra o resultado esperado para o uso deste arquivo de consulta. Nesse exemplo, é possível identificar de que modo os blocos do arquivo foram divididos ao longo dos nós do sistema. Uma operação computacional sobre a região dos primeiros *bytes* teria acesso mais rápido a suas informações se escalonado para o nó de número zero. Se o caminho do arquivo especificado for inválido, o sistema retorna um erro de arquivo não encontrado. O caminho não precisa ser um arquivo imediatamente no diretório referenciado. Por exemplo, *.i/local/subdir/teste* corresponde a um arquivo de nome *teste* armazenado após o diretório *subdir* do diretório local. Dentro diretório *subdir* também existe um *.i* que fornece a mesma informação, porém no caminho *.i/local/teste*.

4.7 EFEITO DA SOLUÇÃO PROPOSTA SOBRE O ESPAÇO DE NOMES

Uma vez apresentado o funcionamento do arquivo de consulta de localidade, cabe discutir os efeitos desse mecanismo sobre o espaço de nomes do SAD. Em cada diretório do sistema podem ser achados arquivos e diretórios reais, gravados previamente pelo usuário. Até este ponto não há diferença entre o *parfs* e qualquer outro SAD. Esses arquivos e diretórios reais estão indicados na figura 4.6 como tais.

Uma vez que o sistema insere o ponto de entrada em cada diretório real do sistema, será encontrado um diretório de nome “.i”, fora do grupo de arquivos e diretórios reais. Entretanto, para o usuário, ele funciona como um diretório qualquer, podendo ser acessado e listado normalmente. Na figura 4.6 é possível observar o diretório “.i” no primeiro nível após o diretório raiz “/”, coexistindo com o arquivo real “*teste*” e diretório real “*dir_a*”.

No ponto de entrada “.i” são inseridos dois diretórios. O diretório “.i/local”, já apresentado, receberá os arquivos de consulta de localidade, cujos nomes reproduzem os nomes dos arquivos reais. O diretório “*control*” abriga os arquivos de controle que serão apresentados na seção seguinte.

Todos os diretórios do SAD que possuem arquivos ou subdiretórios têm sua estrutura de nomes reproduzida fielmente dentro do ponto de entrada “.i/local”. O efeito é a duplicação de ramos inteiros e folhas a partir do ponto de acesso, que forma novos ramos e folhas na árvore. Esses ramos representam exatamente a estrutura do diretório a partir do qual se entrou no “.i”. Entretanto, seus arquivos são de consulta e não reais, de modo que operações de leitura retornam a localidade conforme a seção 4.6, e não os dados do arquivo real.

Figura 4.6: Arquivos e diretórios de consulta e controle formando novos ramos e folhas na árvore do espaço de nomes

Na figura 4.6 é apresentada a duplicação do ramo depois de “.i/local” para um diretório “/” com um arquivo denominado “teste”, e um subdiretório “dir_a”, abrigando dois arquivos, “arq_1” e “arq_2”. Vale ressaltar que os arquivos desse ramo de consulta não possuem dados gravados em disco. Se trata apenas da estrutura de nomes.

4.8 CONFIGURAÇÃO DA ALOCAÇÃO DE DADOS PARA NOVOS ARQUIVOS

A visão da estrutura de consulta e controle criada dentro do ponto de entrada, apresentada na figura 4.6, possui um diretório de controle denominado “control”. Dentro desse diretório existe um arquivo de controle com nome “blksize”, que contém o tamanho do bloco, cuja finalidade é especificar a configuração que será utilizada na criação de um novo arquivo. Outros arquivos podem ser inseridos no diretório “control” para determinar outros possíveis parâmetros operacionais. Alguns exemplos desses parâmetros são apresentados na seção 4.11.

Os arquivos de controle permitem operações de leitura e escrita. A leitura retorna o valor atual do parâmetro. Ao se escrever no arquivo de controle, o parâmetro correspondente é configurado internamente no SAD. O parâmetro, assim configurado, afeta todos os novos arquivos que venham a ser gerados pelas aplicações, ou pelo usuário.

Como exemplo simplificado do uso do arquivo de controle “.i/control/blksize”, o conteúdo do mesmo pode ser lido para se obter o tamanho de bloco para o próximo arquivo. Quando é gravado com texto contendo o número em bytes do tamanho desejado, este valor é alterado. O exemplo da listagem 4.2 demonstra essas operações com ferramentas comuns de um *shell* Unix.

Nesse exemplo, as linhas 1 e 2 demonstram que o valor do parâmetro do tamanho de bloco é de 64 MB. Em seguida, na terceira linha, o parâmetro é configurado para 150 MB.

Listagem 4.2: Alterando e consultando o tamanho do bloco para o próximo arquivo.

```

1 $ cat .i/control/blksize
2 67108864
3 $ echo 157286400 > .i/control/blksize
4 $ cat .i/control/blksize
5 157286400

```

Uma operação de leitura que segue nas linhas 4 e 5 indica que o SAD assumiu o valor indicado.

4.9 TÉCNICA PARA DISPONIBILIZAR OS ARQUIVOS E DIRETÓRIOS DE CONSULTA E CONTROLE

Cabe um detalhamento sobre como os arquivos e diretórios de consulta e controle podem fazer parte do espaço de nomes do SAD. Não existe qualquer registro no conjunto de metadados no servidor *parfsmfs* dedicados aos mesmos. Apenas as informações de mapeamento dos blocos e *inodes* estão de algum modo associados aos arquivos de consulta, mas não diretamente. Uma simples listagem dos diretórios obtida diretamente a partir do *parfsmfs* não traz nem o ponto de entrada “*i*”, nem seus arquivos e diretórios de consulta e controle associados. A seguir é descrito de que modo eles são inseridos e, principalmente, como são diferenciados de arquivos e diretórios reais quando ocorre uma operação.

4.9.1 Arquivos e diretórios de consulta e controle

De acordo com a proposta, um caminho “*/dir/.i/local/teste.txt*” consiste em um arquivo de consulta que deve obter a localidade do arquivo real “*teste.txt*” a partir do diretório de nome “*/dir*”.

Tanto no caso, real quando no caso de consulta, os nomes associados ao caminho são “*teste.txt*”. Isso torna o nome menos interessante para a diferenciação entre eles, porque seria preciso varrer a estrutura até se encontrar um diretório pai de nome “*i*”, ou chegar ao diretório raiz.

Já os *inodes* sempre são números criados sequencialmente para todos os arquivos e diretórios do SAD. Isso sugere o seu uso como identificadores únicos. Portanto, a inserção de um diretório ou arquivo de consulta ou controle no espaço de nomes pode ser realizado de modo simples pelo componente cliente, ao se utilizar um novo número de *inode*. As questões que surgem deste procedimento são:

- como os novos *inodes* devem ser gerados, e

- como associar estes *inodes* aos diretórios e arquivos reais a que se referem.

4.9.2 Mascaramento no *inode*

A técnica proposta é a de estabelecer uma *máscara* no valor numérico do *inode*, que serve para diferenciar entre arquivos e diretórios reais, ou de consulta e controle. Uma vez que o *inode* do caminho real é único, o *inode* com uma máscara também se torna único. A única restrição é que arquivos e diretórios reais não podem conter *inodes* cujos valores se sobreponham aos valores reservados para as máscaras.

Inserindo uma máscara no *inode*. Exemplo. Considere que em um ambiente de 32 bits, o *inode* é um número não assinado desse tamanho. Se o byte mais significativo for utilizado como máscara para um arquivo com *inode* *0x20* em *hexadecimal*, cada um dos bits da máscara pode ser usado para identificar um tipo único de arquivo ou diretório de consulta e controle. Neste caso, para uma máscara *0x10000000* que, por exemplo, identifique a estrutura *./i/local*,

```

0x00000020  inode
|| 0x10000000  máscara ./i/local/
0x10000020  resultado

```

o resultado é um novo *inode* que associa a operação a um arquivo de consulta de localidade.

Diferenciando entre arquivos e diretórios reais ou de consulta e controle. Ao receber uma operação sobre um arquivo ou diretório, o componente cliente pode facilmente decidir se o *inode* se refere a algum arquivo de consulta ou controle fazendo uma operação lógica “E” entre o *inode* e os bytes reservados para as máscaras. Se o resultado for igual a zero, se trata de um arquivo ou diretório real. Caso contrário, se o resultado não for zero,

```

0x10000020  inode
& 0xF0000000  máscaras
0x10000000  resultado

```

significa que é um arquivo ou diretório de consulta ou controle, sendo que o valor do resultado deve corresponder a uma das máscaras definidas pelo sistema. Deste ponto, basta prosseguir com a operação correspondente àquela máscara.

Obtendo o *inode* original. O valor original do *inode* do arquivo ou diretório real pode ser facilmente recuperado com uma operação de “E” no complemento do byte reservado à máscara.

$$\begin{array}{rcl}
0x10000020 & \textit{inode com máscara} \\
& \& \underline{0x0FFFFFFF} \sim \textit{máscaras} \\
0x00000020 & \textit{inode original}
\end{array}$$

4.9.3 Operações em arquivos e diretórios com máscaras

O emprego das máscaras no *inode* possibilita a construção rápida da estrutura proposta a partir do diretório “.i”. Em cada diretório real, determinado após a conferência com a operação “E” entre o *inode* recebido e as máscaras, é inserido um diretório “.i” cujo *inode* é aquele do diretório mais a máscara. Quando um usuário acessa este diretório, a listagem do mesmo consiste em adicionar os diretórios “local” e “control” somando ao *inode pai* suas máscaras correspondentes. Dentro do diretório “.i/local”, basta filtrar a máscara e o cliente obtém o *inode* do diretório real. Assim, uma operação de listar arquivos em um diretório de consulta funciona do seguinte modo:

1. O componente cliente recebe a solicitação de listagem do diretório de consulta e realiza a operação de “E” entre o valor do *inode* e o byte reservado às máscaras.
2. Constata um resultado diferente de zero e correspondente à máscara de consulta de localidade.
3. Efetua a operação “E” do *inode* com o complemento da máscara para recuperar o *inode* do diretório real correspondente.
4. Consulta o serviço de metadados para obter a listagem do diretório real, a partir do *inode* recuperado.
5. Para cada entrada, insere a máscara associada à função de consulta de localidade e devolve a listagem final ao usuário.

A solução descrita foi implementada no componente cliente como parte das operações de listagem de diretório (para introduzir as estruturas após o “.i”), leitura (para obter as informações de localidade e o parâmetro do tamanho de bloco), e escrita (para alterar o parâmetro do tamanho do bloco). Uma vez que as operações de diferenciação são muito rápidas, requerendo apenas operações lógicas sobre as variáveis de *inode*, o desvio para as funções internas, que tratam dos arquivos de consulta e controle, ocorre de modo eficiente.

4.10 CRIAÇÃO DE NOVOS ARQUIVOS

O diretório de consulta “*i/local*”, que provê a consulta às informações de localidade de um arquivo real, depende apenas do parâmetro de qual caminho deve ser consultado. Assim, não há nenhum tipo de parâmetro operacional associado a esta operação, bastando uma consulta ao serviço de metadados e a geração do texto apresentado como conteúdo do arquivo de exemplo.

Entretanto, quando novos arquivos são gravados, espera-se que as operações no *parfs* ocorram como em qualquer outro sistema de arquivos distribuídos, ou seja, com transparência para o usuário. O parâmetro de tamanho de bloco não precisa ser configurado a cada novo arquivo. De fato, o programador pode preparar o parâmetro no começo de uma tarefa e esperar que suas subtarefas, ou *threads*, herdem essa configuração. Adicionalmente, múltiplos processos ou *threads* podem operar de modo concorrente com diferentes parâmetros ao longo do tempo de suas execuções, e não devem interferir um com o outro. Por ora, apenas um parâmetro foi apresentado: o que controla o tamanho do bloco. Entretanto, conforme será apresentado na seção 4.11, muitos outros podem ser adicionados à solução, e cada um teria um parâmetro associado a sua configuração.

Esse requisito exige uma estratégia para o gerenciamento desse conjunto de parâmetros. Ela deve ocorrer no nível das *threads* de execução, e ser realizada de modo transparente pelo cliente do SAD. Para este propósito, é mantida uma tabela de estado em memória. Nessa seção é descrito o fluxo sugerido para este gerenciamento durante a criação de arquivos e em alterações dos parâmetros, ilustrado na figura 4.7. Para essa descrição, será adotado o parâmetro de tamanho de bloco. Entretanto, o fluxo é genérico e atende a outros parâmetros que venham a ser adicionados, por exemplo, estendendo a solução conforme a discussão da sessão 4.11.

1. **Criação de arquivos.** Quando um novo arquivo é criado, o componente cliente precisa decidir qual tamanho de bloco será adotado para o novo arquivo. Este tamanho pode ter sido configurado previamente pela aplicação do usuário, por meio do arquivo “*i/control/blksize*”, ou ter assumido um valor padrão dependente da implementação.
 - Se a *thread* que gerou a operação já possui uma entrada na tabela de estado, o parâmetro é escolhido e o processo continua.
 - Se não há parâmetros associados àquela *thread*, a busca seguirá utilizando-se a *thread pai*, até que um dos critérios abaixo seja satisfeito:
 - Uma entrada para uma *thread pai* seja encontrada na tabela de estado
 - A próxima *thread pai* na hierarquia pertença a outro usuário
 - Caso se trate da primeira *thread* do sistema operacional

Figura 4.7: Fluxo proposto para manutenção dos parâmetros de criação de arquivos

Este procedimento assegura que *threads* recém iniciadas herdarão os parâmetros da *thread* pai, sem interferência das demais. Em qualquer um dos casos, se necessário, o cliente adiciona uma entrada na tabela para a thread que criou o arquivo, a fim de atender às próximas requisições. Neste caso, a entrada é também marcada como possuindo *parâmetros herdados*. A razão disso é que, quando uma *thread* pai altera seus parâmetros, todas as *threads herdeiras* que não alteraram explicitamente seu parâmetro precisam ser atualizadas. Este processo é descrito a seguir.

2. Mudança do parâmetro de tamanho de bloco. Quando a mudança do parâmetro é solicitada, o cliente efetua uma busca na tabela de estado procurando por sua entrada e varre todas as entradas relativas a *processos filhos*, isso é, *threads* ou processos iniciados por aquele que originou a alteração de parâmetro.

- Se não for encontrada nenhuma entrada de processos filho, apenas é executada a alteração do parâmetro na tabela do processo e a operação prossegue.
- Caso existam entradas de processos filhos, cada entrada que estiver marcada como herdada é removida. Isso garante que uma próxima operação deste processo herdará o novo parâmetro do pai.
- Se a entrada de processo filho não estiver marcada como herdada, significa que ela foi configurada explicitamente pelo processo ou *thread* em questão e, por esse fato, não deve ser alterada.

A busca continua até que todos os filhos sejam pesquisados na tabela de estado. Neste ponto, a entrada do processo é alterada na tabela, e a operação prossegue.

Listagem 4.3: Redefinindo o estado padrão para o processo do usuário, configurando o parâmetro de blocos para seu valor padrão.

```
1 $ echo 1 > .i/control/reset
```

4.10.1 Configuração para o valor padrão

O parâmetro de tamanho de bloco, ou qualquer outro parâmetro criado em novas funcionalidades, deve adotar algum valor predefinido pela implementação se não for configurado explicitamente pelo usuário. Alternativamente, após a gravação de um arquivo de controle, pode ser conveniente retornar à configuração de um estado inicial, de modo que o usuário não precise controlar quais parâmetros do estado foram alterados e precisam ser revertidos. Para uma funcionalidade deste tipo, bastaria gravar qualquer dado num arquivo de controle denominado “*.i/control/reset*”, conforme o exemplo da listagem 4.3.

4.11 EXTENSIBILIDADE DA PROPOSTA

A abordagem proposta apresenta extensibilidade, já que múltiplos arquivos podem ser definidos, controlando diversos aspectos da alocação de dados, e provendo informações de metadados. Nesta seção serão apresentados alguns parâmetros operacionais cujo controle é interessante e de que modo podem ser configurados por meio dos arquivos propostos. Os mesmos não foram implementados no presente trabalho, porém servem para demonstrar diversos aspectos adicionais que podem ser tratados com a proposta.

4.11.1 Controle da alocação dos blocos de dados

Para definir em que nós os blocos do próximo arquivo gerado serão gravados, uma interface “*.i/control/placement*” poderia receber a gravação de um conjunto de caracteres, como

- *fN*: todos os blocos serão gravados no nó de número N. Se omitido, assume-se o nó local.
- *rr*: blocos distribuídos uniformemente de modo rotativo (*round-robin*), iniciando pelo nó local.
- *rd*: blocos serão distribuídos de modo aleatório entre os nós da rede.
- *sA;B;C;D;...*: blocos distribuídos entre os nós A, B, C, D (ou mais) de modo rotativo.

Listagem 4.4: Definindo um único nó para a gravação de todos os blocos do próximo arquivo.

```

1 $ echo f3 > .i/control/placement
2 $ echo teste > teste.txt
3 $ cat .i/local/teste.txt
4 #size: 6
5 #blocks: 1
6 #block size: 67108864
7 #nodes: 1
8 #listing: node start end bytes
9 3 0 5

```

Figura 4.8: Blocos desalinhados com o tamanho dos registros do arquivo.

A listagem 4.4 demonstra como pode ser fixado o nó 3 para receber os blocos do próximo arquivo, e o resultado esperado para a consulta às informações de localidade. Na primeira linha é gravado o texto “f3” que instrui o *parfs* que o próximo arquivo deve ter seus blocos gravados no terceiro nó. Em seguida, na linha 2, um pequeno arquivo de texto é gravado e, em seguida, tem sua localidade consultada na terceira linha.

A saída da consulta, de acordo com o especificado na seção 4.6, demonstra que o único bloco se encontra armazenado no nó correspondente.

4.11.2 Blocos desalinhados com a estrutura

Muitas vezes, uma operação de leitura ou escrita ultrapassa os limites de um bloco local e isso torna necessário o acesso a um bloco remoto. Esta situação pode ocorrer porque o tamanho do registro dos arquivos da aplicação pode não estar em alinhamento com o tamanho dos blocos usados na quebra dos mesmos. A figura 4.8 ilustra uma situação deste tipo, em que o registro é ligeiramente maior do que a metade do tamanho de um bloco. Nesse exemplo, o acesso ao segundo registro abrange dois blocos. Se o segundo bloco estiver armazenado remotamente, o aproveitamento da localidade dos dados é reduzido. Um efeito imediato deste tipo de acesso é um aumento significativo no tempo total da operação, porque a latência de acesso à região remota do registro, correspondente ao bloco após a fronteira, tende a ser muito mais alto.

Listagem 4.5: Alinhamento do tamanho do bloco com o registro do próximo arquivo criado

```

1 $ cat .i/info/blksize
2 67108864
3 $ echo 2000 > .i/control/regsize
4 $ cat .i/info/blksize
5 67110000

```

Embora soluções como o GFS e o HDFS usem apenas blocos de tamanho fixo, o aumento da complexidade para se adotar esta funcionalidade de blocos variáveis é relativamente baixo, desde que todos os blocos de um mesmo arquivo utilizem este mesmo tamanho (exceto o último, que poderá ser menor em função do término dos dados). A seguir são propostas algumas abordagens para tratar dessas situações.

4.11.3 *Arquivos com registro de tamanho fixo*

Uma solução simples, por meio dos arquivos de controle, seria a de calcular um tamanho de bloco tal que a fronteira dos registros dos arquivos fique alinhada com o fim de um bloco. O tamanho dos registros de um arquivo pode ser definido ao se escrever no arquivo de controle “*i/control/regsize*” um número inteiro que indique tamanho em bytes de cada registro do próximo arquivo a ser criado. A escolha do tamanho do bloco será realizada de modo automático, buscando alinhar o tamanho do bloco ao múltiplo inteiro mais próximo do tamanho do registro.

A listagem 4.5 demonstra como definir o tamanho de 2000 bytes para um arquivo com registros de tamanhos fixos, a ser criado em seguida. Neste caso, o bloco padrão de 64 MB (67108864 bytes) será alterado para 67110000 bytes, com o intuito de manter os registros alinhados.

Seria possível definir também um controle para um cabeçalho de tamanho diferente, o que consiste em um refinamento deste controle.

4.11.4 *Registros de tamanhos variáveis: arquivos de texto*

Em muitas aplicações, os registros lógicos possuem tamanhos variáveis. Neste caso, não há como evitar de modo simples o acesso a dois blocos consecutivos durante o acesso a um registro único. Para atender a esta situação, seria necessário que os próprios blocos tivessem tamanhos variáveis para um mesmo arquivo, e também seria necessário uma função para identificar cada registro durante a gravação.

Para simplificar a implementação, uma abordagem para esse tipo de arquivo é proposta a seguir. Arquivos de texto possuem suas linhas delimitadas por um ou dois caracteres

Listagem 4.6: Definição do tipo texto para o próximo arquivo criado

```

1 $ echo u > .i/control/textfile
2 $ echo linha 1 > teste.txt && echo linha 2 >> teste.txt
3 $ cat .i/local/teste.txt
4 #size: 2
5 #blocks: 1
6 #block size: 67108864
7 #nodes: 1
8 #listing: node start end bytes
9 0 0 1

```

especiais de fim de linha. É um caso especial de um arquivo com registros de tamanho variável, delimitado por um conjunto de caracteres (um ou mais). Para este tipo de arquivo, o seguinte comportamento pode ser implementado com relativa simplicidade:

1. Ao definir o tipo do próximo arquivo como sendo um texto ou um conjunto de registro delimitados, o tamanho do bloco permanece o mesmo.
2. O cliente do sistema de arquivos distribuídos varrerá o próximo arquivo criado durante a gravação buscando onde começa e onde termina cada uma das linhas ou registros delimitados.
3. Estas informações serão divulgadas para a localidade dos dados no lugar dos bytes dos blocos quando acessado o arquivo de consulta descrito na seção 4.6. Assim, para um arquivo de texto, é listado como informação de localidade o número da primeira e da última linha que cada um dos bloco abriga.

Para se definir um arquivo como sendo de texto, é gravado no arquivo de controle “*i/control/textfile*” um caractere, “*u*” ou “*d*”, que indica um final de linha do tipo Unix (caractere CR) ou DOS (caracteres CR e LF). A listagem 4.6 demonstra como isso pode ser feito, e exibe o resultado da consulta de localidade para um arquivo com duas linhas.

4.11.5 *Registros de tamanhos variáveis: arquivos delimitados por sequência de caracteres*

A solução adotada para arquivos de texto pode ser generalizada para uma sequência arbitrária de delimitadores. No caso de registros delimitados por um ou mais caracteres, bastaria gravá-los em um arquivo denominado “*i/control/delimiter*”. A listagem 4.7 exibe o resultado das localidades para um arquivo de quatro registros, cujo delimitador é o caractere ponto-e-vírgula.

Listagem 4.7: Definição de sequência delimitadora de registros para o próximo arquivo criado

```

1 $ echo \; > .i/control/delimiter
2 $ echo reg1\;registro2\;regis3\;reg4 > teste.txt
3 $ cat .i/local/teste.txt
4 #size: 4
5 #blocks: 1
6 #block size: 67108864
7 #nodes: 1
8 #listing: node start end bytes
9 0 0 3

```

4.12 CONSIDERAÇÕES SOBRE A PROPOSTA

A solução proposta se assemelha à ideia de um sistema de arquivos de consulta, como o *procfs* do Unix [26], provendo ao usuário arquivos e diretórios que oferecem informações que não estariam facilmente acessíveis de outro modo. Combinada com uma implementação de um sistema de arquivos distribuídos onde é inserida, a solução favorece o uso da localidade dos dados em aplicações de processamento paralelo que demandem acesso intensivo a dados, no sentido de se obter alto desempenho.

Ao mesmo tempo, a proposta não impede o desenvolvimento de outros meios complementares para o acesso às funções de localidade, como bibliotecas ou chamadas de sistema. De fato, uma ou outra pode ser mais apropriada, dependendo da natureza da aplicação. Por exemplo, arquivos de texto são de uso trivial em linguagens de alto nível para *scripting* e *shell*, enquanto uma linguagem de nível mais baixo, como C, pode ser mais adequada ao uso de bibliotecas. Esses outros meios de interface podem coexistir com a interface de arquivos virtuais.

As próprias funções de consulta e controle podem ser enriquecidas com a adição de novos arquivos de controle na hierarquia proposta. Assim, a lista de possíveis outros controles da seção 4.11 não é exaustiva, e serve para demonstrar que a proposta oferece extensibilidade, e que pode ser programada para mais funções.

O gerenciamento do estado dos parâmetros associados aos processos que executam na máquina do usuário, pode ser implementado de mais de um modo. Uma estrutura em árvore oferece aderência ao modelo hierárquico de processos e *threads* do sistema operacional, porém, uma simples tabela pode ser suficiente, dependendo do número de processos que demandam o sistema ao mesmo tempo. Esse é um exemplo entre diversas decisões de projeto que surgem dos aspectos apresentados no presente capítulo. O detalhamento da arquitetura e da implementação, com as discussões relativas a essas decisões, será apresentado no capítulo que segue.

Do ponto de vista dos componentes servidores de dados e metadados, a adição dos arquivos e diretórios de consulta e controle não altera as características básicas da arquitetura proposta, já consagrada, em que se baseiam o GFS e o HDFS.

5 DETALHAMENTO DA IMPLEMENTAÇÃO

Nas próximas sessões será apresentada e discutida a implementação do *parfs*, seguindo as especificações da arquitetura estabelecida no capítulo anterior. O ponto de partida é uma breve discussão sobre algumas decisões de projeto. A relação dessas decisões com os objetivos do trabalho é usada para justificar a escolha das mesmas na implementação. Em seguida, os detalhes da realização de cada um dos componentes são apresentados. São detalhadas as funções, ferramentas e bibliotecas utilizadas, bem como apresentados os arquivos de configuração e parâmetros operacionais. Outros aspectos técnicos, limitações da implementação, e sugestões de melhoria, são discutidos conforme cada componente é introduzido.

5.1 ALGUMAS DECISÕES SOBRE AS CARACTERÍSTICAS DA IMPLEMENTAÇÃO

Para que a implementação oferecesse aderência aos objetivos do trabalho, algumas decisões sobre a arquitetura e o funcionamento dos componentes foram avaliadas e escolhidas. O critério empregado é o de implementar o necessário para demonstrar a viabilidade do mecanismo e interface de arquivos e diretórios de consulta e controle, simplificando o máximo possível a programação, porém preservando a extensibilidade da solução, de acordo com a seção 4.11.

5.1.1 *Serviço de metadados centralizado*

Seguindo o modelo do HDFS e do GFS, foi implementado um serviço centralizado de metadados. Uma vez que aplicações de acesso intensivo a dados tendem a operar em arquivos que apresentam quantidades massivas de dados, o número de operações de metadados tende a ser relativamente limitado, já que o tamanho dos blocos de dados, grande, se comparado aos blocos de sistemas de arquivos regulares, auxilia na redução da massa de metadados.

Adicionalmente, quando é realizada uma consulta de metadados de um arquivo, as informações sobre a localidade dos blocos podem ser armazenadas em um cache local no cliente. Assim, operações sequenciais sobre um mesmo arquivo deixam de exigir múltiplos acessos aos servidor de metadados para obter novamente seus atributos, reduzindo ainda mais a carga sobre o serviço.

O emprego de um único serviço centralizado de metadados reduz significativamente a complexidade do projeto, em especial sobre as semânticas de consistência, uma vez que toda a informação relevante se encontra neste componente. Não existe a necessidade de

coordenação entre diversas instâncias, como no caso de sistemas com metadados distribuídos. Entretanto, com essa abordagem, o servidor de metadados se torna um ponto único de falha, assumindo um papel crítico na infraestrutura, já que é o responsável por garantir a integridade e a disponibilidade de todo o sistema. O espelhamento do serviço de metadados, como ocorre no HDFS, é uma estratégia para oferecer maior confiabilidade na operação do sistema. No entanto, o objetivo de validar a proposta do trabalho não depende dessas características e, assim, a implementação conta com uma única instância do *parfsmds*.

5.1.2 *Modelo de consistência*

Uma implementação que garantisse a consistência global do sistema de arquivos seria desejável, pois permitiria que uma classe mais geral de aplicações pudesse ser beneficiada pelo sistema. O fato de ser usado um serviço único de metadados também favorece este tipo de implementação. Entretanto, um sistema que garanta a consistência precisaria efetuar uma sincronização completa antes de finalizar qualquer operação que modifique dados. Isto impõe um alto custo sobre o desempenho geral do sistema e, por isso, tanto o GFS quanto o HDFS preferem comprometer a flexibilidade da escrita, com o intuito de favorecer o desempenho das operações de leitura.

Desse modo, a implementação do *parfs* adota um modelo de imutabilidade de dados. Depois que um novo arquivo é fechado, após a gravação de seus dados, não pode mais ser modificado, apenas removido.

Embora essa limitação reduza significativamente os tipos de aplicação em que o sistema pode ser empregado, não compromete o objetivo do presente trabalho. O tratamento completo da consistência não é uma questão essencial para a avaliação da proposta. Ao mesmo tempo, constitui desafios significativos para a implementação. Esses fatos justificam a imutabilidade como uma abordagem prática para o modelo de consistência na presente versão do *parfs*.

Cabe notar que a arquitetura geral não impede que semânticas completas de consistência sejam desenvolvidas em implementações futuras. Os arquivos e diretórios de controle e consulta podem, inclusive, ser usados nessas implementações aprimoradas para se alternar entre diferentes estratégias de consistência, arquivo a arquivo, conforme a necessidade do usuário.

5.1.3 *Replicação de dados e metadados*

Qualquer abordagem de consistência que se baseie na expectativa de que o arquivo seja criado uma vez, lido muitas vezes, e nunca ou quase nunca modificado, facilita o

uso de réplicas, já que as semânticas de consistência e replicação se tornam mais fáceis de implementar. Isso inclui especialmente o modelo de imutabilidade, que certamente é o que recebe funcionalidades de replicação de modo mais simples. O serviço centralizado de metadados também favorece a implementação de uma estratégia de replicação dos mesmos.

Embora o uso de réplicas de dados e metadados seja desejável e traga vantagens também do ponto de vista de desempenho, já que permite dividir a carga de leituras concorrentes em um mesmo bloco, a complexidade da implementação aumentaria de modo significativo, pois novas interações entre componentes e controles de estado precisariam ser adicionadas à implementação. Uma vez que não favorecem diretamente o objetivo de avaliar a viabilidade da proposta, as funcionalidades de replicação dos blocos de dados e metadados não foram contempladas no escopo da implementação.

Isso significa que falhas no serviço de metadados paralisam o ambiente, e que a perda dos dados de um dos servidores ocasiona perda de arquivos. Isso implica em delegar a responsabilidade pela disponibilidade para as camadas mais inferiores da infraestrutura, e assumir a falha como uma exceção. Essa característica é comum em implementações de SAP [14], conforme visto na seção 2.10.

5.2 SERVIDOR DE DADOS *parfsds*

O papel fundamental do *parfsds* é permitir acesso aos blocos de dados por meio de uma conexão de rede. As operações de leitura e escrita são todas demandadas pelos múltiplos componentes clientes, que permanecem conectados por meio de *sockets* TCP. Por essa razão, o *parfsds* foi implementado como um componente basicamente passivo da arquitetura. Isso significa que ele aguarda os comandos e os responde, não iniciando nenhum processo por conta própria.

Também não há estado associado às operações do servidor de dados. As chamadas são simples funções do tipo *pergunta-resposta*, o que torna o desenvolvimento do componente relativamente simples.

A seguir são apresentadas as operações implementadas e os parâmetros associados a cada uma. Em seguida, será descrito de que modo o componente é configurado e como se dá o armazenamento dos blocos nos discos locais. Ao final da seção, são discutidos outros aspectos técnicos relevantes da implementação.

5.2.1 Operações do servidor de dados

As operações implementadas no *parfsds* estão sumarizadas na tabela 5.1. Os parâmetros enviados pelos clientes nas operações de dados podem incluir:

Tabela 5.1: Sumário das funções implementadas no *parfsds*.

COMANDO	PARÂMETROS	RETORNO
Inicia conexão	Nenhum	Sucesso Falha
Lê dados	Bloco, Deslocamento Tamanho	Sucesso, dados parciais Sucesso, dados completos Falha
Escreve dados	Bloco, Deslocamento, Tamanho, Dados	Sucesso, bytes gravados Falha
Apaga bloco	Bloco	Sucesso Falha
Obtém espaço	Nenhum	Tamanho livre / ocupado

- O **comando** para identificar a operação desejada;
- O **bloco de dados** associado à operação;
- O **deslocamento** (*offset*) em bytes a partir do qual a operação deve ocorrer no bloco;
- O **tamanho** em bytes dos dados a serem lidos ou escritos;
- Os **dados**, no caso de operações de escrita.

O retorno é apenas uma indicação de sucesso ou falha. No caso de sucesso, traz informações sobre o tamanho efetivo (também em bytes) da operação de leitura, seus dados, ou o tamanho do espaço disponível neste servidor de dados. A razão para o tamanho efetivo de uma leitura poder ser menor do que o solicitado é que os clientes podem solicitar leituras que ultrapassem o tamanho do bloco, em particular devido a algoritmos de leitura antecipada (*prefetching* ou *read-ahead*).

Uma escrita especificada com um tamanho diferente dos dados recebidos gera uma situação de erro. Caso sejam recebidos mais dados, apenas os dados correspondentes ao tamanho especificado para a operação são utilizados, e o resto é enviado para a decodificação do protocolo de mensagens como se a operação tivesse sido finalizada. Como os dados não devem fazer sentido como comandos do *parfsds*, vão sendo descartados até que uma mensagem com um comando válido seja identificada. O recebimento de menos dados faz com que o *parfsds* fique aguardando a chegada de mais dados no *socket*, bloqueando a operação.

Isso é capaz de gerar uma cenário de inconsistência, com gravações parciais, e deve ser objeto de avaliação em trabalhos futuros. Isso ocorre porque na presente implementação

Listagem 5.1: Exemplo de configuração para o *parfsds*.

```

1 [
2   {parfsds, [
3     {port, 4000},
4     {node_id, 0},
5     {local_fs, /opt/parfs/n0}
6   ]}
7 ].

```

os dados que chegam do cliente para gravação são enviados diretamente para o arquivo de bloco de dados.

A operação que obtém os espaços livre e ocupado no *parfsds* é utilizada pelos clientes, em todos os servidores de dados, para se obter a soma total de espaço disponível e ocupado no SAD. Evidentemente, se o sistema de arquivo onde os blocos são armazenados forem empregados para outra finalidade, a informação de espaço ocupado ficará inconsistente, pois existirão arquivos não relacionados, cuja ocupação será contabilizada. Essa questão não foi tratada na implementação atual. Sugere-se, em trabalhos futuros, mover o controle do espaço ocupado para o servidor de metadados para evitar esta situação.

5.2.2 Parâmetros de configuração e armazenamento dos blocos

O *parfsds* está associado a um arquivo de texto com seus parâmetros de configuração, indicados na listagem 5.1. Neste exemplo, o identificador do servidor de dados é *0*, que utiliza a porta TCP de número *4000* para a comunicação com os clientes, e armazena seus blocos no diretório local */opt/parfs/n0*. Os blocos são gravados como arquivos regulares neste diretório, com a nomenclatura *xN*, onde *N* é o número inteiro que o identifica unicamente o bloco no serviço de metadados do SAD.

Uma vez que as configurações operacionais do *parfsds* podem ser alteradas, é possível executar múltiplas instâncias do programa em um mesmo servidor. Uma razão imediata para isso seria o uso de múltiplos discos físicos para o armazenamento dos blocos. Entretanto, uma alternativa mais elegante, que pode ser utilizada em trabalhos futuros, é o de permitir múltiplas entradas do parâmetro *local_fs*, realizando a distribuição dos blocos nos múltiplos pontos de montagem especificados.

5.2.3 Outros aspectos da implementação do *parfsds*

Os únicos parâmetros que precisam ser gerenciados pela implementação estão associados ao andamento de uma operação solicitada. Isso inclui o controle de entrada e saída de dados nos *sockets*, determinando quantos bytes foram lidos/escritos naquela conexão e

quantos estão pendentes, o controle do protocolo para o recebimento e envio de mensagens, e o conjunto de conexões ativas.

Dada a sua finalidade, o *parfsds* deve realizar de modo bastante eficiente o transporte de dados entre conexões de redes e os blocos de dados, armazenados no disco local do nó. A implementação deve, portanto, oferecer boa tolerância à concorrência de operações e evitar cópias de dados entre *buffers*. Linguagens do tipo funcional facilitam o desenvolvimento de sistemas concorrentes. Assim, o protótipo do *parfsds*, originalmente em linguagem C, foi reescrito em *Erlang* para a implementação atual. A função *sendfile* do SO Linux é empregada para permitir o envio de dados a partir do bloco diretamente para o *socket* do cliente.

De acordo com a figura 4.1, apenas o componente cliente estabelece comunicação com o servidor de dados. Isso ocorre porque a demanda de todas as operações com blocos é realizada por ele, não havendo comunicação entre o *parfsds* e o serviço de metadados. No caso de futuras versões que implementem replicação, será necessária uma interação entre os serviços. Essa interação pode ser direta, com o servidor de metadados coordenando a geração de réplicas, ou por meio de um novo componente que gerencie essas operações.

5.3 SERVIDOR DE METADADOS *parfsmds*

Assim como o *parfsds*, o servidor de metadados *parfsmds* recebe múltiplas conexões provenientes dos clientes. Entretanto, o papel assumido pelo *parfsmds* é o de uma interface de acesso a um banco de dados que abriga todos os metadados relevantes para a operação do sistema. Por essa razão, a massa de dados envolvida nas operações em seus *sockets* é menor.

Apesar disso, todas as operações no sistema de arquivos distribuídos somente podem ser realizadas depois que o cliente obtém os metadados relevantes. Por exemplo, o início de uma operação de leitura em um arquivo somente ocorre depois que o cliente obtém a lista de blocos e dos nós correspondentes. Assim, o tempo de resposta das consultas para o *parfsmds* é um fator de desempenho relevante.

O conjunto de metadados gerenciado pela implementação do *parfsmds* é apresentado na próxima seção. Em seguida, serão indicadas as operações implementadas e descrito, de modo sumário, o funcionamento de cada uma. A seção finaliza tratando sobre possíveis melhorias na atual implementação.

5.3.1 Atributos e mapeamento de blocos

O armazenamento dos metadados gerenciados pelo *parfsmds* é realizado em um banco de dados relacional, baseado no software *SQLite* [27]. O *SQLite* é uma biblioteca que se

integra às aplicações e provê funções completas para o gerenciamento de um banco de dados, armazenado em um único arquivo no disco local do servidor.

Com o número de *inode* como chave primária, uma tabela é dedicada ao controle do espaço de nomes. Outra é empregada no controle dos atributos de permissão, usuário e grupo donos do arquivo ou diretório, bem como dos tempos de criação, acesso e modificação. O tamanho do arquivo também é um atributo deste grupo de metadados. O mapeamento dos arquivos para os blocos de dados também se dá a partir do número do *inode*, e é realizado em uma terceira tabela. Essa tabela de controle associa os números que identificam blocos de dados existentes aos servidores de dados que os abrigam e, ao mesmo tempo, ao *inode* do arquivo a que pertencem, e à ordem de cada bloco na sequência de dados.

A tabela 5.2 apresenta as informações de metadados ora discutidas, agrupadas de acordo com a finalidade de suas informações. Os tipos das informações são dados numéricos (N), um caractere (C), valores booleanos (B) ou texto (T).

5.3.2 Operações com arquivos e diretórios

A seguir é descrito o funcionamento das operações implementadas no *parfsmds*. A tabela 5.3 sumariza os comandos, seus parâmetros e possíveis retornos.

- *Consultas*: existem três operações de consulta a metadados. Elas consistem em obter a lista dos arquivos e subdiretórios de um *inode* correspondente a um diretório, obter os atributos de um *inode* específico e sua lista de blocos, ou então procurar por um nome em um diretório especificado por um *inode*.
- *Criação de arquivos ou diretórios*: quando um arquivo ou diretório é criado, é inserido no espaço de nomes a entrada associado ao mesmo. A única diferença nesta primeira etapa é que o campo *modo* possui um *bit* que identifica se o objeto é um arquivo regular ou um diretório. Isso se dá pelos comandos *mkdir* e *create*. A criação de um arquivo, entretanto, seguirá para a alocação de novos blocos de dados quando o processo de escrita começar. Inicialmente, ele possui um tamanho igual a zero.
- *Atualização*: a troca de nomes ocorre por meio da função *rename*, que funciona de modo igual para arquivos e diretórios. Os parâmetros indicam os *inodes* pais novos e antigos, e nomes novos e antigos. Podem ser alterados um ou ambos os atributos. Atualizações de permissão e tempos ocorrem por meio da função *setattr*.
- *Deleção*: finalmente, a deleção ocorre de modo distinto, pois diretórios não podem ser apagados se possuírem *filhos*. Já no caso dos arquivos, basta marcá-los como apagados por meio do campo *apagado* da tabela 5.2 e o cliente prossegue com a deleção

Tabela 5.2: Registros de metadados do *parfsmds*.

FINALIDADE	METADADO	TIPO	DESCRIÇÃO
Controle de blocos	inode	N	<i>inode</i> a que o bloco pertence
	bloco	N	Número do bloco no SAD
	ds	N	Servidor de dados que o abriga
	seq	N	Sequência do bloco no arquivo
	size	N	Tamanho do bloco de dados
Controle de atributos	inode	N	<i>inode</i> da estrutura
	em_uso	B	Indica se travado para uso
	apagado	B	Indica se foi apagado
	modo	N	Permissões
	uid	N	Usuário dono do <i>inode</i>
	gid	N	Grupo dono do <i>inode</i>
	atime	N	Data e hora de acesso
	mtime	N	Data e hora de modificação
	ctime	N	Data e hora de criação
	size	N	Tamanho do <i>inode</i>
Espaço de nomes	inode	N	<i>inode</i> do nome
	pai	N	<i>inode</i> pai na árvore
	nome	T	Nome do arquivo ou diretório

Tabela 5.3: Sumário das funções implementadas no *parfsmds*.

COMANDO	PARÂMETROS	RETORNO
inode	inode	Sucesso, atributos, lista de blocos Falha
lookup	inode pai, nome	Sucesso, atributos, lista de blocos Falha
readdir	inode	Sucesso, listagem do diretório Falha
rename	inode pai, nome, inode novo pai, novo nome	Sucesso Falha
setattr	inode, modo	Sucesso / Falha
rmdir	inode pai, nome	Sucesso / Falha
unlink	inode pai, nome	Sucesso / Falha
create	inode pai, nome, modo	Sucesso, inode / Falha
mkdir	inode pai, nome, modo	Sucesso, inode / Falha
newblock	inode, ds	Sucesso, bloco / Falha
closeblock	bloco, size estrutura, primeira, última (opcionais)	Sucesso Falha
closefile	inode	Sucesso / Falha

dos blocos de dados. Por simplicidade, na implementação atual, os metadados não são expurgados.

- *Operação com blocos:* O cliente pode solicitar novos blocos de dados para um arquivo recém criado por meio do comando *newblock*. Depois que um bloco inteiro é gravado, a operação *closeblock* atualiza os parâmetros do bloco. Quando a gravação finaliza, a função *closefile* retira a indicação *em_uso* e o arquivo fica disponível para leitura pelos clientes do *parfs*. Esse parâmetro é adicionado na gravação do primeiro bloco, e depois que é retirado, operações de escrita para o arquivo são negadas.

5.3.3 Comentários sobre a implementação do *parfsmds*

O processo de criação de um novo arquivo, partindo da criação de uma nova entrada no espaço de nomes, até o fechamento com a função *closefile* é capaz de gerar inconsistência

no conjunto de metadados se o cliente desconectar antes de finalizá-lo. Entretanto, o uso do parâmetro *em_uso*, mais os atributos de data de criação, permitem rastrear arquivos perdidos por meio de um processo de varredura periódico. Os blocos de dados gravados também podem ser liberados por este processo.

O *parfsmds* também é implementado na linguagem *Erlang* como um servidor sem estado associado a seus comandos. A única configuração que o usuário deve especificar é a porta TCP em que aguarda conexões do componente cliente.

5.4 CLIENTE *parfscli*

O *parfscli* é o componente que provê ao usuário a visão de um sistema de arquivos comum, com os atributos de transparência de rede e localidade, conforme a seção 2.4. Isso é possível porque o *parfscli* coordena de modo automático o acesso às operações dos módulos de dados e metadados.

Sua implementação é baseada em uma arquitetura de funções de retorno, denominadas *callbacks*. O *parfscli* aguarda em um laço principal até que uma notificação de um de seus módulos seja identificada. Quando a origem da notificação se torna disponível, uma das funções de *callback* predefinidas é acionada para proceder com a operação.

Os eventos de notificação são provenientes do conjunto de *sockets* para os servidores de dados e do núcleo do sistema operacional. Condições que disparam esses eventos incluem operações no SAD solicitadas pelo SO, eventos nos *sockets*, tais como conexões fechadas, fim da transmissão de dados, novos dados disponíveis para leitura, etc. Também são recebidas notificações de fechamento de programas no SO. Esse último tipo de evento é empregado no gerenciamento de estado, descrito na seção 4.10. O *socket* para o serviço de metadados é empregado quando atributos dos arquivos, incluindo os mapeamentos para blocos, se tornam necessários para atender às operações.

As operações de leitura e escrita evitam cópias em memória, para movimentação da massa de dados, por meio da passagem de estruturas de referência. Em uma operação de leitura, existe um *buffer* associado ao *socket* com o servidor de dados que atende a requisição. Quando este *buffer* recebe os dados da leitura, os mesmos não são copiados para o *buffer* de saída para devolver o dado ao usuário. O que ocorre é a passagem de uma estrutura com ponteiros de referência para as áreas de memória onde os dados originais se encontram no *buffer* de entrada. Essa técnica é muitas vezes denominada *zero-copy*, e consiste em operações onde o processador não precisa efetuar cópias de dados de uma área de memória para outra.

Uma vez que todas as operações são assíncronas e cópias de dados são evitadas, o programa é implementado com um único laço principal e não opera com *threads* ou multi processos. Já que a implementação utiliza bibliotecas e se comunica com o *kernel* do SO, a plataforma empregada para o uso no *parfs* deve ser compatível. Em particular,

Listagem 5.2: Exemplo de configuração para o *parfscli*.

```

1 mds 192.168.0.1 8000
2 ds 0 192.168.0.1 4000
3 ds 1 192.168.0.2 4000
4 ds 2 192.168.0.3 4000
5 ds 3 192.168.0.4 4000
6 sockets 2
7 node_id 0
8 local_fs /opt/parfs/n0

```

o *parfscli* é escrito em linguagem C e requer o sistema operacional Linux. As próximas seções descrevem como é configurado, de que modo as solicitações do sistema operacional são tratadas, e como ocorre a comunicação entre o *parfscli* e os demais componentes. Ao final são discutidos outros aspectos relevantes da implementação.

5.4.1 Parâmetros de configuração

Toda a comunicação entre os clientes, servidores de dados, e metadados, é realizada por meio do protocolo TCP. Por essa razão, o *parfscli* deve ser configurado para estabelecer conexões a esses serviços. A listagem 5.2 apresenta um exemplo para o arquivo de configuração.

O único servidor de metadados *parfsmds* deve ser especificado com o parâmetro *mds*, seguido do endereço da máquina que o abriga e a porta TCP na qual o serviço atende. O cliente estabelecerá uma conexão de controle por onde as operações de metadados e sincronização serão tratadas.

Em seguida, todos os servidores de dados devem ser especificados, um por linha, por meio do parâmetro *ds*, seguido do identificador único do servidor de dados no SAD e a porta TCP do serviço. Podem ser abertas múltiplas conexões entre o cliente e os servidores de dados, o que permite múltiplas operações de leitura e escrita em paralelo entre um cliente e um mesmo servidor de dados, em acordo com a seção 5.2. O número de conexões que serão abertas para cada servidor de dados é indicado por meio da chave *sockets*.

Finalmente, é especificado o identificador do nó local por meio do parâmetro *node_id*. Isto serve para o cliente acessar diretamente os blocos de dados armazenados localmente a partir do sistema de arquivos local indicado em *local_fs*, que é compartilhado com o *parfsds*.

5.4.2 Tratamento das requisições do sistema operacional

Toda a comunicação entre o *parfscli* e o SO também ocorre por meio de um *socket*. Do ponto de vista do programador, as semânticas são iguais a uma conexão de rede: é estabelecida uma sessão, ocorrem notificações quando chegam dados, e esses podem ser enviados para um *buffer* de saída. A diferença é que a comunicação deve ser coordenada por meio de funções de uma biblioteca, a qual o *parfscli* é ligado, denominada *libfuse*.

Cada requisição identificada pela *libfuse* cria uma estrutura contendo os parâmetros da operação. Uma vez que o processamento ocorre por meio de uma fila de eventos assíncrona, essa estrutura precisa ser conhecida pela função *callback* em que ocorre o término da chamada aos servidores de dados, ou ao servidor de metadados. Isso ocorre por meio de um ponteiro de referência, em uma estrutura global de estado, onde ficam registradas todas as operações solicitadas que estão pendentes.

A estrutura gerada pela *libfuse* para cada requisição contém um contador numérico que a identifica unicamente. Por esse fato, múltiplas operações podem ser tratadas de modo concorrente pelo *parfscli*, estabelecendo-se um esquema de *pipeline*.

Infelizmente, devido a características do SO, operações únicas de leitura são limitadas a 128 Kb e isso diminui os ganhos de desempenho providos pelo *pipeline* para um único programa em execução, pois o SO não faz novas solicitações de leitura até que os dados previamente solicitados cheguem. Outros programas, entretanto, podem mandar suas próprias solicitações enquanto o primeiro aguarda seus dados.

Nesse cenário, a latência no acesso aos dados assume um importante papel no desempenho das operações de leitura e escrita. No próximo capítulo os efeitos desse aspecto serão apresentados para o caso de leituras locais.

5.4.3 Operações do componente cliente

As operações no componente cliente aparentam finalidades muito semelhantes àquelas do servidor *parfsmnds*, apresentadas na seção 5.3. Entretanto, a implementação das mesmas é significativamente diferente, porque também envolvem a coordenação com os servidores de dados. Além disso, os arquivos e diretórios de consulta e controle são gerenciados pelo *parfscli* e não existem no conjunto de metadados, o que aumenta ainda mais esta distinção.

- *Listagem de diretórios*: quando é requisitada a listagem de um diretório, o programa precisa decidir se é um diretório real, ou de consulta ou controle. A decisão ocorre devido ao resultado da operação com as máscaras nos *inodes*, conforme seção 4.9.2. Nos casos de diretórios reais ou de consulta, é realizada uma chamada para a função *readdir* do servidor de metadados, o que retorna a listagem correspondente. Se for um diretório de consulta, os *inodes* são devidamente mascarados antes de que a lista

seja devolvida ao FUSE. Se for um diretório de controle, os arquivos de controle são inseridos na listagem pelo programa cliente e a listagem é devolvida ao FUSE como resposta à requisição.

- *Consulta e atualização de atributos:* A consulta e atualização de atributos demandadas pelo usuário consiste em interagir com os registros do *inode* no servidor de metadados. Isso somente ocorre quando se trata de arquivos ou diretórios reais.
- *Leitura de dados:* Com a consulta de atributos retornando o mapeamento de blocos de arquivos reais, as operações são despachadas para a fila de acordo com os blocos e servidores de dados. No caso de arquivos de consulta, a localidade é retornada. Arquivos de controle, ao serem lidos, devolvem o valor corrente para cada parâmetro.
- *Escrita de dados:* A escrita em um bloco preexistente ocorre enviando os dados ao servidor de dados correspondente. Ao se atingir a fronteira de um bloco, um novo número de bloco é demandado para o servidor de metadados. Arquivos de consulta não podem ser gravados, enquanto que os de controle alteram os parâmetros de acordo com o fluxo da seção 4.10.
- *Abertura de arquivo:* As aberturas de arquivos reais em modo de escrita somente são permitidas se o arquivo estiver marcado como “*em_uso*” no *parfsmds*. As demais operações de abertura para operações de leitura são autorizadas.
- *Criação de arquivos e diretórios:* A criação de novos arquivos e diretórios reais é apenas uma operação de metadado. A gravação dos dados, propriamente dita, é considerada independente. Nessa função, os atributos são recebidos, e o cliente instrui o servidor de metadados a adicionar o registro em seu banco.
- *Remoção de arquivos:* A remoção de um arquivo se inicia obtendo a lista de blocos. Em seguida, é marcado no servidor de metadados o atributo de arquivo apagado. Finalmente, todos os blocos são removidos quando o cliente comanda todos os servidores de dados que abrigam os blocos. Um diretório não possui dados, então é apenas uma operação enviada ao servidor de metadados.
- *Estatísticas do SAD:* São consultados todos os servidores de dados, obtendo o resultado da função que retorna os espaços livre e em uso. A soma desses valores é retornada ao usuário.

Cabe ressaltar que essas operações são tratadas pela implementação de modo transparente para o usuário. Este, efetivamente, fará uso das funções comuns do sistema operacional para a manipulação de arquivos, como se fosse qualquer outro sistema de arquivos, local ou distribuído.

5.4.4 Gerenciamento da tabela de parâmetros

Cada operação recebida pelo *parfscli* por meio do FUSE está associada a uma estrutura de contexto denominada *fuse_ctx*, que abriga as seguintes informações:

- *uid*: é um número que identifica unicamente o usuário que executa o programa que originou a requisição no SO.
- *gid*: o número que identifica o grupo de processo no SO. Este parâmetro não é empregado no fluxo de controle de estado apresentado na seção 4.10.
- *pid*: é um número denominado *thread id* que identifica unicamente qual *thread* em execução que demandou a operação.

Com isso é possível realizar o fluxo de controle de estado para os parâmetros, conforme proposto na seção 4.7. Uma lista ligada é utilizada como estrutura para armazenar os parâmetros para cada *thread id* que realiza operações nos clientes do SAD. Essa lista é percorrida quando novos arquivos são criados, buscando pelos parâmetros. Se nada for encontrado, de acordo com o fluxo, os valores padrões predefinidos na implementação são utilizados.

Para gerenciar a tabela de parâmetros, o cliente também emprega um *socket* com a interface *NETLINK* do *kernel* do Linux para receber eventos de finalização de *threads* e processos. Isso permite que o *parfscli* impeça que a tabela de estado cresça indefinidamente. Infelizmente, o *netlink* requer execução do componente cliente com privilégios administrativos (*root*).

5.4.5 Acesso direto a dados locais

Uma vez que a abertura do arquivo do bloco de dados possui algum custo, a estrutura do arquivo relativa a um bloco recém acessado é mantida aberta em memória, uma vez que, em função do próprio princípio de localidade, existe uma chance do mesmo bloco ser demandado novamente nas próximas operações.

Como os dados são imutáveis na presente implementação, não surgem problemas de inconsistência devido ao acesso concorrente entre o *parfsds* e o *parfscli* aos blocos de dados. Os efeitos desse desvio para o sistema de arquivos local sobre o desempenho são discutidos no capítulo 6.

A possibilidade de reescrever arquivos previamente criados é uma funcionalidade comum e ampliaria a abrangência do sistema. Trabalhos futuros podem estabelecer uma estratégia de tratamento da consistência e, então, remover a restrição de imutabilidade de dados, permitindo alterar arquivos existentes.

5.5 CONSIDERAÇÕES SOBRE OS ASPECTOS DE IMPLEMENTAÇÃO

Neste capítulo foram apresentados os detalhes da implementação de um sistema de arquivos distribuídos, construído com o objetivo de realizar a proposta dos arquivos e diretórios virtuais. A maior parte da complexidade associada às operações ficou no âmbito do componente cliente. Esta conduta é desejável, pois se fosse o servidor de metadados que tratasse dos arquivos virtuais, haveria um possível fator de limitação para a escalabilidade, conforme o número de clientes aumentasse.

A solução de máscaras introduz um tratamento bastante eficiente, totalmente baseado em memória, e independente de consultas adicionais aos metadados, para diferenciar entre arquivos/diretórios reais e de consulta e controle. Como o número de *inodes* permitidos é elevado, mesmo em ambientes de 32 bits, em que é da ordem de bilhões, o mascaramento do *inode* oferece grande extensibilidade, permitindo a definição de múltiplos pontos de consulta e controle, sem comprometer a capacidade do SAD de abrigar um número elevado de arquivos em sua estrutura.

O gerenciamento do estado associado aos parâmetros operacionais permitiu manter a transparência das operações para os processos e *threads* de aplicações. Essas também contam com o nível de isolamento adequado para que um outro programa não interfira nos parâmetros operacionais de outro. Entretanto, esse gerenciamento somente é possível interagindo com estruturas de nível relativamente baixo, como o recebimento de eventos a partir do *kernel*, e a manutenção de tabelas de parâmetros do SAD que devem ser usados para cada *threads* ou processo do sistema operacional. Cabe, em trabalhos futuros, avaliar meios alternativos para este gerenciamento. Por exemplo, caso seja utilizada uma biblioteca para o acesso ao SAD, a manutenção do estado pode ser elevada para o nível da aplicação.

A escolha das linguagens de programação está associada às características do serviço. Os servidores de dados e metadados são, basicamente, serviços de consulta e comandos, sem estados, que devem oferecer grande capacidade de concorrência. Ao mesmo tempo, não é esperado que realizem operações de alto uso de processador, pois cópias de dados entre *buffers* são evitadas. Por essa razão, uma linguagem funcional como *Erlang* é apropriada. A atual versão dos componentes *parfsds* e *parfsmds* são reimplementações nas quais se observou uma redução significativa no número de linhas de código. Já o componente cliente requer o uso de múltiplas bibliotecas, o gerenciamento de estruturas de memória em nível relativamente baixo, e operações que demandam processamento. Por essa razão, a linguagem C foi mantida para este componente.

6 TESTES REALIZADOS

Este capítulo apresenta os testes realizados na implementação do *parfs*. Inicialmente é apresentado um descritivo sobre o ambiente em que os testes foram conduzidos. São indicados os servidores adotados e suas respectivas capacidades de disco, memória, e processamento. Uma análise inicial do desempenho direto no acesso a dados nos discos locais, e da capacidade de transmissão de rede, serve para estabelecer uma linha de base de desempenho dos servidores, sem o uso do *parfs*. Esses resultados serão empregados em comparações com os resultados obtidos nos testes realizados com o *parfs*.

A discussão está dividida em três partes. Na primeira, ocorre a **caracterização do ambiente**, com o estabelecimento da linha de base de desempenho. Ainda nesta parte, é analisado o desempenho do serviço de metadados, que é centralizado. A discussão dos resultados demonstra que a implementação de um único componente *parfsmds* é adequada para a avaliação da proposta.

Em seguida, são discutidas **estratégias de movimentação de dados** entre o componente cliente *parfscli* e o servidor de dados *parfsds*. Uma vez que o bloco em operação está localmente armazenado, são analisados os efeitos de diferentes meios de acesso aos dados: passando pelo *socket* e interagindo com o *parfsds* local, ou abrindo diretamente o arquivo de bloco de dados. Os efeitos sobre o desempenho das operações de leitura, e a latência resultante, são apresentados nesta parte do capítulo.

A terceira parte consiste na **avaliação da proposta** de uso dos arquivos de consulta para oferecer ganhos de desempenho por meio da localidade de dados. Uma discussão prévia sobre o comportamento do desempenho no acesso a um arquivo distribuído, dependendo da região que é lida ser local ou não, demonstra que o seu uso é relevante no ambiente de testes.

Finalmente, são avaliadas as taxas de transferência de operações de acesso a dados, realizadas com e sem o uso da localidade de dados, na tentativa de identificar os ganhos de desempenho esperados. O intuito dos testes realizados é avaliar a viabilidade da proposta para oferecer ganhos de desempenho em aplicações que trabalhem com acesso intensivo a dados.

6.1 DESCRIÇÃO DO AMBIENTE DE TESTES

Os benefícios de desempenho ao se levar a computação para perto dos dados se tornam mais significativos se a capacidade agregada de transferência de discos for maior do que a banda disponível nas conexões de rede. Se os discos fossem muito mais lentos do que

Figura 6.1: Visão simplificada do ambiente de testes

a rede, apenas a latência seria um fator de vantagem no uso da localidade, conforme a discussão exposta na seção 3.3.

Por essa razão, foi empregado um pequeno *cluster* composto por 6 máquinas virtuais (MV) com discos do tipo SSD (*solid state drive*) configurados em um volume RAID (*redundant array of inexpensive disks*). Esse tipo de configuração provê altas taxas de transferência. As máquinas virtuais foram criadas em máquinas físicas distintas. Em contrapartida, as interfaces de rede eram todas limitadas a cerca de 1 Gbps, o que foi confirmado durante os testes que seguem. Esse cenário estabelece um *cluster* onde a soma das capacidades de transferências dos discos é maior do que a capacidade de banda na interconexão dos nós, o que favorece o uso da localidade de dados.

Essas máquinas virtuais foram contratadas de um serviço de *computação em nuvem* (*cloud computing*). Isso introduz uma flutuação nos níveis de desempenho, pois o ambiente é compartilhado. Esse aspecto foi observado durante o estabelecimento da linha de base, que apresentou desvio padrão elevado nos valores médios. A tabela 6.1 apresenta os detalhes da configuração das máquinas virtuais, os resultados da linha de base, e especifica os detalhes do arquivo empregado nos testes. Infelizmente não são fornecidas especificações detalhadas do servidor físico que abriga as máquinas virtuais. Por essa razão, tentou-se estabelecer a linha de base de desempenho de rede e disco, conforme apresentado na próxima seção.

A figura 6.1 apresenta uma visão simplificada do ambiente de testes, os servidores envolvidos e os componentes instalados em cada um.

6.1.1 Linha de base de desempenho

Uma vez que os testes foram realizados em um ambiente de *cloud computing* com recursos compartilhados, o primeiro passo dos testes foi estabelecer uma linha de base para o desempenho das transferências de acesso aos discos locais, e a banda de comunicação entre os nós. Uma vez que todos os testes se baseiam em operações com uso intensivo de dados, sem processamento, não foram realizados *benchmarks* para avaliar a capacidade de processamento das máquinas virtuais.

Tabela 6.1: Detalhes da configuração dos servidores virtuais e do arquivo de testes

Especificações das máquinas virtuais e desempenho	
Número de <i>cores</i> virtuais	2 (limitados a 2000 MHz, Intel 6-Core)
RAM	2 GB
Tamanho do disco e tipo	40 GB armazenados em RAID com discos SSD
Sistema operacional	CentOS 6.4 (64 bit)
Versão do <i>kernel</i>	2.6.32
Taxa média da transf. de disco	(569 $\pm$ 123) MB/s
Taxa média da transf. de rede	(921 $\pm$ 3) Mb/s $\sim$ 115.1 MB/s
Parâmetros do arquivo de testes e configuração do <i>cluster</i>	
Tamanho do arquivo de testes	100 GB
Tamanho do bloco	256 MB
Número de nodes	5 <i>parfscli</i> / <i>parfsds</i> , 1 <i>parfsm</i> <i>ds</i> /NFS

Foi empregada a ferramenta *iperf* para medir as taxas de transferência de rede entre todas as máquinas. Foram executados 100 ensaios de um minuto. A média e o desvio padrão são apresentados na tabela 6.1, e está de acordo com o uso de interfaces de rede da ordem de 1 Gbps.

Em seguida, foi realizada uma série de operações de leituras sequenciais em um arquivo de 4 GB, armazenado no sistema local de cada máquina virtual. Em cada execução, o *cache* do sistema de arquivos foi expurgado, para que a medição refletisse a capacidade nativa dos discos. Foram executados 50 testes, em que o tempo de transferência foi medido em cada um dos nós. Como não é incomum em ambientes de *cloud* [23], foi observada uma grande flutuação nas taxas de acesso ao disco, o que resultou no alto desvio padrão da média apresentada na tabela 6.1. Também foram observados diferentes níveis de desempenho de disco entre as máquinas virtuais, o que contribuiu ainda mais para o desvio padrão.

Mesmo com a grande flutuação nos resultados, as taxas médias de transferências indicam que o ambiente é plenamente aceitável para se validar o uso da localidade de dados, pois os discos se mantiveram consistentemente mais rápidos do que a capacidade de transferência das interfaces de rede, que apresentam desempenho até 5 vezes menor do que os discos locais.

6.2 CARACTERIZAÇÃO DO DESEMPENHO DO SERVIDOR DE METADADOS

O servidor centralizado de metadados precisa responder rapidamente às solicitações dos clientes, pois as operações de leitura e escrita somente podem prosseguir com os atributos e mapeamento de blocos dos arquivos. O *parfsm**ds* também deve oferecer tolerância à concorrência, já que múltiplas instâncias do componente cliente estarão com *sockets* abertos

realizando consultas ao mesmo tempo. Além do tempo de resposta, o comportamento do serviço sob alta carga também é relevante. Assim, foram avaliados dois cenários de testes:

- *com saturação*: o comportamento do ambiente é avaliado na máxima capacidade de atendimento. Novas conexões vão sendo adicionadas, e o comportamento do número de operações realizadas, e do tempo de atendimento, são medidos;
- *sem saturação*: espera-se que o tempo de resposta mantenha-se relativamente baixo quando o sistema não se encontra sob alta carga.

6.2.1 *parfsmds: cenário com saturação*

Para avaliar como as operações de metadados associadas aos arquivos de consulta de localidade se comportam, foram realizados testes de desempenho com múltiplas conexões TCP, enviando requisições continuamente. Isso caracteriza o teste com uma carga de saturação, em que as medidas são tomadas sempre na máxima capacidade de atendimento possível.

As conexões foram disparadas a partir de uma única máquina virtual por meio de um programa cliente que abre uma thread para cada conexão. O servidor de metadados se achava instalado em outra máquina, para que a latência de rede fizesse parte da latência total das operações. Foi variado o número de *sockets* de 1 a 128, e medidos dois parâmetros, que são intrinsecamente ligados:

- Média e desvio padrão da quantidade de requisições atendidas por segundo pelo servidor de metadados, e
- Média e desvio padrão da latência de cada requisição.

O gráfico da figura 6.2 apresenta os resultados para três funções empregadas no mecanismo que provê os arquivos de consulta de localidade:

- *readdir*: listagem de um diretório real contendo 10 arquivos e subdiretórios;
- *lookup*: pesquisa por um nome de arquivo em um diretório contendo 10 arquivos e subdiretórios.
- *inode*: listagem de atributos e blocos de dados de um arquivo com 7 blocos.

6.2.2 *Capacidade de atendimento*

A complexidade das operações realizadas pelo *parfsmds* afeta diretamente o seu tempo de processamento e, conseqüentemente, a quantidade de requisições a que o servidor é capaz de atender por unidade de tempo.

Figura 6.2: Número de requisições atendidas por segundo em função do número de conexões, para as operações *readdir*, *lookup* e *inode*. Cenário com saturação.

A função *readdir* é claramente de relativa complexidade computacional, pois envolve varrer parte do espaço de nomes e, ao mesmo tempo, obter os atributos dos arquivos e diretórios encontrados. A função *inode* consiste em obter os atributos e a listagem de blocos para um arquivo específico. É o resultado de duas consultas de metadados indicadas na tabela 5.2. A operação *lookup* é semelhante à *inode*, porém ocorre que o arquivo é procurado por seu *nome* no diretório cujo *inode* é especificado. Por essa razão, seu desempenho é ligeiramente inferior ao da função *inode*.

Os valores das funções *lookup* e *inode* apresentaram valores consideráveis para o desvio padrão. Isso está associado à concorrência na obtenção do acesso ao banco de dados, que não segue a ordem de chegada das requisições. Mesmo assim, as taxas médias para a capacidade de atendimento assumiram valores bastante interessantes para um ambiente onde o número de operações de metadados deve ser limitado, conforme a discussão da seção 5.1.1.

Adicionalmente, pode-se dizer que um aspecto tão relevante quanto a capacidade de atendimento é o comportamento do sistema conforme o aumento da concorrência. Nos três casos observados na figura 6.2, a partir de um número de conexões, ocorre uma saturação, sem aparente tendência de degradação de desempenho conforme mais conexões vão sendo adicionadas. Essas características se devem ao eficiente gerenciamento de concorrência da linguagem *Erlang*, adotada no *parfsmds*. Uma vez que o conjunto de metadados cabe na memória, o gargalo é basicamente de processamento. O emprego de MV mais poderosas, com mais núcleos de processamento, deve favorecer o desempenho do servidor de metadados.

Figura 6.3: Latência das operações *readdir*, *lookup* e *inode* em função do número de conexões. Cenário com saturação.

6.2.3 Latência das operações

A latência da operação está diretamente associada ao nível de concorrência. Como se trata de um teste de saturação, o efeito de se adicionar mais conexões ao servidor *parfsmds*, requisitando continuamente suas operações, resulta, no melhor caso, em um aumento linear da latência.

Esse fenômeno pode ser claramente observado no gráfico da figura 6.3, e está de acordo com a discussão apresentada no item anterior: o comportamento do servidor, conforme aumenta a concorrência, se mantém regular. Apenas ocorre um aumento, basicamente linear, dos tempos médios das operações.

Esse comportamento sugere que, mesmo com um grande número de conexões, se não houver saturação do processador do servidor de metadados, as operações tendem a se manter com baixa latência.

6.2.4 *parfsmds*: cenário sem saturação

Para confirmar a proposição de baixa latência mesmo com múltiplas conexões ao *parfsmds*, o programa de testes foi modificado para introduzir uma latência, gerada randomicamente, com média de um segundo entre requisições. Com esse comportamento induzido, é simulado um cenário com múltiplos clientes conectados, porém, que solicitam as funções do servidor de metadados de modo esporádico. O comportamento da latência para a função *lookup* é apresentado no gráfico da figura 6.4, o que confirma a expectativa de baixa latência quando o ambiente executa fora de sua zona de saturação.

Figura 6.4: Latência da operação *lookup* em função do número de conexões. Cenário sem saturação.

6.2.5 Comentários sobre a caracterização do *parfsmds*

O fato do GFS e do HDFS serem amplamente empregados em aplicações reais indica que a escolha de utilizar um serviço centralizado de metadados não é um fator de limitação de desempenho, desde que este seja bem implementado. Com efeito, os testes realizados demonstram que o servidor de metadados *parfsmds*, mesmo sendo um componente único do *cluster*, realiza operações simples o suficiente para oferecer bom desempenho, devendo atender aos usuários com baixa latência, da ordem de poucos milissegundos, desde que seja mantido fora de uma situação de saturação. Entretanto, mesmo sob saturação, o comportamento quase linear da latência indica que a implementação é razoavelmente eficiente, oferecendo escalabilidade.

6.3 ESTRATÉGIAS PARA MOVIMENTAÇÃO DOS DADOS NO COMPONENTE CLIENTE

A avaliação de desempenho do componente cliente depende da interação do mesmo com o componente de dados. Isso porque o cliente deve movimentar a massa de dados correspondente ao conteúdo dos arquivos, que é, por sua vez, obtido a partir dos blocos servidos pelo *parfsds*. Entretanto, fatores como a latência e banda de rede estão intrinsecamente ligados ao desempenho dessas operações, o que torna a caracterização do desempenho do *parfscli* mais complexa.

Existem dois procedimentos que podem ser adotados para minimizar as interferências citadas:

Figura 6.5: Desempenho para operações de leitura, releitura e leitura randômica nos casos de uso de *socket* e acesso direto aos blocos de dados.

- Utilizar o cache de arquivos do sistema operacional nos blocos que serão lidos durante a avaliação, para mitigar limitações de banda dos discos locais. Isso pode ser feito realizando leituras prévias nos arquivos de blocos de dados;
- Realizar os testes de comunicação via *socket* de rede por meio das interfaces de *loopback*, o que elimina a limitação de banda da interface e a latência do meio físico da rede.

Entretanto, conforme a discussão realizada na seção 4.4, o componente cliente pode melhorar o acesso a dados locais, desviando de toda a complexidade associada ao *socket*, acessando os dados correspondentes ao servidor de dados local diretamente. Os testes a seguir foram realizados para avaliar os efeitos dessa otimização, que é realizada automaticamente pelo componente cliente.

A figura 6.5 apresenta as taxas de transferências obtidas quando se utilizou o *socket* para obter os dados a partir do *parfsds* e quando o acesso ocorreu diretamente no arquivo de bloco de dados. O desempenho da operação do caso local é até uma ordem de grandeza maior do que o caso em que os dados precisam trafegar desde o servidor de dados até o cliente. Mesmo as operações randômicas com tamanhos de leitura menores, onde a massa de dados trafegada é menor, são favorecidas.

O fato das operações de leitura e releitura apresentarem valores compatíveis nos dois casos é resultado o uso do cache para eliminar o disco local como um potencial gargalo que poderia afetar os resultados.

Figura 6.6: Latência das operações de leitura com blocos de 4 Kb, nos casos de uso de *socket* e acesso direto aos blocos de dados.

A redução da latência das operações é o segundo efeito desse desvio. O componente cliente é capaz de realizar a leitura antecipada de blocos em até 128 Kb. O efeito é que operações com solicitações de dados de pequeno tamanho, como 4 Kb, devem apresentar latências muito menores depois que uma primeira solicitação é realizada. Esse efeito pode ser observado nos gráficos da figura 6.6, onde também se caracteriza uma significativa redução na latência, devida ao acesso direto aos blocos.

As leituras antecipadas são demandadas pelo núcleo do sistema operacional. Eventualmente, duas operações de leitura podem ser enviadas para o *parfscli*, e o efeito é que a operação seguinte apresenta latência muito menor. Este comportamento pode ser observado na figura 6.7, em que as operações de leitura ocorreram com tamanho de 128 Kb. No caso do acesso local, a latência menor não favoreceu o uso da leitura antecipada, de modo que as operações apresentaram aproximadamente a mesma latência.

Os resultados demonstram que a estratégia de acessar os blocos diretamente oferece redução de latência, e aumento na capacidade de transferência do componente cliente. Isso melhora a sua capacidade de prover ganhos de desempenho às aplicações que utilizam as informações de localidade. A avaliação dessa capacidade é tratada na próxima seção.

6.4 AVALIAÇÃO DA PROPOSTA: BLOCOS DE DADOS LOCAIS E REMOTOS

Quando as taxas de transferência de disco são maiores do que a banda de interconexão disponível, operações de leitura que envolvam blocos em múltiplos nós devem exibir taxas

Figura 6.7: Latência das operações de leitura com blocos de 128 Kb, nos casos de uso de *socket* e acesso direto aos blocos de dados.

de transferência limitadas pela capacidade de banda da rede, alternando com taxas mais elevadas quando são lidas as regiões do arquivo correspondentes aos blocos armazenados localmente. Este comportamento ocorre porque nessas regiões o sistema movimentava dados internamente, sem as limitações de desempenho das interfaces de rede e da latência associada à transmissão na rede.

Para avaliar este comportamento, a taxa de leitura do arquivo de teste foi medida a cada 100 ms no primeiro nó do cluster, que também executa o servidor de dados. A figura 6.8 apresenta a taxa de transferência como função da posição de leitura nos primeiros 8 GB do arquivo. As regiões do arquivo que consomem dados dos blocos locais apresentam taxas de transferências maiores do que em regiões com blocos remotos.

Esse comportamento corrobora a estratégia de uso das informações de localidade a partir dos arquivos de consulta para obter taxas de leitura com alto desempenho. Os testes apresentados na próximas seções obtêm ganhos de desempenho a partir da distribuição das leituras de acordo com as informações de localidade.

6.5 TESTES COM OS ARQUIVOS DE CONSULTA DE LOCALIDADE

Para calcular a taxa de transferência para múltiplos servidores, foram executados testes de leitura partindo de uma máquina a até 5 servidores. A taxa média de transferência foi calculada a partir das medidas de 50 ensaios. Os testes foram executados com a ferramenta

Figura 6.8: Aumentos na taxa de transferência devido a leitura de regiões do arquivo com blocos armazenados localmente. Áreas sombreadas indicam as regiões locais.

Unix *dd*, com operações em blocos de 128 Kb. Essas configurações dão ao teste um caráter de leitura sequencial.

Os testes foram organizados do seguinte modo:

- *Caso sem localidade*: o primeiro conjunto de testes foi executado sem o uso do arquivo de consulta às informações de localidade. A expectativa é que o desempenho ficasse limitado pela capacidade da rede, possivelmente se degradando conforme a concorrência aumentasse. O objetivo desses testes é possibilitar a comparação com o caso de uso da localidade de dados. O servidor de metadados *parfsmds* também recebeu a instalação de um servidor Network File System (NFS) para fins de comparação com essa solução. Neste cenário, o arquivo de testes foi igualmente dividido entre os nós para que todos os dados fossem cobertos. Com uma única máquina, o arquivo inteiro é lido a partir de um único nó do sistema. Duas máquinas dividem as metades do arquivo, e assim por diante, até que as 5 máquinas tratassem, cada uma, 1/5 do arquivo.
- *Caso com localidade*: o segundo teste consistiu no uso da informação de localidade provida pelo arquivo de consulta. Com o conhecimento sobre as regiões locais correspondentes a cada máquina, as leituras foram restritas a essas regiões para garantir que apenas dados locais fossem acessados. Assim, a leitura do arquivo completo somente ocorreu quando o teste atingiu 5 máquinas.

Figura 6.9: Resultados das taxas de transferências agregadas dos testes com localidade e sem localidade

6.5.1 Resultados e discussão

A figura 6.9 ilustra as taxas de leitura calculadas para os casos que usam e não usam a localidade. Cada barra vertical do gráfico representa a média da taxa leitura de uma máquina. As barras são empilhadas para denotar a taxa de transferência agregada.

Os resultados para uma única máquina são compatíveis com as velocidades de transferência observadas na figura 6.8, que alterna de 435 MB/s em leituras locais para cerca de 94 MB/s em blocos remotos. Conforme nós vão sendo adicionados, o caso com localidade evolui quase linearmente, enquanto o caso sem localidade apresenta-se severamente limitado.

É possível observar na figura 6.9 que, no caso sem localidade, a adição de mais máquinas parece reduzir a taxa média de transferência de cada máquina individual. Esse comportamento pode ser justificado pela seguinte análise:

- Com uma única máquina, 1/5 dos nós são lidos localmente. Isso resulta em uma média de leituras locais de 20% do total de blocos do arquivo;
- Quando uma segunda máquina é adicionada, cada máquina lê metade do arquivo. A metade não lida contém blocos que poderiam ser lidos localmente, mas que agora são tratados por outra máquina. Neste caso, em particular, 10% dos blocos poderão ser lidos localmente;
- Conforme mais máquinas vão sendo adicionadas, a quantidade de blocos lidos localmente é ainda mais reduzida.

Tabela 6.2: Quantidade de dados lidos em cada máquina e *speedups* médios como função do número de nós.

Número de nós	1	2	3	4	5
Bytes lidos por nó, sem localidade	100G	50G	33G	25G	20G
<i>Speedup</i> sem localidade	1	1,5	1,7	2,09	2,3
<i>Speedup</i> com localidade	1	1,7	2,4	3,4	4,4
Relação do <i>speedup</i> com/sem localidade	4,4	5,3	6,7	7,9	9,0
<i>Speedup</i> do NFS	1	1,06	1,04	0,94	0,94

Esse comportamento justifica a queda de desempenho observado nas barras individuais. No caso sem localidade, conforme as máquinas vão sendo somadas, a altura de cada uma das barras empilhadas diminui, embora o valor agregado aumente.

A escala dos histogramas da figura 6.9 demonstra ganhos significativos quando se comparam os casos. Os *speedups* das taxas médias de leitura dependendo do número de nós são apresentados na tabela 6.2, que inclui a quantidade média de dados lida em cada execução.

A taxa de transferência por meio do NFS foi medida no servidor e sua média é apresentada para comparação com o caso sem localidade, já que a taxa de transferência não aumentou conforme os demais clientes eram adicionados. A relação entre os *speedups* dos cenários com e sem localidade denotam os ganhos de desempenho obtidos devido ao uso da localidade de dados a partir dos arquivos de consulta.

6.6 COMPARAÇÃO DE DESEMPENHO COM HDFS

Com o intuito de comparar o comportamento das taxas de transferência do *parfs* com o HDFS, foi realizada a instalação do Hadoop versão 2.3.0 e foram realizados testes de leitura. Para uma comparação em condições compatíveis, o HDFS foi instalado em um mesmo número de máquinas virtuais, com as mesmas configurações empregadas nos testes anteriores. Além disto, o HDFS foi montado como um sistema de arquivos regular por meio de um conector FUSE, o *hadoop-fuse-dfs*.

Entretanto, quando o HDFS é montado como um sistema de arquivos regular, as informações de localidade não podem ser acessadas pelo ponto de montagem. Isso só é possível se o usuário também associar ao seu programa a biblioteca do HDFS ou utilizar alguma ferramenta que o faça. Desse modo, os testes foram realizados com dois cenários, a saber:

1. Foram criados dois arquivos idênticos. Por meio de uma ferramenta de administração denominada *hdfs*, os blocos de dados foram movimentados ao longo do *cluster* para que o primeiro arquivo tivesse todos os seus blocos armazenados localmente (no nó de teste), e o segundo arquivo fosse armazenado remotamente (nenhum bloco local).

Tabela 6.3: Taxas médias de transferências de blocos locais e remotos obtidas com o HDFS.

Localização dos blocos	Locais	Remotos
Taxa média de transferência	(223 ± 7) MB/s	$(80,4 \pm 1,3)$ MB/s
<i>Speedup</i> médio	2,8	1

2. A partir de um dos nós, foram realizadas 50 medidas de desempenho de leitura nos arquivos local e remoto, e extraídas as médias e os desvios-padrão.

O meio de identificação da localidade dos blocos é apresentado no Apêndice B, em que também se encontram listados os arquivos de configuração do *cluster* HDFS.

Uma vez que o objetivo é comparar o desempenho das implementações, os testes de leitura foram realizados com arquivos de menos de 2 GB de tamanho, de modo que todos os blocos de dados pudessem ser lidos diretamente do *cache* do sistema operacional. Isso é análogo ao que foi realizado na seção 6.3. Os resultados estão apresentados na tabela 6.3.

Quando o HDFS é montado por meio do FUSE, não é aproveitado o acesso diretamente aos blocos de dados como faz o *parfs*, conforme discussão da seção 6.3. Por essa razão, as taxas de leitura para os blocos locais ficam compatíveis com os valores obtidos com o *parfs* sem a otimização, conforme gráfico da figura 6.5. O *parfs*, no caso do acesso a dados locais, apresenta maior desempenho do que o HDFS quando este é montado com o FUSE. As taxas de transferência associadas ao tráfego de rede, quando blocos remotos são lidos, ficam próximas, uma vez que o principal fator limitante é a rede.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho propôs a combinação de um sistema de arquivos distribuídos com um mecanismo e interface para o acesso a informações de localidade e controle da alocação de dados. A abordagem baseada nos arquivos e diretórios de consulta e controle oferece uma solução prática para obter ganhos de desempenho para aplicações que realizam acesso intensivo a dados.

Isso é possível porque, por meio desses arquivos e diretórios de consulta, o programador é capaz de levar a computação para perto dos dados, resultando em taxas mais altas de transferência, e latência reduzida. Os ensaios demonstraram que, com o emprego das informações de localidade obtidas por meio dos arquivos de consulta, ganhos da ordem de até 9 vezes no desempenho das leituras foram possíveis, no ambiente de testes.

A solução é particularmente interessante nos casos em que não é possível modelar as aplicações como um fluxo do MapReduce, o que impede o programador de aproveitar o uso integrado da localidade em ambientes como o Hadoop.

A proposta também apresenta extensibilidade, permitindo o desenvolvimento de novos controles de parâmetros e arquivos para a consulta de informações de metadados, com alguns exemplos foram apresentados na seção 4.11.

Uma vez que a avaliação da proposta requeria um sistema de arquivos distribuídos simplificado, implementando o mecanismo de arquivos e diretórios de consulta e controle, foram adotadas algumas decisões de projeto que reduzem o conjunto de programas que podem se favorecer do sistema. Em particular, a capacidade de alterar os dados dos arquivos é um dos aspectos que merece ser tratado por trabalhos futuros. Outros aspectos que podem ser desenvolvidos em trabalhos futuros incluem:

- O desenvolvimento de uma solução de replicação. Por meio dos arquivos de controle, seria possível controlar o nível de redundância e distribuição de arquivos e diretórios individualmente;
- Explorar outros tipos de interface para a consulta e controle. Por exemplo, programas de nível mais baixo, escrito em linguagens como C, devem se favorecer de uma biblioteca que possam utilizar no lugar dos arquivos de texto. Cabe ressaltar que o desenvolvimento de uma biblioteca não impede o uso dos arquivos. De fato, as interfaces poderiam coexistir e ser escolhidas conforme a conveniência do usuário;
- Associar o *parfs* a uma ambiente de programação paralela e distribuída. Por exemplo, o NPFS apresenta um mecanismo para o disparo de processos nos servidores em que executa. Alguns trabalhos relacionados, apresentados no capítulo 3, criaram camadas de compatibilidade para associar seus sistemas ao Hadoop, no lugar do HDFS.

REFERÊNCIAS

- [1] GUARDIA, H. *Considerações Sobre as Estratégias de um Sistema de Arquivos Paralelos Integrado ao Processamento Distribuído*. Tese (Doutorado) — Departamento de Computação e Sistemas Digitais, Universidade de São Paulo, São Paulo, 1999.
- [2] GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The google file system. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 37, n. 5, p. 29–43, out. 2003. ISSN 0163-5980. Disponível em: <<http://doi.acm.org/10.1145/1165389.945450>>.
- [3] FILESYSTEM in Userspace. 2013. Disponível em: <<http://fuse.sourceforge.net/>>.
- [4] THANH, T. D. et al. A taxonomy and survey on distributed file systems. In: *Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management - Volume 01*. Washington, DC, USA: IEEE Computer Society, 2008. (NCM '08), p. 144–149. ISBN 978-0-7695-3322-3-01. Disponível em: <<http://dx.doi.org/10.1109/NCM.2008.162>>.
- [5] MIKAMI, S.; OHTA, K.; TATEBE, O. Using the gfarm file system as a posix compatible storage platform for hadoop mapreduce applications. In: *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2011. (GRID '11), p. 181–189. ISBN 978-0-7695-4572-1. Disponível em: <<http://dx.doi.org/10.1109/Grid.2011.31>>.
- [6] SHVACHKO, K. et al. The hadoop distributed file system. In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. Washington, DC, USA: IEEE Computer Society, 2010. (MSST '10), p. 1–10. ISBN 978-1-4244-7152-2. Disponível em: <<http://dx.doi.org/10.1109/MSST.2010.5496972>>.
- [7] HILDEBRAND, D.; HONEYMAN, P. Exporting storage systems in a scalable manner with pnfs. In: IEEE. *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE/13th NASA Goddard Conference on*. [S.l.], 2005. p. 18–27.
- [8] CHOWDHURY, M. et al. Managing data transfers in computer clusters with orchestra. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 41, n. 4, p. 98–109, ago. 2011. ISSN 0146-4833. Disponível em: <<http://doi.acm.org/10.1145/2043164.2018448>>.
- [9] DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, ACM, New York, NY, USA, v. 51, n. 1, p. 107–113, jan. 2008. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/1327452.1327492>>.

- [10] WHITE, T. *Hadoop: The Definitive Guide*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2009. ISBN 0596521979, 9780596521974.
- [11] CALLAGHAN, B.; PAWLOWSKI, B.; STAUBACH, P. *NFS Version 3 Protocol Specification*. United States: RFC Editor, 1995.
- [12] SATYANARAYANAN, M. A survey of distributed file systems. In: *Annual Review of Computer Science*. [S.l.: s.n.], 1989.
- [13] DENNING, P. J. The locality principle. *Commun. ACM*, ACM, New York, NY, USA, v. 48, n. 7, p. 19–24, jul. 2005. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/1070838.1070856>>.
- [14] TANTISIRIROJ, W.; PATIL, S.; GIBSON, G. *Data-intensive file systems for Internet services: A rose by any other name ...* Pittsburgh, PA, 2008.
- [15] ANANTHANARAYANAN, G. et al. Disk-locality in datacenter computing considered irrelevant. In: USENIX ASSOCIATION. *Proceedings of the 13th USENIX conference on Hot topics in operating systems*. [S.l.], 2011. p. 12–12.
- [16] PORTER, G. Decoupling storage and computation in hadoop with superdatanodes. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 44, n. 2, p. 41–46, abr. 2010. ISSN 0163-5980. Disponível em: <<http://doi.acm.org/10.1145/1773912.1773923>>.
- [17] BARROSO, L. A.; HÖLZLE, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, Morgan & Claypool Publishers, v. 4, n. 1, p. 1–108, 2009.
- [18] GREENBERG, A. et al. Vl2: a scalable and flexible data center network. In: ACM. *ACM SIGCOMM Computer Communication Review*. [S.l.], 2009. v. 39, n. 4, p. 51–62.
- [19] GRAY, J. Distributed computing economics. *Queue*, ACM, New York, NY, USA, v. 6, n. 3, p. 63–68, maio 2008. ISSN 1542-7730. Disponível em: <<http://doi.acm.org/10.1145/1394127.1394131>>.
- [20] KAUSHIK, R.; BHANDARKAR, M.; NAHRSTEDT, K. Evaluation and analysis of greenhdfs: A self-adaptive, energy-conserving variant of the hadoop distributed file system. In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. [S.l.: s.n.], 2010. p. 274–287.
- [21] ZAHARIA, M. et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: *Proceedings of the 5th European conference on Computer systems*. New York, NY, USA: ACM, 2010. (EuroSys '10), p. 265–278. ISBN 978-1-60558-577-2. Disponível em: <<http://doi.acm.org/10.1145/1755913.1755940>>.

- [22] XIE, J. et al. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In: *IPDPS Workshops*. [S.l.: s.n.], 2010. p. 1–9.
- [23] LEE, G. *Resource Allocation and Scheduling in Heterogeneous Cloud Environments*. Tese (Doutorado) — EECS Department, University of California, Berkeley, May 2012. Disponível em: <<http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-78.html>>.
- [24] KLEIMAN, S. R. et al. Vnodes: An architecture for multiple file system types in sun unix. In: *USENIX Summer*. [S.l.: s.n.], 1986. v. 86, p. 238–247.
- [25] RAJGARHIA, A.; GEHANI, A. Performance and extension of user space file systems. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2010. (SAC '10), p. 206–213. ISBN 978-1-60558-639-7. Disponível em: <<http://doi.acm.org/10.1145/1774088.1774130>>.
- [26] SCHROCK, E. *A brief history of /proc*. 2004. Disponível em: <https://blogs.oracle.com/eschrock/entry/the_power_of_proc>.
- [27] OWENS, M. *The definitive guide to SQLite*. [S.l.]: Apress, 2006.

A USO DOS ARQUIVOS DE LOCALIDADE

Neste apêndice é apresentado um detalhamento sobre a utilização dos arquivos de consulta para a realização dos testes de desempenho apresentados no capítulo 6. São indicados exemplos dos comandos utilizados nos *shells* para iniciar os programas que realizam as operações de leitura. A primeira seção consiste em um exemplo para filtrar as saídas de localidade para que cada máquina do ambiente identifique suas regiões locais. Em seguida, são apresentados os parâmetros empregados nos testes para métricas de latência banda de acordo com o tamanho do bloco da operação, apresentados na seção 6.3.

A.1 OBTENDO REGIÕES DE DADOS LOCAIS

A utilização do programa *grep* na saída apresentada na listagem 4.1 permite filtrar as regiões locais do arquivo. No caso do nó de número zero, o comando apresentado na listagem A.1 a seguir exibe apenas as linhas correspondentes a este nó.

Essa saída pode ser utilizada em um laço que comanda a leitura dessas regiões. Isso foi realizado para os testes de leitura com perfil sequencial, de acordo com a listagem A.2, descrita a seguir.

O programa *dd* realiza a leitura de dados a partir do arquivo *teste*. Uma vez que o tamanho do bloco de dados é fixo, interessa ao laço apenas o parâmetro de deslocamento (*offset*). Para tanto, o comando da listagem A.2 foi substituído pelo programa *awk*, que imprime apenas a coluna desejada, para as linhas que começam como número zero. Esta saída é então utilizada como variável de laço que executa o comando *dd* sucessivamente, com os deslocamentos apropriados para que sejam atingidas as regiões locais.

No exemplo da listagem A.2, a saída do programa *dd* é novamente processada pelo programa *awk* para imprimir o tempo (em segundos) para a cópia das regiões locais.

Listagem A.1: Filtrando regiões locais de um arquivo *teste* a partir do arquivo de consulta.

```

1
2 $ ls -l teste
3 -rwxr-xr-x 1 root root 2560376832 Jul 16 2013 teste
4
5 $ grep ^0 .i/local/teste
6 0 0 67108863
7 0 335544320 402653183
8 0 671088640 738197503
9 0 1006632960 1073741823
10 (continuação, suprimida)

```

Listagem A.2: Laço para leituras locais a partir do arquivo de consulta.

```

1
2 $ for offset in $(awk '/^0/ {print $2/1024/1024}' .i/
   local/teste); \
3 > do \
4 > dd if=teste of=/dev/null bs=1M count=64 skip=$offset; \
5 > done 2>&1 | \
6 > awk '/copied/ {print $6}'
7 0.0462019
8 0.0459676
9 0.0628573
10 0.0456152
11 0.0625863
12 0.0548945
13 0.0455458
14 0.0687876

```

A.2 EMPREGO DO PROGRAMA *iozone*

Para a análise das estratégias de movimentação de dados, apresentada na seção 6.3, foi empregado o programa de *benchmark iozone*. Os testes realizados foram os seguintes: cujos resultados médios foram apresentados nos gráficos das figuras 5.4.5, 6.6, e 6.7.

1. Desempenho de leitura, releitura e leitura randômica (opção “-i 1 -i 2”) na região relativa a um primeiro bloco, armazenado localmente. O modo automático (“-a”) repete a operação variando o tamanho da operação de leitura de 4 KB a 16 MB.
2. Latência para a operação de leitura e leitura randômica (opção “-Q”), em operações de leitura com tamanhos de 4 KB e 128 Kb (“-r 4k” e “-r 128k”).

Comandos e saídas de exemplo são apresentados na listagem A.3 a seguir. Na primeira parte é exibida a saída do teste de transferência. No exemplo, um bloco de 64 MB é lido. A segunda coluna indica o tamanho de cada operação de leitura, seguida das taxas de transferência obtidas.

Em seguida, é indicado o comando empregado no cálculo da latência das operações para blocos de 128 Kb. Os dados são gravados em um arquivo denominado “*rol.dat*”, cujas primeiras linhas são apresentadas ao final da listagem.

Listagem A.3: Desempenho de leitura e releitura em função do tamanho da operação.

```

1
2 $ iotzone -+E -i 1 -i 2 -f ./teste -s 64m -a
3 (saída editada)
4
5         KB      reclen          read      reread      random
6 65536         4      801908      799019      190626
7 65536         8      926949      928750      308718
8 65536        16      971608      973618      476350
9 65536        32     1009073     1009228      648440
10 65536        64     1612159     1606092      943255
11 65536       128     1629097     1613361     1130897
12 65536       256     1613938     1600042     1386973
13 65536       512     1624490     1556229     1458724
14 65536      1024     1480927     1479939     1400882
15 65536      2048     1498915     1479149     1424549
16 65536      4096     1517005     1518158     1439880
17 65536      8192     1565926     1545162     1430235
18 65536     16384     1749124     1746575     1399184
19
20 $ iotzone -+E -i 1 -f ./teste -s 64m -a -r 128k -Q
21 (saída suprimida)
22
23 $ cat rol.dat
24 Offset in Kbytes      Latency in microseconds      Transfer size
25         0           26          131072
26        128          110          131072
27        256          110          131072
28        384          109          131072
29        512          115          131072
30        640          120          131072
31        768          117          131072
32 (continuação suprimida)

```

B CONFIGURAÇÕES DO HDFS

Neste apêndice são apresentados os parâmetros de configuração adotados no *cluster* HDFS empregado nos testes da seção 6.6. Foi instalada a versão 2.3.0 nos mesmos servidores que abrigaram a instalação do *parfs*, executando a *Máquina Virtual Java* versão 1.7.0 *Update* 51. Uma vez que este ambiente apresenta um consumo de memória mais elevado do que o *parfs*, os sistemas foram executados um de cada vez.

Nas seções seguintes são apresentados os detalhes sobre a identificação da localidade dos blocos dos arquivos de testes e, em seguida, são apresentadas as listagens dos principais arquivos de configuração do *cluster*.

B.1 LOCALIDADE DOS BLOCOS NO HDFS

Para viabilizar a comparação apresentada na seção 6.6, foi utilizado o comando *fsck* do programa de administração *hdfs*. Com os parâmetros “*-blocks -locations*” é exibida a distribuição dos blocos e a listagem dos nós que abrigam suas réplicas.

A listagem B.1 apresenta uma listagem parcial da saída do programa, o que permitiu a confirmação de que o arquivo *saida.txt* se achava distribuído apenas em nós remotos, e que o arquivo *saida2.txt* continha todos os seus blocos armazenados localmente. O IP do *datanode* é exibido entre colchetes nas listagens dos blocos.

B.2 ARQUIVOS DE CONFIGURAÇÃO

As listagens B.2, B.3, e B.4, apresentam os arquivos de configuração do ambiente.

Listagem B.1: Identificação da localidade dos blocos de dados no HDFS.

```

1 $ ./hdfs fsck / -files -blocks -locations (saída editada)
2 /saida.txt 1849942016 bytes, 7 block(s): OK
3 0. BP-(...) len=268435456 repl=1 [10.128.236.231:8029]
4 1. BP-(...) len=268435456 repl=1 [10.128.236.231:8029]
5 2. BP-(...) len=268435456 repl=1 [10.128.35.18:8029]
6 (continuação, suprimida)
7
8 /saida2.txt 1849942016 bytes, 7 block(s): OK
9 0. BP-(...) len=268435456 repl=1 [10.128.235.28:8029]
10 1. BP-(...) len=268435456 repl=1 [10.128.235.28:8029]
11 2. BP-(...) len=268435456 repl=1 [10.128.235.28:8029]
12 (continuação, suprimida)

```


Listagem B.2: Arquivo de configuração global do HDFS: *hdfs-site.xml*.

```

1 <configuration>
2   <property>
3     <name>dfs.namenode.name.dir</name>
4     <value>/hdfs/name</value>
5     <final>true</final>
6   </property>
7   <property>
8     <name>dfs.datanode.data.dir</name>
9     <value>/hdfs/data1</value>
10    <final>true</final>
11  </property>
12  <property>
13    <name>dfs.replication</name>
14    <value>1</value>
15  </property>
16  <property>
17    <name>dfs.blocksize</name>
18    <value>268435456</value>
19  </property>
20 </configuration>

```

Listagem B.3: Arquivo de configuração global do Hadoop: *core-site.xml*.

```

1 <configuration>
2   <property>
3     <name>fs.defaultFS</name>
4     <value>hdfs://ds0/</value>
5     <final>true</final>
6   </property>
7   <property>
8     <name>io.file.buffer.size</name>
9     <value>131072</value>
10  </property>
11 </configuration>

```

Listagem B.4: Arquivo de configuração dos *datanodes* do HDFS: *slaves*.

```

1 ds0
2 ds1
3 ds2
4 ds3
5 ds4

```