ERICO C. DA SILVA

Exploração da Localidade dos Dados e Locks Distribuídos para Leitura e Escrita em um Sistema de Arquivos para Big Data ou Computação Científica

São Paulo
2024

ERICO C. DA SILVA

Exploração da Localidade dos Dados e Locks Distribuídos para Leitura e Escrita em um Sistema de Arquivos para Big Data ou Computação Científica

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Ciências.

São Paulo
2024

ERICO C. DA SILVA

Exploração da Localidade dos Dados e Locks Distribuídos para Leitura e Escrita em um Sistema de Arquivos para Big Data ou Computação Científica

Versão Corrigida

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Ciências.

Área de concentração:
Engenharia de Computação

Orientadora:
Profa. Dra. Liria Matsumoto Sato

São Paulo
2024

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 26 de março de 2024

Assinatura do autor: Erico G da Silva

Assinatura do orientador: Liria Matsumoto Sato

Catalogação-na-publicação

da Silva, Erico
Exploração da Localidade dos Dados e Locks Distribuídos para Leitura e Escrita em um Sistema de Arquivos para Big Data ou Computação Científica / E. da Silva -- versão corr. -- São Paulo, 2024.
176 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Sistemas de arquivo distribuídos 2.Localidade dos dados 3.Big Data 4.Gerenciamento de lock distribuído 5.Análise de dados científicos I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

À minha esposa Luciana, amor da minha vida, a pessoa mais incrível que já conheci.

AGRADECIMENTOS

Agradeço à minha amada esposa Luciana que sempre compreendeu a importância que o doutorado tem na minha vida, me incentivando a cada passo, para que eu não desistisse diante das dificuldades. Foram incontáveis fins de semana, incontáveis noites, mas só eu sei como sou grato por tanto apoio, amor e compreensão.

Agradeço aos meus filhos Tiago e Mateus por terem sido meus companheiros por toda a jornada, sempre me incentivando, sem nenhum questionamento, nenhuma cobrança. Nunca vou conseguir exprimir o tamanho da minha gratidão em palavras.

Agradeço aos meus pais, José e Luiza, por nunca duvidarem do meu potencial, acompanhando todo o meu esforço, rezando para que Deus me iluminasse com sabedoria para evoluir no aprendizado, vencendo cada obstáculo.

Agradeço aos meus irmãos, Fábio e Damaris, por todo incentivo, em especial ao meu irmão Fábio por ouvir minhas "lamentações técnicas"sempre me encorajando e dando sugestões.

Agradeço aos meus sogros, cunhados, sobrinhos, familiares e amigos, que sempre compreenderam quando abdiquei de momentos de alegria e boas conversas, para ficar na frente do computador, codificando, depurando, testando, documentando e... aprendendo.

Agradeço aos amigos do CENAPAD-SP e ao amigo professor Philippe Devloo que me ensinaram tanto no início dos meus estudos de pós-graduação e não deixaram de me incentivar quando decidi iniciar o doutorado, mesmo depois de anos.

Agradeço à Dell Technologies, em especial aos meus amigos e gestores Daniel Dystyler, Humberto Silvestre e Rosângela Pazin, por toda a compreensão e apoio, para que eu pudesse continuar os estudos, mesmo com os desafios tão demandantes dos nossos projetos.

Agradeço ao amigo Fabricio Bronzati pelas dicas sobre os laboratórios de computação de alto desempenho e por todo o apoio me apresentando os caminhos corretos dentro da empresa.

Agradeço aos meus amigos consultores, cientistas, analistas, técnicos e gestores dos tempos da EMC e do projeto Next Best Action, por tantas conversas inspiradoras. Eu admiro cada um de vocês.

Agradeço especialmente ao amigo Ricardo Sugawara pelo companheirismo e inspiração. Só eu sei como foram incríveis os papos sobre teorema CAP, call back, idem-

potência, zookeeper, kerberos e outras tantas viagens. Aprendi muito em todas essas conversas e sobretudo nos tantos exemplos de humildade que me passou no meu início do doutorado.

Agradeço aos professores e funcionários da Escola Politécnica da USP, em especial a todos do Departamento de Engenharia de Computação e Sistemas Digitais (PCS), por terem me acolhido tão bem, me ensinando tanto a cada momento. Meu muito obrigado do fundo do meu coração.

Agradeço todo o companheirismo dos amigos do LAHPC, em especial ao professor Edson Midorikawa por toda paciência e esforço, me ensinando muito, sempre com sugestões enriquecedoras.

Gostaria de agradecer à minha querida orientadora, professora Líria Matsumoto Sato, por compreender o desafio que é conciliar trabalho e pesquisa e por toda a paciência e carinho, sempre indicando os melhores rumos em todos os aspectos. Foram longos anos de uma parceria riquíssima, que mudou a minha vida, das mais variadas formas. Muito obrigado por ter aceitado ser minha orientadora.

Agradeço a Deus por ter me concedido essa oportunidade, me iluminando e me dando forças pra continuar.

“Olhe sempre para a frente, mantenha o olhar fixo no que está adiante de você. Veja bem por onde anda, e os seus passos serão seguros.”

Provérbios 4:25-26

RESUMO

O Big Data tem revolucionado a exploração de dados em larga escala. Ao mesmo tempo, clusters HPC são usados em simulações científicas com resoluções cada vez mais altas, utilizando um volume de dados que vem crescendo de forma acentuada. Embora façam uso de sistemas de arquivo distribuídos mais robustos, clusters HPC movimentam o dado pela rede durante o processamento, enquanto frameworks de Big Data exploram a localidade dos dados para processá-los sem movimentação pela rede, utilizando hardware de baixo custo. Para promover o uso simultâneo de um mesmo cluster por aplicações científicas e processamento de Big Data, este trabalho propõe um novo sistema de arquivos distribuído, o AwareFS. Baseado no padrão POSIX, o AwareFS possui uma arquitetura escalável e resiliente, usando um protocolo de escrita local para explorar a localidade dos dados mesmo durante atualizações. Essa nova tecnologia de armazenamento permite reescrita e acesso randômico, utilizando um sistema distribuído de controle de locks para garantir consistência e flexibilidade no acesso concorrente de múltiplos clientes, tanto na leitura quanto na escrita. Resultados obtidos com benchmarks de mercado comprovaram a eficiência do AwareFS em diferentes perfis de leitura e escrita, sequencial e randômica, demonstrando o benefício do protocolo de escrita local e a escalabilidade acrescentando servidores. O AwareFS contribui para a convergência de tecnologias, possibilitando o uso de um mesmo cluster, mesmo de baixo custo, em cargas de trabalho de Big Data, computação científica e aplicações tradicionais.

Palavras-chave: Sistemas de arquivo distribuído. Hadoop. Big Data. Gerenciamento de lock distribuído. Análise de dados científicos. Localidade dos dados.

ABSTRACT

Big Data has revolutionized the exploration of data on a large scale. Simultaneously, HPC clusters are employed in scientific simulations with increasingly higher resolutions, utilizing a rapidly growing volume of data. While they make use of more robust distributed file systems, HPC clusters transfer data across the network during processing, whereas Big Data frameworks leverage data locality to process them without network movement, utilizing low-cost hardware. To promote the simultaneous use of the same cluster for scientific applications and Big Data processing, this work proposes a new distributed file system, AwareFS. Based on the POSIX standard, AwareFS features a scalable and resilient architecture, using a local write protocol to exploit data locality even during updates. This novel storage technology enables rewriting and random access, employing a distributed lock control system to ensure consistency and flexibility in concurrent access by multiple clients, both in reading and writing. Results obtained from market benchmarks have validated the efficiency of AwareFS across different read and write profiles, sequential and random, demonstrating the benefits of the local write protocol and the scalability by adding servers. AwareFS contributes to the convergence of technologies, enabling the use of the same, even low-cost, cluster in Big Data workloads, scientific computing, and traditional applications.

Keywords: Distributed file systems. Hadoop. Big Data. Distributed lock management. Scientific data analysis. Data locality.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIAÇÕES

NameNode	componente do HDFS que gerencia e armazena os metadados.
DataNode	componente do HDFS que armazena os blocos de dados.
HDFS	Hadoop Distributed File System.
NFS	Network File System.
Metadado	dados que descrevem outros dados, como informações sobre a estrutura, formato ou contexto dos dados.
Namespace	abstração que organiza elementos, como nomes de arquivos para permitir a organização lógica.
Inode	estrutura de dados em sistemas de arquivos que armazena informações sobre um arquivo ou diretório.
WORM	Write Once, Read Many - WORM é um princípio em sistemas de armazenamento que permite a gravação de dados apenas uma vez, mas permite leituras múltiplas.
Scheduling	Escalonamento - processo de atribuição de recursos a diferentes tarefas ou processos.
POSIX	Portable Operating System Interface.
FUSE	Filesystem in Userspace.
Pipeline	sequência de processos nos quais a saída de um processo é a entrada para o próximo.
Checksum	Soma de Verificação - Checksum é um valor numérico calculado a partir de um conjunto de dados para verificar a integridade dos dados.
RPC	Remote Procedure Call.
MPI	Message Passing Interface.
HPC	High Performance Computing.

SUMÁRIO

1 INTRODUÇÃO

Com os avanços tecnológicos recentes, especialmente na última década, o aumento do volume de dados disponíveis para as empresas, gerados por elas ou não, tem crescido para volumes nunca observados. Em (DOBRE; XHAFA, 2014) os autores lembram que 90% dos dados acumulados até 2013, foram gerados depois de 2010. O IDC prevê que mundialmente os dados crescerão de 33ZB em 2018 para 175ZB em 2025 (RYDNING; REINSEL; GANTZ, 2018). Além de grandes bases de dados, logs gerados por máquinas e sensores são fonte valiosa de informação para análises promovidas por cientistas de dados (BLOMER, 2015), mesmo com os desafios próprios de um armazenamento que não é organizado em bancos de dados. Assim, conjuntos de dados cada vez maiores precisam ser manipulados, demandando tecnologias que promovam um processamento eficiente, associado a recursos de compartilhamento e movimentação de dados de alto desempenho, trabalhando com grande diversidade de dados (DOBRE; XHAFA, 2014). Nesse contexto, mesmo com os desafios para o processamento ágil de dados volumosos dos mais variados tipos, como descritos pelo modelo 3V (volume, velocidade, variedade) (BLOMER, 2015),(PATGIRI; AHMED, 2016), as tecnologias de Big Data ganham destaque possibilitando a análise de grandes volumes de dados a partir do uso eficiente de clusters de servidores de baixo custo. Na última década, frameworks para processamento distribuído de Big Data como o Hadoop (WHITE, 2015) e o Spark (ZAHARIA et al., 2010) ganharam destaque, ressaltando a importância dos sistemas de arquivos distribuídos que são centrais para tais ferramentas. No caso de instalações do Hadoop baseadas em ambientes de nuvem privada, o sistema de arquivos distribuído usado na versão original é o HDFS (WHITE, 2015). Com uma arquitetura centrada em dados (do inglês "*data-centric*") (J. Wang et al., 2020), o Hadoop e o HDFS possibilitam um processamento utilizando recursos computacionais do mesmo servidor que armazena os dados, com tarefas distribuídas alocadas por um escalonador orientado à localidade dos dados. Para ga-

rantir redundância, o HDFS divide o dado em partes de cerca de 64MB chamadas *chunk* e armazena pelo menos três cópias de cada *chunk* em servidores distintos, possibilitando operações de leitura com maior desempenho, explorando a localidade do dado (WHITE, 2015), já que todas as cópias podem servir a solicitações de leitura de forma consistente. A arquitetura do HDFS possibilita a utilização de clusters compostos por servidores de baixo custo, onde a falha de hardware é considerada normal ("*failure as norm*") (T. D. Thanh et al., 2008). Contudo, devido à sua semântica WORM (*write-once, read-many*), operações de escrita no HDFS só são possíveis de forma sequencial, ao final de cada arquivo, inviabilizando escrita randômica e/ou concorrente de arquivos. Por exemplo, o HBase (GEORGE, 2011),(YADAV, 2017) que é o banco de dados orientado por colunas do Hadoop, incorpora procedimentos específicos de limpeza, chamados "*compactions*", para renovar arquivos após mesclar dados válidos dos arquivos originais e excluir entradas desnecessárias. Como o HDFS não permite alteração dos dados, tais procedimentos são necessários em clusters HBase, mas podem afetar o desempenho e os resultados de consultas (YADAV, 2017). O HDFS também possui limitações quanto à sua escalabilidade, uma vez que o gerenciamento de metadados não escala horizontalmente, ou seja, não escala com a adição de outros servidores de metadados (WHITE, 2015). Mesmo com suas restrições, o HDFS também é usado como camada de armazenamento para o Spark (ZOU; IYENGAR; JERMAINE, 2020), viabilizando um uso mais eficiente de clusters de servidores a partir de uma arquitetura centrada em dados.

Grandes clusters também são utilizados desde os anos 90 para atender o chamado "processamento de alto desempenho" ou HPC (do inglês "*high performance computing*"), voltado para problemas computacionais complexos próprios de setores como aeroespacial, ciências da vida, finanças e energia (NETTO et al., 2018). Embora a tecnologia de HPC já esteja bem estabelecida, sua arquitetura é voltada para simulações (ELIA; FIORE; ALOISIO, 2021), sendo baseada em uma arquitetura centrada em computação (do inglês "*compute-centric*"), onde o processamento paralelo é baseado em troca de mensagens com um padrão MPI (do inglês "*message passing in-*

terface") manipulando um conjunto de dados compartilhado entre os vários servidores ou nós do cluster HPC. Nesse modelo os dados são armazenados em um sistema de arquivos distribuído e são transferidos pela rede desde os nós de armazenamento até os nós usados no processamento paralelo. Aplicações científicas baseadas na troca de mensagens com bibliotecas como o MPI são construídas de forma que o processamento possa ser dividido em várias partes independentes, cada uma executada em um nó diferente, sem que seja necessária uma sincronização entre elas para chegar à solução final. Esse modelo de processamento, conhecido como embaraçosamente paralelo, requer um sistema de arquivos que permita o uso concorrente dos dados pelas várias partes em execução, de forma consistente e coerente, ou seja, garantindo que o mesmo dado esteja disponível para todos os nós e que todos os nós observem as modificações na mesma ordem. Em (J. Wang et al., 2020) os autores citam que aplicações de análise genética como o parallel BLAST e aplicações de visualização como ParaView e Chimara dependem de clusters HPC para trabalhar com dados de informações genéticas que apresentam volumes cada vez maiores. Com uma semântica muito mais próxima do padrão POSIX, os sistemas de arquivos distribuídos utilizados em clusters HPC como o Lustre (WANG et al., 2009), o PVFS (CARNS et al., 2000) e o GPFS (SAPUNENKO, 2014) proporcionam uma semântica mais flexível, possibilitando o compartilhamento dos dados com acessos concorrentes de leitura e escrita, sequencial ou randômica, garantindo compatibilidade com aplicações também baseadas no padrão POSIX. Porém, como nos clusters HPC as aplicações não processam os dados nos mesmos nós responsáveis pelo armazenamento, a localidade do dado não é explorada, obrigando uma movimentação dos dados até os chamados "nós computacionais" antes do processamento propriamente dito (J. Wang et al., 2020). Para acelerar a movimentação de dados entre os nós, clusters HPC também empregam hardware de rede mais elaborado. Além de não explorar a localidade dos dados, sistemas de arquivos distribuídos como o Lustre e o PVFS dependem da resiliência presente no hardware, inviabilizando o uso de hardware de baixo custo já que usam uma arquitetura onde a falha é a exceção ("*failure as exception*") (T. D. Thanh et al., 2008).

Nesse contexto, temos de um lado as tecnologias de Big Data, que viabilizam o uso de grandes volumes de dados com um processamento que explora a localidade dos mesmos, com movimentação mínima pela rede, usando hardware de baixo custo. Por outro lado, temos as tecnologias de HPC com aplicações bem estabelecidas que fazem uso de clusters com armazenamento desacoplado, permitindo uma utilização mais dinâmica com acesso sequencial e randômico, porém inviabilizando o seu uso com hardware de baixo custo. Essa dualidade de cenários representa um desafio para o uso de clusters de baixo custo no processamento simultâneo de cargas de trabalho de Big Data, simulações e outra aplicações compatíveis com o padrão POSIX .

1.1 Motivação

O sistema de arquivos distribuído é central na arquitetura tanto de clusters HPC como na arquitetura de frameworks para o processamento de Big Data (T. D. Thanh et al., 2008), (BLOMER, 2015), sendo responsável por armazenar os dados e compartilhá-los entre os nós do cluster computacional. Embora nuvens públicas sejam cada vez mais utilizadas em processamento de Big Data, é desafiador prever os custos do uso de computação em nuvem para aplicações de HPC (NETTO et al., 2018), porque as cargas de trabalho dos usuários de HPC envolvem o ajuste de seus aplicativos e a exploração de cenários diferentes por meio da execução de várias combinações de tarefas. Tal situação motiva a utilização de recursos computacionais locais ou em nuvem privada.

Aplicações tradicionais existentes, sem requisitos de paralelismo, em geral são escritas com base no padrão POSIX e podem ser usadas com dados armazenados em sistemas de arquivos distribuídos, desde que estes possuam compatibilidade com este padrão.

Aplicações científicas dependem da semântica mais flexível de sistemas de arquivo distribuídos usados em clusters HPC, desta forma, aspectos de consistência e coerência devem ser observados no armazenamento das informações, o que depende

de um sistema eficiente de controle de *locks* (bloqueios).

Aplicações de Big Data exploram a localidade dos dados no processamento de grandes volumes de informação usando sistemas de arquivo como o HDFS em hardware de baixo custo.

Em todos os casos, o sistema de arquivos distribuído deve ser escalável para que a capacidade de processamento de leituras e escritas cresça à medida que novos nós forem adicionados ao cluster, em um modelo de escalabilidade horizontal.

Para ser usado de forma concorrente por aplicações tradicionais, científicas e cargas de trabalho de Big Data, o sistema de arquivos distribuído deve ser compatível com hardware de baixo custo, deve explorar a localidade dos dados e deve possibilitar acesso consistente e coerente aos dados. Assim, definimos três perguntas de pesquisa: é possível criar um sistema de arquivos que explore a localidade dos dados inclusive em operações de escrita randômica? O processamento de *locks* pode ser gerenciado de forma distribuída para viabilizar leitura e escrita, sequencial e randômica, em um sistema de armazenamento horizontalmente escalável? É possível a construção de um sistema de arquivos distribuído que possibilite a execução de aplicações tradicionais e científicas, com leitura e escrita randômica e que viabilize sua utilização com aplicações de Big Data? A partir desses questionamentos temos a hipótese de que um sistema de arquivos distribuído horizontalmente escalável pode ser construído para promover leitura e escrita, sequencial e randômica, consistente e coerente usando um sistema de gerenciamento de *locks* distribuído e explorando a localidade dos dados. Todas essas características combinadas atenderão perfis de E/S de cargas de trabalho de Big Data e de aplicações científicas ou tradicionais.

1.2 Objetivos

A presente tese visa comprovar a hipótese estabelecida por meio da proposição de um sistema de arquivos distribuído otimizado para processamento de grandes volumes de dados, como os encontrados em ambientes de Big Data. Este sistema deve

explorar a localidade dos dados em operações de leitura e escrita, tanto sequenciais quanto randômicas, mantendo compatibilidade também com aplicações científicas. Além disso, deve oferecer redundância e resiliência no armazenamento, embora o desenvolvimento de recursos de recuperação de falhas não esteja dentro do escopo deste estudo.

Para atingir estes requisitos, tem-se os seguintes objetivos específicos:

- A arquitetura proposta deve ser escalável, permitindo um aumento de desempenho à medida que mais nós são adicionados ao sistema de armazenamento.
- Como o sistema de arquivos deve suportar operações concorrentes de leitura e escrita, também é objetivo propor um sistema distribuído de controle de locks escalável.
- Uma extensa avaliação do sistema de arquivos proposto deve contribuir na comprovação da hipótese.

Dessa forma, serão respondidos todos os questionamentos que embasaram a construção da hipótese.

1.3 Justificativa

Os sistemas de arquivos distribuídos criados para HPC não foram pensados para explorar a localidade dos dados, dificultando sua utilização eficiente com cargas de trabalho de Big Data (T. D. Thanh et al., 2008), (J. Wang et al., 2020), (BLOMER, 2015). Em (DUNNING; FRIEDMAN, 2018) os autores reforçam que o HDFS possui diversas limitações, como permitir um único processo de escrita por arquivo, e apontam o MapR-FS (SRIVAS et al., 2017) como uma “plataforma de dados com as capacidades técnicas necessárias para apoiar tendências emergentes de dados em larga escala na produção”, contudo o MapR-FS, embora muito mais flexível e eficiente que o HDFS, usa um protocolo de escrita que direciona todas as solicitações de escrita sempre para um mesmo nó, ou seja, um protocolo de escrita remota baseado em

primário (TANENBAUM; STEEN, 2007). Mesmo sendo um sistema de arquivos distribuído para Big Data que também é compatível com o padrão POSIX, o MapR-FS não muda o nó com a cópia primária durante o processo de escrita. Com o MapR-FS, a consistência de escritas é garantida por um mecanismo livre de bloqueios ("*lockless*") que, caso um conflito seja identificado, reverte a operação de escrita usando um procedimento de custo computacional elevado. Outros sistemas de arquivos distribuídos usados em HPC, como o GPFS (SCHMUCK; HASKIN, 2002), (SAPUNENKO, 2014) e o Lustre (WANG et al., 2009) usam *locks* para garantir a consistência, porém não foram concebidos para um processamento que explore a localidade do dado, como é a premissa para cargas de trabalho de Big Data.

Assim, a criação de um sistema de arquivos distribuído que explore a localidade do dado até para operações de escrita, usando um sistema de *locks* distribuído e escalável é uma opção inovadora que viabiliza padrões de leitura e escrita próprios de cargas de trabalho de Big Data além dos padrões de E/S característicos de aplicações tradicionais e científicas. O modelo de sistema de arquivos distribuído proposto possibilita o uso de um mesmo cluster para execução de cargas de trabalho de Big Data e aplicações científicas, dispensando o uso de redes de alto desempenho e otimizando o compartilhamento de recursos computacionais entre usos diversos.

1.4 Metodologia

Para embasar todo o trabalho de pesquisa, uma revisão bibliográfica foi conduzida para aprofundamento da compreensão sobre diferentes estratégias para armazenamento distribuído de dados. Vários sistemas de arquivos distribuídos foram estudados, tanto aqueles criados especialmente para processamento de Big Data, quanto aqueles inicialmente concebidos para outros fins como um uso em HPC ou compartilhamento de dados simplesmente. Após essa etapa, teve início uma fase de projeto de componentes com uma verificação de algoritmos e estruturas para o gerenciamento de dados e metadados, estratégias de localização e replicação de dados, técnicas de empacotamento e desempacotamento de dados ("*marshaling/unmarshalling*"), esque-

mas de consistência, comunicação por RPC e troca de mensagens, além de um plano sobre mecanismos de resiliência. Considerando o desempenho necessário, decidiu-se pelo uso do C++14 como linguagem de desenvolvimento e foi criada uma primeira versão do sistema de arquivos, aqui denominado de AwareFS. Essa primeira versão, ainda sem um gerenciamento distribuído de *locks*, mostrou a adequação das estratégias escolhidas para comunicação entre os nós do cluster. Em seguida, uma nova fase de desenvolvimento se seguiu para a implementação usando múltiplas linhas de execução ("*threads*"), já complementando com o desenvolvimento do gerenciamento distribuído de *locks*. Para viabilizar o desenvolvimento futuro de capacidades de detecção de falhas e recuperação, todo o protótipo foi desenvolvido com operações de replicação de dados e persistência de estado, já que capacidades recuperação não são necessárias para os objetivos do presente trabalho de pesquisa. Ao final dessa fase, um módulo cliente básico do AwareFS foi elaborado também em C++14, viabilizando testes integrados mais abrangentes.

Já com o sistema de arquivos distribuído pronto, foi desenvolvida uma primeira versão de um cliente com o padrão POSIX para operações de leitura, escrita e manipulação de metadados, visando viabilizar o uso de ferramentas de "*benchmark*" de mercado na avaliação final do AwareFS. Usando escrita sequencial, o desempenho do protocolo de escrita local do AwareFS foi então comparado ao desempenho do protocolo de escrita remota de outro sistema de arquivos distribuído. Uma avaliação de escalabilidade horizontal foi feita demostrando a robustez da arquitetura do AwareFS.

Como escrita randômica sempre foi um objetivo, todo o processo de replicação de dados foi então reescrito para viabilizar a atualização consistente de pequenas regiões dentro de um arquivo. O trabalho de pesquisa se encerrou repetindo a comparação dos desempenhos dos protocolos de escrita local e escrita remota, procedendo também uma nova avaliação de desempenho de leitura e escrita, sequencial e randômica, usando requisições de 4KB, 128KB e 1MB, com clusters variando de tamanho entre 8 e 36 nós, com e sem o uso do mecanismo de migração do papel de cópia primária (com e sem o protocolo de escrita local). Essa bateria de avaliações promoveu um en-

tendimento claro sobre as vantagens do AwareFS, o sistema de arquivos distribuído proposto.

1.5 Organização do trabalho

O capítulo 2 traz uma revisão bibliográfica com conceitos e ferramentas relacionadas ao processamento e armazenamento de Big Data, destacando tecnologias envolvidas, características de sistemas de arquivos distribuídos, suas vantagens e deficiências.

O capítulo 3 explora detalhes sobre os trabalhos relacionados aprofundando em características de sistemas de arquivos distribuídos e os desafios relacionados, cobrindo tanto os sistemas usados em clusters HPC como os usados em processamento de Big Data.

A proposta do trabalho com detalhes da arquitetura do sistema de arquivos distribuído é detalhada no capítulo 4.

Detalhes técnicos sobre a implementação inicial da proposta são destacados no capítulo 5.

Os testes e métodos para avaliação dos resultados estão no capítulo 6.

O trabalho se encerra no capítulo 8 com as conclusões e possíveis evoluções para o presente trabalho.

2 REVISÃO DA LITERATURA

Em (CHEN; MAO; LIU, 2014) Chen e outros citam que o volume de dados criado e copiado no mundo era da ordem de 1,8ZB em 2011 (GANTZ; REINSEL; others, 2011), tendo crescido cerca de nove vezes em cinco anos. Por exemplo, só o YouTube recebe em torno de 72 horas de vídeo a cada minuto (CHEN; MAO; LIU, 2014). A EMC/IDC cita em (GANTZ; REINSEL; others, 2011) que entre 2011 e 2016 o número de arquivos (ou *containers* de dados) terá crescido a um fator de 8, sendo que o número de profissionais de TI disponíveis para gerenciar esses dados crescerá em um ritmo muito menor. Esse comportamento traz à tona a discussão quanto à eficiência das técnicas de processamento, transferência e armazenamento de grandes volumes de dados. Desse crescimento nunca observado antes do volume de dados gerado e processado continuamente vem o termo Big Data, inicialmente usado para referenciar conjuntos de dados que não pudessem ser trabalhados com tecnologias tradicionais de computação (CHEN; MAO; LIU, 2014)(HASHEM et al., 2014). O termo Big Data vem sendo tratado de forma mais elaborada para se referir a conjuntos de dados que apresente os 3Vs (Volume, Velocidade e Variedade) (DOBRE; XHAFA, 2014)(CHEN; MAO; LIU, 2014). Contudo mais recentemente, passou-se a definir Big Data como uma nova geração de tecnologias e arquiteturas que apresentem 4Vs (Volume, Velocidade, Variedade e Valor) (CHEN; MAO; LIU, 2014),(HASHEM et al., 2014), considerando uma publicação do IDC de 2011 que salienta a necessidade de se encontrar valor no conjunto de dados. Em (DOBRE; XHAFA, 2014), porém, os 4Vs são citados como sendo Volume, Velocidade, Variedade e Veracidade, citando também a importância da análise do conjunto de dados e reforçando a ideia de que os resultados precisam ser confiáveis. Em (ZHANG; XU, 2013), Zhang e Xu ressaltam ainda a complexidade dos dados, propondo o modelo 4V+C (Volume, Velocidade, Variedade, Valor e Complexidade). Independente de quantos “vês” e quais seus significados, as pesquisas em Big Data devem buscar formas de extrair valor do conjunto de dados (CHEN; MAO; LIU,

2014).

2.1 Sistemas de arquivos Distribuídos

Em (T. D. Thanh et al., 2008) Thanh et. al. propõe um modelo de taxonomia para sistemas de arquivos distribuídos, onde as principais características de um sistema de arquivos distribuído são elencadas e discutidas para alguns sistemas de arquivos como o HDFS, GFS, Lustre, entre outros. A taxonomia proposta em (T. D. Thanh et al., 2008) se divide nos pontos citados a seguir.

2.1.1 Arquitetura

As principais arquiteturas são (T. D. Thanh et al., 2008): **Client-Server**: são sistemas de arquivos que disponibilizam uma visão padronizada dos dados, possibilitando que vários clientes, de características diferentes, acessem os dados de forma concorrente, como é o caso do NFS.

Sistemas de arquivos distribuídos baseados em cluster: como o Google File System (GFS) e o HDFS. Nesse tipo de arquitetura um servidor controla os metadados referentes a dados divididos em chunks e distribuídos em outros servidores de dados.

Arquitetura simétrica: onde se enquadram sistemas de arquivos baseados em tecnologia peer-to-peer e DHT (tabelas hash distribuídas);

Arquitetura assimétrica: onde dados e metadados são distribuídos em vários servidores que controlam o sistema de arquivos e suas estruturas de dados. É o caso do Lustre e do Panasas ActiveScale.

Arquitetura paralela: onde os dados são divididos em blocos (striped) e distribuídos por vários servidores de dados. Sistemas de arquivos distribuídos com arquitetura paralela apresentam escalabilidade horizontal e tolerância a falhas, permitindo solicitações concorrentes de leitura e escrita, promovendo acesso simultâneo aos blocos com maior desempenho.

Thanh et. al. reforça que os sistemas de arquivos em geral apresentam características de mais de uma das arquiteturas aqui citadas (T. D. Thanh et al., 2008).

2.1.2 Processos

Os sistemas de arquivos distribuídos são constituídos de processos diferentes que cooperam entre si, como por exemplo os servidores de armazenamento e os gerenciadores de arquivos (TANENBAUM; STEEN, 2007). Quanto aos processos inerentes de um sistema de arquivos, a principal característica sugerida em (T. D. Thanh et al., 2008) é se os processos são sem estado (*stateless*) ou não. Um sistema de arquivos é dito sem estado se o servidor não controla o estado dos arquivos associados a operações de open e close (TANENBAUM; FILHO, 1995). Embora a implementação de um sistema de arquivos distribuído *stateless* seja mais simples, essa característica dificulta operações de *lock* nos arquivos. Por conta dessa abordagem o PVFS2 pode escalar até centenas de servidores e milhares de clientes (T. D. Thanh et al., 2008).

2.1.3 Comunicação

Em geral os sistemas de arquivos distribuídos são compostos por processos e não possuem nada especialmente diferente de outros sistemas distribuídos, sendo que vários deles são baseados em comunicação usando RPC (*Remote Procedure Call*) (TANENBAUM; STEEN, 2007). O RPC é amplamente usado para comunicação entre sistemas, garantindo a independência entre o sistema de arquivos e o tipo de sistema operacional ou arquiteturas em uso pelos componentes. Além do RPC, outra forma de comunicação entre sistemas é o 9P (Plan 9) que é um sistema distribuído onde todos os recursos, mesmo processos e interfaces de rede, são acessados como se faz para o caso de arquivos (T. D. Thanh et al., 2008).

2.1.4 Metadados

O sistema de controle de estruturas de acesso aos dados (onde se enquadram metadados como diretórios, endereços de blocos, tabela de inodes, etc.) é ponto central em um sistema de arquivos distribuído. Assim, são de grande importância itens como:

- O mapeamento entre as estruturas de dados e o dado em si;
- As interfaces para uso do sistema de arquivos;
- Sua capacidade de manter a integridade do dado em caso de falhas.
- Dessa forma, os sistemas de arquivos distribuídos se dividem entre sistemas de arquivos com servidor de metadados central (redundante ou não) e sistemas de arquivos com metadados distribuídos em todos os nós.

2.1.5 Sincronização

Thanh et. al. (T. D. Thanh et al., 2008) classifica a sincronização como item vital a ser considerado no projeto de um sistema de arquivos distribuído. A semântica envolvida no compartilhamento dos dados deve garantir a consistência dos mesmos, impedindo que ações concorrentes de leitura e escrita, executadas por processos e clientes diferentes, possam causar problemas como perda de integridade do dado, erros de processamento, etc.

2.1.5.1 Semânticas de compartilhamento de dados

Vários autores citam as dificuldades de implementação da semântica imposta pelo padrão POSIX em sistemas distribuídos (TANENBAUM; FILHO, 1995),(PARALLEL...,),(STERLING, 2002),(KUBIATOWICZ, 2014),(PATE; BOSCH, 2003),(P; MARCO; others, 2007). Pelo padrão POSIX toda operação de leitura e escrita deve ser atômica e todo dado escrito deve ficar imediatamente disponível para todos os clientes tão logo o processo de escrita tenha acabado. Essa imposição do padrão POSIX

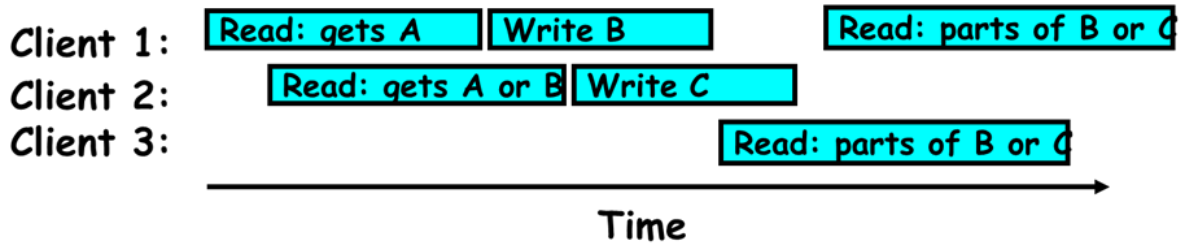

Figura 1 - Operações concorrentes de leitura e escrita ao longo do tempo
Fonte: (KUBIATOWICZ, 2014)

acarreta uma maior complexidade no desenvolvimento de sistemas que empreguem caches distribuídos (STERLING, 2002). Considerando a situação da Figura 1 observamos que as operações de leitura podem ter resultados diferentes dependendo da forma como o sistema de arquivos atende as operações de escrita e armazena os dados. Em um sistema de arquivos local o comportamento provável seria:

- Se a leitura começa depois que uma escrita termina, o dado novo é retornado;
- Se a leitura termina antes que uma escrita inicia, o dado antigo é retornado;
- Caso contrário, é retornado o dado novo ou antigo dependendo da ordem de execução das operações envolvidas.

A eficiência do padrão adotado para definir o comportamento de leitura/escrita (como na situação da Figura 1) vai depender da aplicação que faz uso do sistema de arquivos. Contudo, um tipo de aplicação pode precisar de garantias que duas gravações concorrentes não resultarão em uma “mistura” dos dados, enquanto outro tipo de aplicação pode abrir mão dessa garantia buscando um desempenho melhor. Das diversas semânticas presentes na literatura, podemos destacar as seguintes (TANENBAUM; STEEN, 2007):

- **Semântica Unix**: Uma leitura sempre retorna o valor armazenado pela última escrita. Assim, quando duas escritas se sucedem rapidamente, seguidas de uma leitura, o valor lido será o que foi armazenado pela última escrita.

- **Semântica de sessão**: As mudanças feitas em um arquivo aberto só se tornam visíveis para outros processos quando o arquivo é fechado. Mais relaxada que a semântica Unix, esse tipo de semântica facilita o uso de caches e é amplamente usado em sistemas de arquivos distribuídos.

- **Semântica de arquivos imutáveis**: Uma vez criados, todos os arquivos não podem mais serem alterados. Os arquivos só podem ser abertos para criação ou leitura. Essa semântica também é referida como WORM (write once, read many), sendo a de implementação mais simples em sistemas de arquivos distribuídos já que simplesmente desaparece a situação em que um processo lê um arquivo enquanto outro o altera.

- **Semântica de transações**: Todas as leituras e/ou escritas em um arquivo são iniciadas com um comando de INICIO_DE_TRANSACAO e encerradas com um comando de FIM_DE_TRANSACAO. Dessa forma, é garantido que todas as operações são feitas em ordem e sem interferência de outras operações concorrentes.

2.1.5.2 Locks - Bloqueios para dados e metadados

Quando um arquivo pode ser acessado por mais de um processo ao mesmo tempo, o sistema de arquivos precisa garantir que um processo não cause a corrupção de um dado por manipulá-lo enquanto ele está sendo usado por outro processo. Por exemplo, o que acontece se dois processos escrevem ao mesmo tempo em um mesmo trecho de um arquivo compartilhado (P; MARCO; others, 2007)? Em sistemas de arquivos distribuídos, o compartilhamento de arquivos, seus dados e metadados, é necessidade central e demanda mecanismos de sincronismos que garantam a consistência dos dados em condições de uso concorrente de dados compartilhados.

Para garantir a consistência dos dados, muitos sistemas de armazenamento de dados permitem que vários processos leiam um dado, desde que este não esteja sendo alterado por nenhum outro processo. Estes sistemas de armazenamento possuem um

gerenciador de travamento (*lock*) que controla a concessão de locks e a consistência do dado só é garantida se os locks cabíveis forem concedidos para os processos manipulando os dados. Cada operação, como leitura e escrita, possui um *lock* específico a ser requisitado e o gerenciador de locks só concede o *lock* se o tipo solicitado for compatível com os locks já concedidos anteriormente para operações ainda em curso. Os locks podem ser classificados em duas categorias (P; MARCO; others, 2007):

- **Advisory locks**: São locks onde o efeito só é atingido se os processos compartilhando o arquivo verifiquem a existência de locks antes do uso dele.
- **Mandatory locks**: Os locks são verificados e honrados pelo sistema de arquivos no momento da abertura do arquivo.

No NFSv4, por exemplo, um processo pode requerer um *lock* de leitura ou escrita em uma região do arquivo e este só é concedido se não houver uma outra operação conflitante em curso. Caso haja uma operação conflitante em andamento o processo solicitando o *lock* recebe uma mensagem de erro e deve submeter novamente o pedido de *lock* depois de algum tempo (TANENBAUM; STEEN, 2007).

No caso do NFS para sistemas Windows, ao abrir um arquivo o processo informa quais os acessos que ele requer (leitura, escrita ou ambos) e quais os tipos de acessos que o sistema de arquivos deve negar para outros processos (nenhum, leitura, escrita ou ambos). Essa forma implícita de *lock* é conhecida como reserva de compartilhamento (*share reservation*) (TANENBAUM; STEEN, 2007).

O mesmo tipo de preocupação quanto à integridade dos dados compartilhados acontece durante o acesso concorrente de metadados. Considere o caso de um sistema de arquivos organizados em uma árvore de diretórios onde os arquivos são referenciados pelo caminho completo (*pathname*). Nesse cenário se um processo estiver escrevendo em um arquivo ‘/a/b/file.txt’, um outro processo não poderá renomear o diretório de ‘/a/b’ para ‘/a/c’ ou a consistência da árvore de diretórios será perdida (MALHOTRA, 2014).

2.1.6 Consistência, replicação e tolerância a falhas

Os sistemas de arquivos distribuídos em geral controlam a consistência dos dados com o uso de *checksum* (um *checksum* é calculado e verificado após a transferência entre sistemas).

Para garantir a disponibilidade dos dados, os metadados são protegidos em todos os sistemas de arquivos distribuídos. Alguns têm os metadados armazenados de forma redundante, outros proporcionam um log de transações para reconstrução dos metadados.

Quanto à replicação dos dados os sistemas de arquivos distribuídos em geral usam duas estratégias:

A falha na camada física é um caso de exceção (Falha como exceção): O dado é disponível desde que o meio físico esteja disponível, ou seja, os dados são armazenados e a redundância e disponibilidade do dado ficam a cargo da camada de armazenamento físico, ou seja, o hardware deve ter redundância robusta. É o caso do Lustre e do Panasas.

A falha na camada física é algo esperado (Falha como regra): O dado é replicado em diferentes servidores, mesmo que componentes de hardware falhem, os componentes de software devem reestabelecer a operação sem perdas. É o caso do HDFS e do GFS que usam um *pipeline* de replicação e demandam maior banda de comunicação entre os servidores.

Quanto à tolerância à falha para o controle de metadados, os sistemas de arquivos distribuídos possuem diferentes estratégias para recuperação após a falha de um componente, seja com a redundância de controles de metadados, mantendo um servidor de metadados em stand-by, seja mantendo backups ou logs de transações para recuperação dos metadados para um ponto consistente.

2.1.7 Segurança

Assim como em sistemas centralizados, a autenticação de usuários e o controle de acesso aos dados também é importante em sistemas de arquivos distribuídos.

Sistemas de arquivos distribuídos como o GFS e o HDFS baseiam sua segurança nos mecanismos implementados no sistema operacional em uso.

2.1.8 Uso de hardware de baixo custo

Uma última característica dos sistemas de arquivos distribuídos elencada em (T. D. Thanh et al., 2008) é a utilização efetiva de hardware de baixo custo. Esse tipo de estratégia viabiliza a criação de clusters de servidores usando hardware disponível em larga escala no mercado, diminuindo os custos envolvidos.

2.2 Sistemas de Arquivos Distribuídos baseados em Clusters

Considerando que aplicações de alto desempenho em geral fazem uso simultâneo de vários servidores, é natural que esse tipo de software tenha como base um sistema de arquivos próprio para os clusters em questão (TANENBAUM; STEEN, 2007). Assim, para possibilitar que um mesmo arquivo seja trabalhado por vários servidores ao mesmo tempo, em geral ele é dividido em partes menores que são espalhadas pelo cluster. Uma técnica frequentemente usada é a de usar estratégias para dividir o arquivo em tiras (*stripes*) que são distribuídas nos servidores de forma a possibilitar que cada tira seja processada em paralelo, cada uma pelo servidor que a armazena. Tanenbaum e Steen ressaltam que essa técnica funciona bem para o tratamento de arquivos relativamente grandes e de estrutura regular, como é o caso de matrizes densas (TANENBAUM; STEEN, 2007), porém existem outros casos para os quais faz mais sentido distribuir os arquivos pelos servidores sem dividi-los em parte menores (Figura 2). Considerando a Figura 2 (a) observamos que todos os acessos do arquivo “a” serão direcionados para o mesmo armazenamento, ao passo que usando a estra-

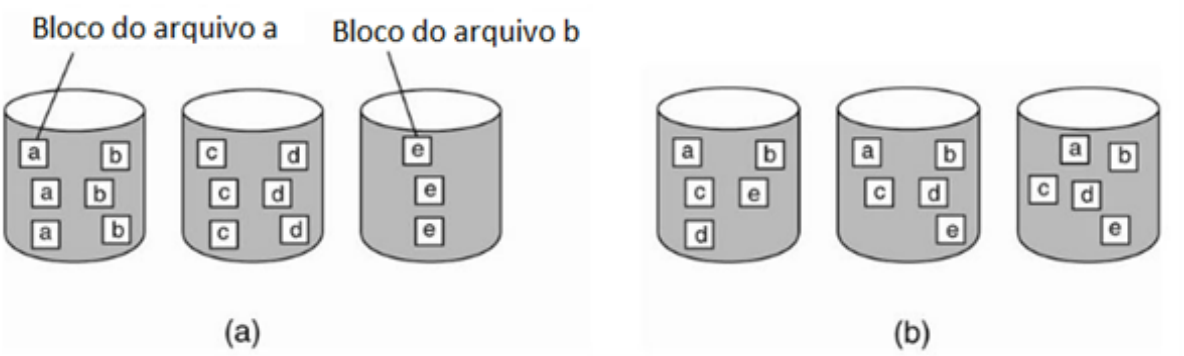

Figura 2 - Diferença entre (a) distribuir os arquivos sem dividi-los e (b) dividi-los em tiras para acesso paralelo
Fonte: (TANENBAUM; STEEN, 2007)

tégia de divisão por tiras, a carga referente ao acesso ao arquivo “a” será distribuída pelos três diferentes armazenamentos.

Vários sistemas de arquivos distribuídos como o GFS e o HDFS usam a estratégia na parte b da Figura 2. Tanto o HDFS como o GFS dividem os arquivos em partes menores chamadas *chunks*. O termo *chunk* é usado por diversos sistemas de arquivos distribuídos para designar as fatias em que um arquivo é dividido para ser armazenado e processado em paralelo.

2.3 Ferramentas de Big Data

Tradicionalmente, o tratamento de dados é feito em geral depois que estes são organizados e estruturados, com o uso de bancos de dados relacionais (RDBMS, *relational database management system*) (CHEN; MAO; LIU, 2014),(HASHEM et al., 2014). Devido às características do Big Data de heterogeneidade e volume, a comunidade científica e profissionais da área vêm propondo novas alternativas, das quais se destacam os bancos de dados NoSQL (bancos de dados *schema-free* para grandes volumes de dados distribuídos (HASHEM et al., 2014)) e os sistemas de arquivos distribuídos (CHEN; MAO; LIU, 2014).

Os bancos de dados NoSQL (*Not only SQL*) ganharam destaque depois da publicação de (CHANG et al., 2008) por técnicos da Google, descrevendo a ideia do

Bigtable, um sistema de armazenamento estruturado de dados que pode atender volumes da ordem de petabytes. Com base em (CHANG et al., 2008) surgiram outros bancos como o HBase, MongoDB, Cassandra, Voldemort, entre outros (HASHEM et al., 2014).

2.3.1 MapReduce

Para o processamento distribuído de grandes volumes de dados, usando hardware de baixo custo, técnicos da Google publicaram (DEAN; GHEMAWAT, 2008) onde é detalhada uma forma de abstração onde se podem criar programas baseados em primitivas de programação map e reduce sem se preocupar com complexidades inerentes de processamento paralelo, como redundância do dado e tolerância a falhas. O modelo de programação usando primitivas map e reduce não é uma ideia nova, tendo sido originalmente usada em linguagens funcionais como o Lisp,

2.3.2 Hadoop

Baseado em (DEAN; GHEMAWAT, 2008), Doug Cutting, que estava à frente do projeto Nutch da Apache (um *open source web search engine*), iniciou um subprojeto para implementar o modelo MapReduce usando o NDFS (sistema de arquivos distribuído do Nutch). Esse subprojeto deu origem ao Hadoop e seu sistema de arquivos distribuído (o HDFS) (WHITE, 2015).

O Hadoop é hoje a ferramenta de processamento de Big Data mais difundida, sendo usada por empresas como Yahoo! (CHEN; MAO; LIU, 2014),(SHVACHKO et al., 2010) e Facebook (CHEN; MAO; LIU, 2014) para tratar petabytes de dados usando milhares de computadores.

Embora o Hadoop possa ser usado como plataforma básica para desenvolvimento baseado em map e reduce, toda a eficiência da ferramenta fica restrita ao que o sistema de arquivos distribuído provê. Isso acontece pois o sistema de processamento MapReduce é acoplado ao sistema de arquivos distribuído (HDFS), assim o

sistema de armazenamento não é fisicamente separado do sistema de processamento (WHITE, 2015) (HASHEM et al., 2014).

Em (SHAFER; RIXNER; COX, 2010), Shafer et. al. propõe formas para melhorar o HDFS e conclui dizendo que o sistema de arquivos distribuído servindo de base para o MapReduce do Hadoop pode ser a causa de baixo desempenho.

3 TRABALHOS RELACIONADOS

Além de usos em Internet Services e Cloud Computing, sistemas de arquivos distribuídos é um assunto já bem estudado para usos no âmbito do HPC (*High Performance Computing*) (YANG et al., 2014),(TANTISIRIROJ; PATIL; GIBSON, 2008). Desse interesse nasceram vários sistemas de arquivos distribuídos, destacados a seguir.

3.1 MapR-FS

MapR é uma distribuição comercial do Hadoop implementada em C/C++ para melhorar a performance. Uma das principais características do MapR é seu file system, o MapR-FS (SRIVAS et al., 2017), que foi desenvolvido em cima de uma tecnologia chamada "*lockless storage services*"que propõe um armazenamento distribuído (*random read-write*) com alta disponibilidade.

3.1.1 Características básicas do MapR-FS

Embora seja considerada uma distribuição de Hadoop, o MapR e seu MapR-FS possuem características que o diferenciam das demais distribuições baseadas no Hadoop Apache, das quais podemos citar as seguintes:

- **NameNode distribuído**: Em distribuições Hadoop tradicionais o NameNode, o nó que gerencia os metadados, é o serviço responsável por controlar todo o "*namespace*" (ou "*naming space*") que é o catálogo de arquivos e pastas armazenados, associando seus nomes com os blocos de dados armazenados em disco. Nas versões básicas do HDFS, o NameNode é um componente único e centralizado, limitando o número de arquivos e diretórios a uma quantia que possa ser representada com o espaço de memória disponível para tal servidor. Além dessa limitação quanto ao tamanho do "*naming space*", o fato de o NameNode ser um

único servidor implica em cuidados para garantir uma redundância desse componente que possibilite o reestabelecimento do ambiente com o menor tempo de indisponibilidade possível. O MapR-FS por sua vez usa um modelo onde o metadado é replicado e persistido em pelo menos três nós diferentes. Com o MapR-FS é possível controlar um número maior de arquivos e diretórios aumentando o número de servidores de metadados.

- **Semântica de reescrita**: Diferente do HDFS presente em outras distribuições Hadoop, o MapR-FS não usa uma semântica de arquivos imutáveis. Com isso, embora pouco usual em processos MapReduce tradicionais, o MapR-FS possibilita que um arquivo seja alterado.

- **NFS-Mountable**: Para expor mais facilmente os dados, o MapR-FS pode ser apresentado para máquinas clientes usando o protocolo NFS. Isso facilita a manipulação dos dados.

- **Dados em raw device**: Com o MapR-FS é possível criar volumes de armazenamento usando diretamente dispositivos de blocos (raw devices) do Linux. Essa alternativa apresenta melhor desempenho por eliminar camadas de gerenciamento de dados como um sistema de arquivos do Linux.

- **Outros recursos para gerenciamento dos dados**: O MapR-FS dispõe de recursos encontrados em sistemas de arquivos de primeira linha tais como cópia instantânea de volumes (*snapshots*), duplicação de volumes (*mirrors*), quotas para controle de uso, colocação de dados que permite que o administrador indique qual nó vai armazenar qual dado (*data placement*).

3.1.2 Estrutura de armazenamento do MapR-FS

Para armazenar os dados de forma eficiente e distribuída o MapR-FS dispõe das seguintes estruturas para organizar o armazenamento:

- **Storage Pools**: Composto em geral por pelo menos três discos, um *storage*

pool é um agrupamento de discos para facilitar a administração do sistema de arquivos distribuído;

- **Containers**: Para um uso mais eficiente os volumes são particionados em pedaços relativamente pequenos chamados *containers*. Um *container* do MapR-FS tem um tamanho variando entre 16 e 32GB. Um arquivo é dividido em pedaços menores chamados "*chunks*" que são distribuídos em *containers* diferentes para armazenamento.

- **Volumes**: Os arquivos e diretórios são organizados em volumes. Um arquivo presente em um volume pode extrapolar *containers*, *storage pools* e nós.

- **Chunks**: São divisões de tamanho variável, sendo esse tamanho configurado no nível do diretório. O tamanho default de um *chunk* é 256MB, porém é possível configurar qualquer tamanho múltiplo de 64KB. O tamanho do *chunk* tem relação direta com o paralelismo alcançado em um processo MapReduce.

3.1.3 CLDB - Container Location Data Base

O MapR-FS possui uma arquitetura com características específicas para garantir a escalabilidade e desempenho maiores que a média dos outros sistemas de arquivos distribuídos usados em ambientes Hadoop. A arquitetura do MapR-FS é composta por clientes, servidores de arquivos e um componente chave na arquitetura: o CLDB (*Container Location Data Base*). Trata-se de uma base de dados que armazena basicamente informações sobre em quais nós os *containers* estão armazenados. Embora esta seja uma base de dados única e centralizada, o MapR-FS faz um *cache* eficiente dessa base em todos os demais componentes da arquitetura. Isso é possível pois a CLDB é pequena, equivalente ao número de *containers*, e não sofre alterações frequentes já que só é alterada se um *container* for criado, removido ou movido.

3.1.4 Leitura, Escrita e Replicação

Assim como em outros sistemas de arquivos distribuídos, para garantir tolerância a falhas, o armazenamento dos dados é replicado entre os servidores de arquivos do MapR-FS. Cada *container* possui uma cadeia de replicação composta por nós (servidores de arquivos), sendo que um deles assume o papel de mestre da cadeia.

Em processamento baseado em MapReduce, o processamento é direcionado para o nó que armazena o dado. Da mesma forma que acontece no Hadoop e seu HDFS, no MapR-FS, qualquer nó que tenha uma réplica do dado pode ser usado para executar uma tarefa sobre ele. Especificamente no caso do MapR, um dado pode ser acessado para leitura em qualquer nó que possua uma réplica, porém em casos de escrita, o dado deve necessariamente ser manipulado no nó mestre da cadeia de replicação. Essa estratégia, citada em (TANENBAUM; STEEN, 2007) por Steen e Tannenbaum como “protocolos baseados em primários”, faz com que um nó seja obrigado a transferir os dados relacionados com uma escrita para o mestre da cadeia de replicação, mesmo que ele possua localmente uma réplica do dado.

3.1.5 Atualização sem bloqueio (lockless updates)

Mesmo sem empregar nenhum recurso de *lock*, o MapR-FS garante consistência para atualizações dos dados. O gerenciamento de arquivos compostos por vários *chunks* que são armazenados e alterados em nós diferentes requer transações envolvendo vários nós e *containers* (SRIVAS et al., 2017). Para viabilizar a atualização de arquivos desse tipo, o MapR-FS usa uma estratégia que dispensa o uso de locks, consistindo dos seguintes pontos:

- Em situações de atualização, todos os nós envolvidos gravam nos seus respectivos logs de transações chamados WAL (*Write Ahead Logging*).
- Existem dois tipos de WAL, um para registrar as transações nos dados e outro registrando as operações nos metadados. Ambos registram também as ações

requeridas para desfazer a transação (*rollback*).

- Com esses logs uma transação pode ser desfeita de forma razoavelmente custosa.
- As réplicas vão monitorar as ações para identificar conflitos. Caso um seja identificado, a transação é desfeita (*rollback*), do contrário a transação é confirmada.

3.2 Lustre

Em (WANG et al., 2009) Wang et. al. descreve o Lustre como sendo um sistema de arquivos distribuído e paralelo de arquitetura escalável. Lustre é um sistema de arquivos baseado em objetos que apresenta uma interface POSIX para os clientes com capacidade de acesso paralelo aos objetos compartilhados (dados e metadados) que compõem os arquivos.

A arquitetura básica do Lustre se baseia nos seguintes componentes escaláveis:

- **Servidor de metadados (MDS)**: Cada servidor desse tipo é responsável por gerenciar um MDT (*metadata target*) onde são armazenadas informações de arquivos e diretórios como nome, datas de acesso, etc. para um sistema de arquivos.
- **Servidor de armazenamento de objetos (OSS)**: Esse servidor gerencia um ou mais OST (*object storage target*) que é onde são armazenados os dados dos arquivos armazenados. É o OSS que vai expor dispositivos de blocos (*block devices*) e servir os dados.
- **Servidor de gerenciamento (MGS)**: É esse servidor que deve servir informações de configuração do sistema de arquivos Lustre.
- **Clientes**: É o componente que implementa chamadas do padrão POSIX como open(), read() e write(). O cliente Lustre se integra com o VFS (*virtual file system*) do Linux.

3.2.1 LDLM - Lustre Lock Manager

A ideia do LDLM, gerenciador de locks do Lustre, vem do VAX DLM (distributed lock manager). Para o Lustre, sempre que um lock é solicitado, a operação é feita sobre um item em um domínio chamado namespace. No Lustre, dados gerenciados por OSTs, metadados gerenciados por um MDS ou itens de configuração gerenciados pelo MGS, são recursos (resources) que compõem namespaces e podem ter seu uso bloqueado por locks gerenciados pelo LDLM.

Os modos locks gerenciados pelo LDLM são os seguintes:

- EX: Exclusivo, onde o solicitante requer acesso exclusivo ao recurso;
- PW: Protective Write (escrita normal), para operações de escrita onde um cliente deve alterar um recurso de um OST;
- PR: Protective Read (leitura normal), para operações de leitura onde um cliente deve acessar um recurso de um OST;
- CW: Concurrent Write, lock solicitado pelo MDS quando um cliente solicita acesso de escrita a um arquivo;
- CR: Concurrent Read, lock usado pelo MDS para os itens intermediários de um caminho (pathname) de um arquivo;
- NL: Null Mode, sem uso.

Os locks podem ainda ser classificados dentro de quatro tipos distintos, definidos pelo cliente, a saber:

- Extent lock: para proteção de recursos de um OST;
- flock: para atender requisição de lock da aplicação no modelo de lock padrão flock (POSIX);
- Inode bit: lock para proteção de metadados;

- Plain lock: foi definido na arquitetura inicial do Lustre e substituído pelo lock de tipo inode bit. Foi mantido na arquitetura, mas atualmente não tem função.

Ainda no processo de geração e manutenção de *locks*, o Lustre usa em seu LDLM o conceito de função de call-back, que é uma função passada pelo cliente no momento da solicitação do lock, a ser invocada pelo LDLM em momento oportuno. Existem três tipos de call-back:

- Blocking: Invocado se outro cliente pede um lock conflitante ou se o lock é revogado;
- Completion: Invocado se o lock é cedido ou se o lock é convertido para outro modo;
- Glimpse: Para passar informações ao cliente sem necessariamente liberar o lock;

A Figura 3 mostra a compatibilidade entre *locks* para os modos possíveis. Os *locks* são considerados compatíveis entre si se para cada lock as áreas correspondentes do arquivo não se sobrepuserem ou se a interseção de seus modos na tabela da Figura 3 resultar em 1. Por exemplo, conforme indicado na Figura 3, uma solicitação de leitura protegida (lock PR) será recusada se for feita para uma região do segmento onde um lock de gravação protegida (lock PW) foi previamente concedido, pois a célula PR/PW será 0, indicando a incompatibilidade entre esses tipos de *locks*.

O Lustre também implementa no seu LDLM um conceito de "*intent*", onde o cliente pode passar ao gerenciador de *locks* mais informações sobre o objetivo final da operação envolvendo o *lock*, de forma que o número de chamadas RPC possa ser diminuído. Assim, o *intent* são informações passadas ao LDLM que indicam que é preciso um processamento diferente durante o enfileiramento do *lock*.

São seis intenções possíveis: Get atributes, Set atributes, Insert ou delete, Open, Create e Read links.

	NL	CR	CW	PR	PW	EX
NL	1	1	1	1	1	1
CR	1	1	1	1	1	0
CW	1	1	1	0	0	0
PR	1	1	0	1	0	0
PW	1	1	0	0	0	0
EX	1	0	0	0	0	0

Figura 3 - Matriz de compatibilidade de locks.
Fonte: (WANG et al., 2009)

A requisição de *locks* no Lustre segue basicamente os seguintes passos:

1. O cliente LDLM examina se o *lock* pertence ao *namespace* local (cada LDLM atende um *namespace*). Se for local, ou seja, se não requer chamada remota, pula pro passo 7 abaixo;

2. Se não for local, o cliente LDLM envia comando de *lock* enqueue para o servidor de LDLM em questão;

3. Se o pedido de *lock* possui *lock intent*, pula para o passo 6 abaixo. Se não tem *intent*, o LDLM invoca uma função de política (policy function) específica para o tipo do *lock*. Como existem quatro tipos de *lock* (inodebits, extent, flock e plain), existem quatro funções de política. A função de política do LDLM tem por objetivo determinar se o *lock* solicitado é conflitante com outra solicitação existente.

4. O LDLM verifica então se existem conflitos com *locks* cedidos ou enfileirados. Se não houver o lock é cedido.

5. Havendo conflito, a solicitação de *lock* é colocada em uma fila e o cliente recebe uma resposta “*being blocked*” informando que o *lock* foi requisitado com sucesso, porém não foi cedido de imediato.

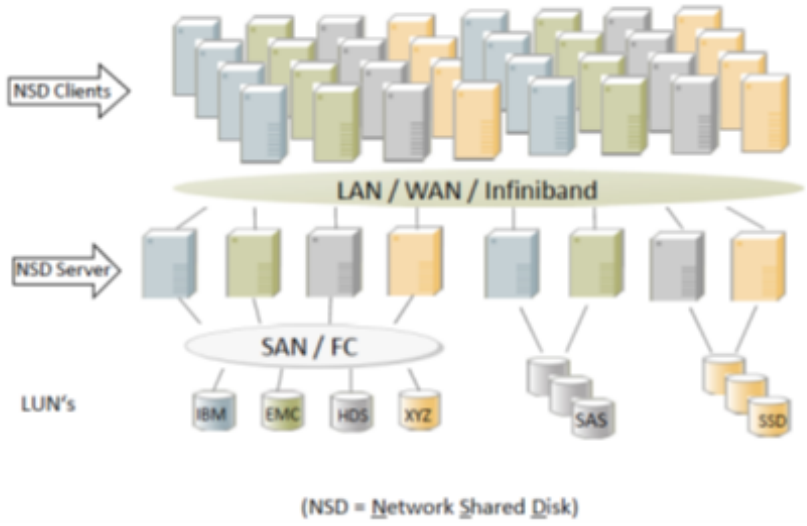

Figura 4 - Arquitetura do GPFS
Fonte: (SAPUNENKO, 2014)

6. Se um *intent* foi passado no pedido de *lock*, o LDLM deve chamar uma função específica para cada intente. Essa função é registrada pelo gerenciador do namespace (OST ou MDS). O LDLM só retorna o veredito dado por tal função.

7. Sendo um *lock* local, o LDLM cria o *lock* e verifica se ele pode ser cedido, retornando o *lock* concedido ou uma resposta “being blocked”.

3.3 GPFS

Assim como o Lustre, o GPFS é um sistema de arquivos paralelo para clusters que chega próximo ao que determina o padrão POSIX (SCHMUCK; HASKIN, 2002). O GPFS consegue grande escalabilidade graças à sua arquitetura de compartilhamento de discos. Os servidores que compõem o cluster do GPFS conectam-se aos discos ou subsistemas de discos usando uma rede de troca (*switching fabric*), sendo que todos os nós têm acesso a todos os discos. Os arquivos são fatiados e espalhados nos discos do cluster. A rede para conexão com os discos pode ser do tipo SAN ou iSCSI. A arquitetura do GPFS se divide nos seguintes componentes (SAPUNENKO, 2014):

- **Node**: unidade dentro do cluster (host com sistema operacional);

- **Network Shared Disk (NSD)**: dispositivo de armazenamento disponível ao cluster para uso no sistema de arquivos;
- **NSD server**: servidor dentro do cluster que gerencia o acesso ao NSD;
- **GPFS file system**: sistema de arquivos construído usando um conjunto de NSDs;
- **Application node**: nó do cluster que monta o sistema arquivos e executa aplicações que acessam o sistema de arquivo.

3.3.1 Gerenciamento de locks e acesso aos metadados

Para aumentar desempenho e eficiência, o GPFS usa um esquema de gerenciadores de bloqueio (*lock managers*) distribuídos. Um *lock manager* central trabalha em conjunto com *lock managers* presentes em cada nó. Cada *lock manager* distribuído recebe *lock tokens* (fichas de bloqueio) que delegam o direito de fazer locks sobre um recurso sem que haja a necessidade de comunicação com o *lock manager* central a cada *lock*. O nó que possui o *lock token* pode fazer cache do dado em questão pois não se altera nada sem que um *lock* seja negociado.

Byte range *token*: Quando um segundo nó precisa escrever o mesmo dado, o *token* em poder do primeiro nó precisa ser revogado para pelo menos parte do dado em questão (*byte-range token*). Se o primeiro nó já tiver terminado de escrever, o *token* é revogado por inteiro. Os conflitos diminuem se o padrão de acesso permitir que seja prevista a região do arquivo a ser acessada. Isso cobre acessos sequenciais e espaçados, adiante e reverso, desde que seja em regiões grandes (*coarse-grain sharing*).

Se a semântica POSIX não for necessária, pode-se mudar o esquema de bloqueio de byte-range para “data-shipping”, que é um esquema onde o dado é distribuído igualmente entre os nós (*round-robin*) e um bloco é lido ou gravado sempre por um único nó. Isso é mais eficiente para compartilhamentos de grão mais fino (*fine-grain*

sharing) do que locks distribuídos. Data shipping é usado principalmente em aplicações com MPI/IO, mas é possível usar com qualquer tipo de aplicação (SCHMUCK; HASKIN, 2002).

Para o caso de metadados, a atualização sincronizada usando *locks* de escrita exclusiva no inode resultaria em conflito para cada escrita. Por isso, no GPFS a escrita usa um tipo de *lock* de escrita compartilhado (*shared write lock*) no *inode*, permitindo escrita concorrente por vários nós. Essa forma só gera conflitos em operações que precisam obter o tamanho exato do arquivo e/ou tempo de modificação. Para atualização eficiente dos metadados, um nó é designado como "*metanode*" e só ele lê ou escreve o inode do disco. O *metanode* combina as atualizações no *inode* oriundas de vários nós, retendo o maior tamanho do arquivo e o momento de modificação mais recente.

Desta forma, o GPFS usa lock distribuído para garantir semântica POSIX, mas o acesso aos *inodes* é feito de forma centralizada via *metanode*, o que possibilita vários nós escrevendo sem conflito nas atualizações ao metadado, dispensando a troca de mensagens com o *metanode* em toda operação de escrita (SCHMUCK; HASKIN, 2002).

3.4 OrangeFS - PVFS

Outro exemplo de sistema de arquivos paralelo é o PVFS (CARNS et al., 2000) e seu sucessor, o OrangeFS (MOORE et al., 2011). O OrangeFS segue um modelo do tipo "*Object based storage*" onde o serviço de dados e serviço de metadados trabalham separadamente. Embora o OrangeFS busque uma semântica próxima da presente nos sistemas de arquivos tradicionais, assim como os demais sistemas de arquivos paralelos/distribuídos, ele não chega a atender 100% o padrão POSIX. Sua semântica não apresenta o conceito de arquivo aberto, já que o servidor não armazena o estado das conexões que estão manipulando os arquivos. No caso em que um arquivo é removido por um cliente, por exemplo, os demais clientes perdem acesso ao

arquivo mesmo que a abertura do arquivo tenha sido bem-sucedida.

Para garantir a consistência o processo de criação de um arquivo do OrangeFS segue a seguinte ordem:

1. Os objetos para armazenamento dos dados do novo arquivo são criados;
2. Os objetos para armazenamento dos metadados do novo arquivo são criados;
3. Os metadados são “apontados” para os objetos de dados;
4. A entrada de diretório é criada para referenciar os metadados para o novo arquivo.

Dessa forma o arquivo só fica visível para o usuário final se todo o processo for bem-sucedido.

O OrangeFS só garante atomicidade no processo de escrita para o caso de regiões sem sobreposição.

Em se tratando de metadados, o OrangeFS pode fazer cache de diretorios por tempo configurável.

Tantisiriroj et. al. citam em (TANTISIRIROJ; PATIL; GIBSON, 2008) que o PVFS não faz cache dos dados no lado do cliente, o que simplifica seu funcionamento por dispensar o uso de protocolos complexos para a consistência de *caches*. Contudo, essa estratégia não é ideal para arquivos relativamente pequenos.

3.5 GFS

Precursor do HDFS, o Google File System (GFS) foi descrito por Ghemawat et. al. em (GHEMAWAT; GOBIOFF; LEUNG, 2003), onde descrevem a ideia do trabalho sem grandes detalhes de como foi desenvolvido. O GFS foi criado pela Google para atender necessidades as quais não poderiam ser supridas com sistemas de arquivos distribuídos desenvolvidos até então, por exemplo, devido ao tamanho do cluster, era

prudente assumir que a qualquer momento no tempo, é provável que pelo menos um nó esteja apresentando problemas (TANENBAUM; STEEN, 2007).

O projeto do GFS teve como base algumas premissas (GHEMAWAT; GOBIOFF; LEUNG, 2003) citadas a seguir:

- **A falha é a regra**: um sistema de arquivos que pudesse escalar até centenas ou milhares de servidores deveria ser capaz de trabalhar em situações em que falhas de hardware fossem esperadas.
- **Grandes arquivos**: o tamanho dos arquivos seria da ordem de gigabytes, dado que a Google comumente combinava centenas de arquivos da ordem de kilobytes em arquivos maiores.
- **Append Simp**: os arquivos sendo manipulados sofriam adição de dados ao seu final, sem necessidade de alteração em pontos aleatórios.
- **Integração com as aplicações**: as APIs para uso do GFS foram projetadas em conjunto, de forma a flexibilizar o desenvolvimento. Como exemplo, os autores citam em (GHEMAWAT; GOBIOFF; LEUNG, 2003) que o modelo de consistência do GFS foi flexibilizado de forma a simplificá-lo sem transferir a complexidade para as aplicações que o utilizassem.

3.5.1 Arquitetura do GFS

O GFS possui os seguintes componentes básicos (TANENBAUM; STEEN, 2007):

- **Mestre**: um componente único no cluster, responsável principalmente por gerenciar e prover metadados sobre os arquivos. É o mestre que faz o mapeamento desde o nome do arquivo, até as divisões que o compõem.
- **Servidor de porção**: servidores responsáveis pelo armazenamento das várias porções (*chunks*) que compõem um arquivo.

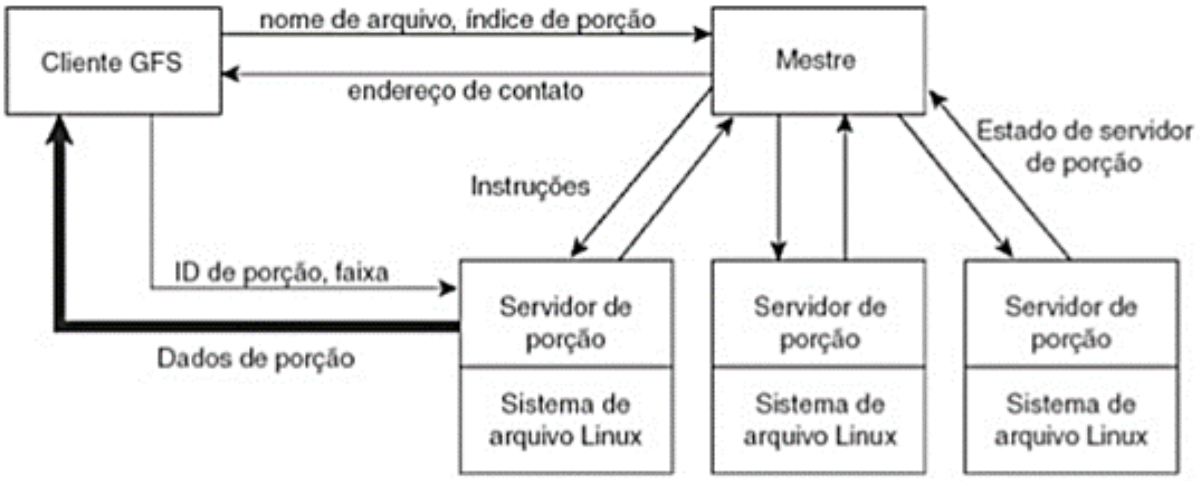

Figura 5 - Organização de um cluster de servidores da Google
Fonte: (TANENBAUM; STEEN, 2007)

O armazenamento no GFS é feito dividindo os arquivos em porções menores chamadas *chunks*. Os *chunks* são replicados para redundância e distribuídos nos servidores do cluster para armazenamento. Tanenbaum e Steen ressaltam em (TANENBAUM; STEEN, 2007) que o mestre não tenta manter uma lista precisa das localizações das porções, fazendo contatos esporádicos com os servidores de porção para se informar de quais porções foram armazenadas em quais servidores. A Figura 5 mostra como um cluster GFS é organizado.

3.6 HDFS

Para servir de base para o conjunto de ferramentas que compõem o Hadoop e seu MapReduce, tem-se um sistema de arquivos distribuído que é definido por White em (WHITE, 2015) como “um sistema de arquivos projetado para armazenar arquivos muito grandes com padrões de acesso a dados em fluxo contínuo, executados em clusters de hardware de baixo custo”. Sobre essa definição, White segue destacando que:

- *Arquivos muito grandes* faz referência a arquivos da ordem de centenas de megabytes, gigabytes ou terabytes;

- *Dados em fluxo contínuo* se refere à ideia de que o HDFS é feito para trabalhar com arquivos em uma semântica "*write once, read many*" em que, para o tipo de processamento alvo do Hadoop, os arquivos em geral são lidos desde a fonte e processados na sua totalidade, sem que seja possível alterá-los depois do primeiro processo de escrita.

- *Hardware de baixo custo* se refere ao fato de que o HDFS não precisa de nenhum hardware específico, nem com grandes recursos de redundância e resiliência. Assim, o HDFS pode conviver com muitas falhas nos servidores que o compõem.

3.6.1 Arquitetura do HDFS

Assim como outros sistemas de arquivos distribuídos, o HDFS armazena os metadados separadamente dos dados. Os metadados são controlados por um componente chamado NameNode e os dados por um conjunto de servidores chamados DataNodes, todos interligados por uma rede TCP-IP (SHVACHKO et al., 2010).

- **NameNode**: Também no HDFS, os arquivos são organizados em uma hierarquia de diretórios compondo um espaço de nomes. Arquivos e diretórios são representados por inodes, que são estruturas para armazenar detalhes como nomes, permissões, datas de alteração, etc. (SHVACHKO et al., 2010). Todo o espaço de nomes do HDFS é gerenciado pelo NameNode, sendo a capacidade de memória deste componente um limitador para a escalabilidade do sistema de arquivos (SRIVAS et al., 2017). Por ser um ponto único da arquitetura, o NameNode conta com um servidor secundário que constantemente recebe uma cópia das informações de metadados, podendo assumir o papel do NameNode primário em caso de falha.

- **DataNode**: Todos os arquivos armazenados no HDFS são divididos em partes menores de cerca de 64MB chamadas de blocos, sendo que arquivos que forem menores que um bloco ocuparão apenas o espaço em disco necessário (WHITE, 2015). Cada bloco armazenado no HDFS é composto por dois arquivos: um com

os dados em si e um segundo com alguns metadados para garantir a consistência dos dados (SHVACHKO et al., 2010). Cada bloco é replicado em pelo menos mais outros dois DataNodes (WHITE, 2015). Durante o processo de inicialização, todos os DataNodes se comunicam com o NameNode para se identificar e passar informações sobre os dados que armazena (WHITE, 2015) (SHVACHKO et al., 2010).

- **HDFS Client**: As aplicações tiram proveito do HDFS por meio de uma biblioteca referida como HDFS Client. É o HDFS Client que se comunica com o NameNode para trabalhar com informações de metadados e com os DataNodes para transferir e receber os dados. O HDFS, assim como sistemas de arquivos semelhantes, provê operações como leitura e escrita de arquivos, criação e exclusão de arquivos e diretórios e outras operações próprias de sistemas de arquivos (SHVACHKO et al., 2010). Durante o processo de escrita, o HDFS Client negocia a operação com o NameNode, trabalhando os metadados, enviando o dado para um DataNode que transfere os blocos para os DataNodes que armazenarão as réplicas. Todos os DataNodes trabalham em conjunto com o NameNode. De forma semelhante, no processo de leitura o HDFS Client recebe do NameNode uma lista com os DataNodes que possuem os blocos e tenta obtê-los dos DataNodes mais próximos, passando, em caso de falha, para os outros nós com réplicas dos dados, em sequência (SHVACHKO et al., 2010).

3.7 Limitações do Hadoop e do HDFS

Considerando os ambientes de programação com map e reduce, o Hadoop é certamente um dos mais robustos e difundidos. Estudos como (DOBRE; XHAFA, 2014), (SHAFER; RIXNER; COX, 2010) e (HUA et al., 2014) citam vários pontos de atenção quando ao desempenho do Hadoop, dos quais se destacam os seguintes:

Tratamento dos arquivos temporários: Todos os resultados intermediários são armazenados em disco, localmente em cada nó (DOBRE; XHAFA, 2014). Assim,

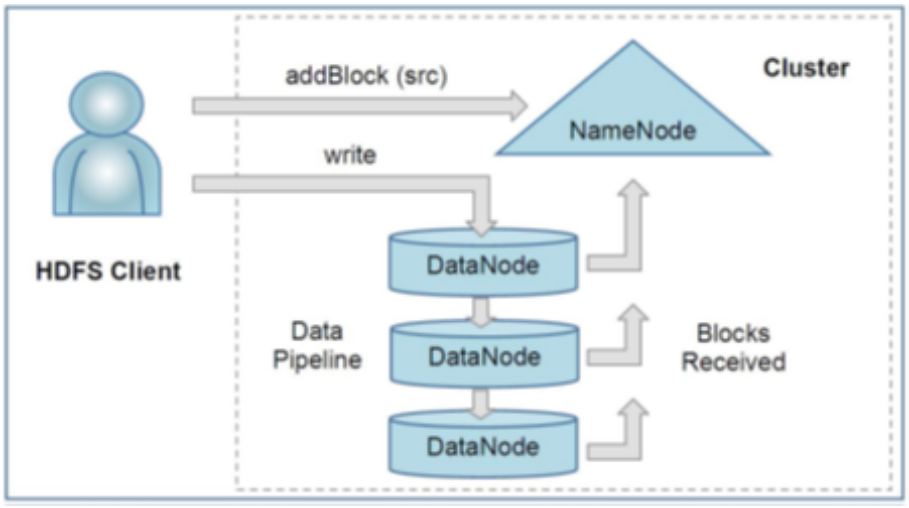

Figura 6 - Criação de arquivo e blocos no HDFS
Fonte: (SHVACHKO et al., 2010)

processamentos que envolvam vários ciclos consecutivos de map e reduce perdem desempenho por esperarem a escrita e leitura dos arquivos de resultados. Spark (ZAHARIA et al., 2010) é um ambiente para processamento distribuído de dados que tem ganhado destaque por causa de ganhos de desempenho justamente nesse ponto.

Modelo de consistência WORM: Para diminuir a necessidade de sincronização entre os processos, o HDFS se baseia em um modelo de consistência onde os arquivos não podem ser alterados depois da primeira escrita (SHAFER; RIXNER; COX, 2010), garantindo a coerência do dado. Esse modelo atende bem aplicações com características de *streaming* de dados, dificultando, porém, o uso eficiente do Hadoop em outros tipos de aplicações.

Single writer: Em HDFS, um único processo pode gravar um arquivo em um dado momento (WHITE, 2015). Em (YANG et al., 2014) Yang et. al. salienta que o Hadoop e seu HDFS atende apenas escrita paralela tipo inter-file N-N, onde N processos escrevem N arquivos, tendo cada processo um arquivo independente.

Append only: Em um sistema de arquivos HDFS tradicional, os dados são adicionados com a criação de novos arquivos ou adicionando dados ao final de um arquivo existente (WHITE, 2015)(SHVACHKO et al., 2010). Dessa forma, aplicações que impliquem em alteração ou remoção de blocos intermediários de um arquivo existente

não são atendidas de forma eficiente pelo HDFS padrão.

Arquivos pequenos: No caso do HDFS, todos os metadados do sistema de arquivos devem caber na memória do NameNode (servidor único que guarda o mapeamento entre arquivos, blocos e servidores de dados ou DataNodes) (WHITE, 2015). Assim, usar arquivos menores para armazenar os dados implica em um uso ineficiente do NameNode. Além disso, o uso de arquivos pequenos facilita a ocorrência de fragmentação de arquivos causando perda de espaço em disco (ZHANG; XU, 2013).

Blocos grandes versus uso concorrente: Em HDFS o tamanho de bloco utilizado é 64MB por default (WHITE, 2015),(SHAFER; RIXNER; COX, 2010). A estratégia de usar blocos grandes melhora o desempenho de leitura e escrita por diminuir o tempo em que um nó fica ocioso esperando scheduling (SHAFER; RIXNER; COX, 2010). Contudo, as requisições de I/O chegam ao disco em pacotes menores (64KB) e atividades de MapReduce simultâneas podem obrigar que as requisições de disco sejam intercaladas, aumentando a ocorrência de seeks, anulando o efeito da estratégia de se usar blocos grandes (SHAFER; RIXNER; COX, 2010). Em outras palavras, com o HDFS a leitura e escrita concorrente de dados debilita o desempenho do sistema de arquivos.

Não há prefetch: Mesmo em um uso de streaming, com uma forma de acesso altamente previsível, não há nenhuma antecipação de carga de dados, técnica conhecida como prefetch (o HDFS só faz prefetch de metadados) (SHAFER; RIXNER; COX, 2010).

3.8 Ceph

Dentro dos sistemas de arquivos paralelos, onde os dados são armazenados em servidores específicos de forma desacoplada, o Ceph (DEPARDON; MAHEC; SÉGUIN, 2013),(WEIL et al., 2006),(WEIL, 2007) vem se destacando como sistema de armazenamento para ambientes que requerem compatibilidade como padrão POSIX (SPECIFICATIONS, 2008). Baseado em um "*distributed object store*" chamado RA-

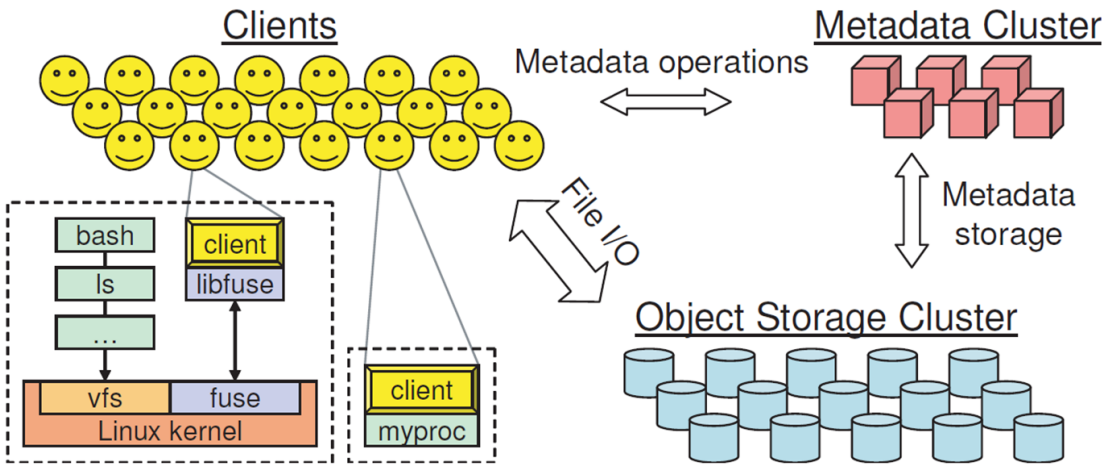

Figura 7 - Arquitetura do Ceph
Fonte: (WEIL et al., 2006)

DOS, o Ceph se destaca por separar o gerenciamento de dados e de metadados (Figura 7), fazendo uso de um algoritmo de distribuição de dados pseudoaleatório chamado CRUSH (WEIL et al., 2006).

3.8.1 RADOS

RADOS (*Reliable, Automatic Distributed Object Store*) é um sistema distribuído de armazenamento de objetos que busca promover uma distribuição equilibrada de dados e carga de trabalho sobre um cluster de heterogêneo de servidores, permitindo que as aplicações possam interagir logicamente como interagem com um único sistema de armazenamento de objetos, com semântica de segurança bem definida e fortes garantias de consistência (WEIL, 2007). Tanto replicação de dados como detecção e recuperação de falhas são operações gerenciadas pelos OSDs (*Object Store Daemons*), serviços implementados em cada servidor do cluster de armazenamento. Um conjunto de serviços de monitoração (*monitors*) trabalha para controlar a formação do cluster de OSDs.

3.8.2 CRUSH

Para garantir distribuição uniforme de dados, o Ceph distribui os dados em PGs (*Placement Groups*) usando uma função hash simples. Cada PG é então assinalado aos OSDs usando um algoritmo chamado CRUSH (*Controlled Replication Under Scalable Hashing*), uma função de distribuição pseudoaleatória de dados. Diferente de outras estratégias baseadas em listas de metadados, com o CRUSH a localização de objetos requer apenas o PG e o mapa do cluster de OSDs que é uma descrição hierárquica e compacta dos dispositivos que compõem o cluster de armazenamento (WEIL et al., 2006),(WEIL, 2007).

3.8.3 Controle de metadados

Para lidar com o desafio que é o controle de metadados em sistemas de arquivos altamente escaláveis, o Ceph conta com um cluster específico de MDSs (*Metadata Servers*). São os MDSs que controlam operações de metados (e.g. open, rename) enquanto os serviços clientes se comunicam diretamente com os OSDs para operações de leitura e escrita dos dados. A carga de trabalho para gerenciamento de metadados é dinamicamente distribuída atribuindo subdivisões da árvore de diretórios a cada MDS. Para garantir consistência sem perda de desempenho, o Ceph trabalha com três semânticas diferentes no controle de metadados. Uma semântica para atributos de segurança como owner e mode, outra semântica para atributos imutáveis como *inode number, ctime* e *layout*, e uma terceira para os atributos de tamanho e horário de modificação (*size* e *mtime*), sendo que cada semântica é governada por máquinas de estado finito independentes, com estados e transições específicas para o comportamento de atualizações de cada atributo de metadado.

3.9 Outras iniciativas

OneFS: O Dell EMC Isilon (SATO et al., 2013), hoje chamado de PowerScale, é um sistema de armazenamento composto por vários nós com armazenamento local,

utilizando o OneFS como sistema de arquivos distribuído para gerenciar todo o armazenamento resultante. Os dados podem ser acessados via LAN, usando HDFS além de diversos protocolos como NFS, HTTP, FTP, REST e SMB. Desacoplando o armazenamento de dados dos nós computacionais, o Isilon permite que as aplicações carreguem os dados usando protocolos mais acessíveis como NFS ou CIFS, sendo que as aplicações Hadoop podem acessar o mesmo dado via HDFS. Com o Isilon o armazenamento dos dados pode escalar independente dos nós computacionais.

SCALER: Uma evolução do HDFS proposta em (YANG et al., 2014). Proporciona escrita tipo N-1 (N processos gravando um arquivo) em HDFS por aplicações MPI. O Hadoop obriga escrita N-N, sendo que cada processo escreve seu próprio arquivo, o que dificulta o uso do HDFS em aplicações de HPC. Com "*inter block sharing*" os processos compartilham o mesmo arquivo, mas não podem compartilhar o mesmo bloco. Para viabilizar o compartilhamento de blocos por vários processos concorrente, o SCALER propõe um novo gerenciamento de locks para o HDFS. Com o "*intra block sharing*", onde os processos escrevem no mesmo bloco de forma concorrente, o SCALER utiliza o método de "*block aggregation*" como o MPI-IO, adaptado para o HDFS, onde a localidade do dado é considerada. O SCALER proporciona o uso do HDFS também por aplicações baseadas em MPI, via MPI-IO. O SCALER é baseado no esquema de *streaming write* do HDFS, onde não há função seek para arquivos abertos para escrita.

FusionFS: Sistema de arquivos local aos compute nodes de HPC que permite que as aplicações manipulem resultados intermediários e *checkpoints* sem tráfego de rede. O FusionFS suporta operações intensivas de metadado, com desempenho de escrita superior (ambas características importantes para HPC) (ZHAO, 2014).

LBFS: Low Bandwidth Network File System é um sistema de arquivos com armazenamento "desduplicado", onde os dados são divididos em blocos e armazenados uma única vez, dispensando solicitações de escrita para blocos já armazenados. O LBFS utiliza um armazenamento de tamanho de bloco variável que consome uma ordem de magnitude menos recursos de rede (MUTHITACHAROEN; CHEN; MAZIE-

RES, 2001).

MemFS: Sistema de arquivos *in-memory* que não depende da localização do dado, é completamente simétrico e faz *stripe* dos arquivos por todos os nós. Porém é um sistema de arquivos com padrão diferente do POSIX, com uma semântica restritiva do tipo *write once, read many* (UTA; SANDU; KIELMANN, 2016).

DeDu: Sistema de armazenamento “desduplicado”, que evita o armazenamento duplicado de arquivos em um ambiente baseado no framework Hadoop. Consiste em uma camada cliente que identifica se um arquivo já foi armazenado anteriormente calculando um *hash* e verificando em um índice se um arquivo com o mesmo *hash* já foi armazenado anteriormente, evitando que dados sejam armazenados em duplicidade (SUN; SHEN; YONG, 2013). É uma estratégia com foco especificamente na eliminação de duplicidade em nível de arquivo, mantendo a semântica restritiva do HDFS.

A literatura cita frequentemente outros sistemas de arquivos como BeeGFS, GlusterFS, XtreemFS (DOBRE; XHAFA, 2014),(WHITE, 2015),(TANTISIRIROJ; PATIL; GIBSON, 2008),(DEPARDON; MAHEC; SéGUIN, 2013),(ZHAO, 2014),(UTA; SANDU; KIELMANN, 2016),(CHOWDHURY et al., 2019).

3.10 Comparação entre sistemas de arquivos distribuídos

Segundo a taxonomia proposta em (T. D. Thanh et al., 2008) a Tabela 1 traz um resumo sobre os principais sistemas de arquivos distribuídos considerados.

Complementando a tabela 1 elaborada em (T. D. Thanh et al., 2008), é importante citar o seguinte sobre o MapR-FS (SRIVAS et al., 2017):

Arquitetura: Baseada em cluster, assimétrica, paralela, baseada *blockbased*

Processos: *Stateful*

Comunicação: RPC / TCP

Tabela 1 - Comparação de diferentes sistemas de arquivos distribuídos

Sistema de arquivos	GFS	KFS	Hadoop	Lustre	Panasas	PVFS2	RGFS
Arquitetura	baseada em cluster, assimétrica, Paralela, objectbased	Baseada em cluster, assimétrica, Paralela, objectbased	Baseada em cluster, assimétrica, paralela, objectbased	Baseada em cluster, assimétrica, paralela, objectbased	Baseada em cluster, assimétrica, paralela, objectbased	Baseada em cluster, simétrica, paralela, baseada em agregação	Baseada em cluster, simétrica, paralela, blockbased
Processos	Stateful	Stateful	Stateful	Stateful	Stateful	Stateless	Stateful
Comunicação	RPC / TCP	RPC / TCP	RPC / TCP e UDP	Independente de rede	RPC / TCP	RPC / TCP	RPC / TCP
Metadados	metadados Central	Metadados Central	Metadados Central	Metadados Central	Metadados Central	Metadados distribuídos em todos nós	Metadados distribuídos em todos os nós
Sincronização	Write-once-readmany, Multipleproducer / singleconsumer, locks em objetos para os clientes, utilizando leases	Write-once-readmany, locks em objetos para os clientes, utilizando leases	Write-once-readmany, da locks em objetos para os clientes, usando leases	Mecanismo de bloqueio híbrido, utilizando leases	Locks em objetos para clientes	Nenhum método de lock, não há leases	Locks em objetos para clientes
Consistência e replicação	Replicação do lado do servidor, Replicação assíncrona, checksum, consistência relaxada entre replicações de objetos de dados	Replicação do lado do servidor, Replicação assíncrona, checksum	Replicação do lado do servidor, Replicação assíncrona, checksum	Replicação do lado do servidor - Apenas replicação de metadados, Cache do lado do cliente, checksum	Replicação do lado do servidor Apenas replicação de metadados	Nenhuma replicação, semântica relaxada para consistência	Sem replicação
Tolerância a falha	Falha é regra	Falha é regra	Falha é regra	Falha é exceção	Falha é exceção	Falha é exceção	Falha é exceção
Segurança	Sem mecanismo de segurança dedicado	Sem mecanismo de segurança dedicado	Sem mecanismo de segurança dedicado	Segurança na forma de autenticação, autorização e privacidade	Segurança na forma de autenticação, autorização e privacidade	Segurança na forma de autenticação, autorização e privacidade	Segurança na forma de autenticação, autorização e privacidade

Fonte: (T. D. Thanh et al., 2008)

Metadados: Metadados central e distribuído em todos os nós. Parte dos metadados é central e replicado e parte dos metadados é replicada em todos os nós.

Sincronização: *Write-once-read-many*, da locks em objetos para os clientes, usando *leases*. Consegue permitir escrita randômica, tipo *write-many-read-many*, com consistência.

Consistência e replicação: Replicação do lado do servidor, Replicação assíncrona, *checksum*. Replicação baseada em *snapshots* de *containers*. *Updates* distribuídos sem bloqueio (*lockless*). Toda escrita é feita usando o nó *master* da cadeia de replicação.

Tolerância a falha: Falha é regra.

Segurança: Segurança na forma de autenticação, autorização e privacidade.

Tabela 2 - Comparação Ceph, HDFS, Lustre e outros

	HDFS	**iRODS**	**Ceph**	**GlusterFS**	**Lustre**
Arquitetura	Centralizada	Centralizada	Distribuida	Descentralizada	Centralizada
Naming	Índice	base de dados	CRUSH	EHA	Índice
API	CLI, FUSE, REST, API	CLI, FUSE, API	FUSE, mount, REST	FUSE, mount	FUSE
Detecção de falha	Totalmente conectada	P2P	Totalmente conectada	Detectada	Manualmente
Disponibilidade	Sem failover	Sem failover	Alta	Alta	Failover
Disponibilidade dos dados	Replicação	Replicação	Replicação	RAID-like	Não implementado
Estratégia de posicionamento	Automática	Manual	Automática	Manual	Não implementado
Replicação	Assíncrona	Síncrona	Síncrona	Síncrona	RAID-like
Consistência de cache	WORM, lease	Lock	Lock	Não implementado	Lock
Balanceamento de carga	Automática	Manual	Manual	Manual	Não implementado

Fonte: (DEPARDON; MAHEC; SéGUIN, 2013)

Além do trabalho feito em (T. D. Thanh et al., 2008), Benjamin Depardon et. al. fazem uma outra comparação onde enquadram o Ceph em uma taxonomia similar comparando-o com GlusterFS e iRODS além do HDFS e do Lustre na Tabela 2 (DEPARDON; MAHEC; SéGUIN, 2013).

3.11 FUSE

FUSE (*File System in User Space*) (BIJLANI; RAMACHANDRAN, 2019) é um framework para desenvolvimento de sistemas de arquivos em modo usuário. Composto por um driver de kernel do Linux e uma biblioteca de desenvolvimento, o FUSE provê uma interface de funções que possibilita a escrita de um sistema de arquivos completo em modo não privilegiado (modo usuário). O driver do FUSE é uma camada que serve de canal de comunicação entre a aplicação em modo usuário e o módulo VFS (virtual file system) do sistema operacional (PATE; BOSCH, 2003). Toda a robustez e facilidades entregues pelo FUSE garantiram seu uso em inúmeros sistemas de

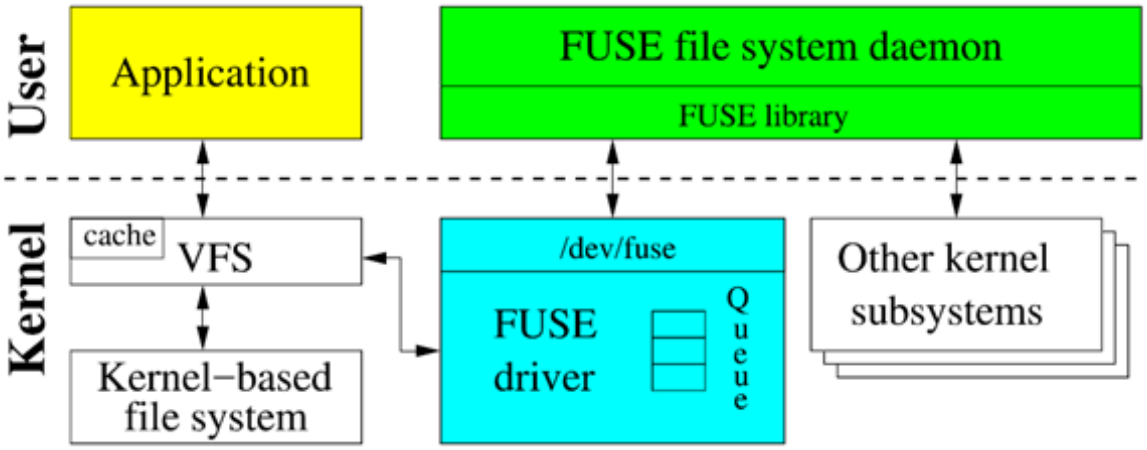

Figura 8 - Arquitetura do FUSE
Fonte: (VANGOOR et al., 2019)

arquivos especializados, sendo utilizado também em sistemas de arquivos distribuídos para computação de alto desempenho e Big Data (BIJLANI; RAMACHANDRAN, 2019),(VANGOOR et al., 2019).

3.12 Sistemas de arquivos distribuídos – desafios

Considerando os principais sistemas de arquivos distribuídos usados em processamento de alto desempenho e Big Data, podemos destacar os pontos a seguir como desafios de implementação.

3.12.1 Desafios de arquitetura – acesso a disco

Sistemas de arquivos distribuídos precisam garantir velocidade de leitura e escrita sem que haja queda de desempenho impeditiva quando o número de processos concorrentes cresce. No caso do HDFS, por exemplo, a arquitetura beneficia usos onde os dados são lidos de forma contínua (*streaming*), o que não acontece quando o número de sessões de escrita aumenta e o disco é forçado e se mover entre *streams* de dados diferentes (SHAFER; RIXNER; COX, 2010). Para melhorar o desempenho de I/O, Shafer et. al. propõe em (SHAFER; RIXNER; COX, 2010) que seja usado uma thread para cada disco, de forma que não haja *interleave* de sessões de escrita em

um mesmo disco, aumentando assim a granularidade de leitura. Em outras palavras, a leitura/escrita de um fluxo de dados não é interrompida tão frequentemente para atender outras requisições de leitura/escrita.

3.12.2 Stateful versus stateless

Para garantir que os arquivos e/ou blocos sejam acessados de forma a manter a consistência do dado, o sistema de arquivos precisa ter um controle de estado para cada arquivo/bloco. Por exemplo, se um arquivo for aberto por um processo com exclusividade, nenhum outro processo poderá acessá-lo. Esse impedimento é liberado quando o arquivo é fechado (TANENBAUM; FILHO, 1995). Os sistemas de arquivos com esse tipo de controle de estado são chamados de *stateful*.

Embora o uso de bloqueios (*locks*) seja importante para garantir a consistência do dado sob uso concorrente de diversos processos, sua implementação em sistemas de arquivos distribuídos não é trivial. Esses sistemas de bloqueio trazem uma maior latência para o sistema e são frequentemente muito complicados por causa da necessidade de suportar falhas (PARALLEL. . . ,). Assim, sistemas de arquivos distribuídos como o OrangeFS (ou o PVFS2) optam por implementações *stateles*, onde o acesso aos dados é feito com uma mensagem de lookup, contendo o nome do arquivo com uma solicitação para procurá-lo e retornar sua estrutura de metadados possibilitando o acesso. Nesse tipo de controle sem armazenamento de estado, uma eventual falha do servidor não causa problemas pela perda do controle dos acessos definidos na operação de open (TANENBAUM; FILHO, 1995).

3.12.2.1 Escrita concorrente e o uso de cache distribuído

Para aumentar o desempenho de leitura e escrita, os sistemas de arquivos distribuídos podem fazer uso de *cache* em níveis diferentes. Contudo, para garantir a consistência do dado, os mecanismos de consistência e sincronização de *cache* se tornam uma fonte potencial de gargalos (TANTISIRIROJ; PATIL; GIBSON, 2008).

Dessa forma o HDFS utiliza *cache* no lado do cliente, enviando dados ao DataNode apenas quando o volume de dados a ser transferido alcança o tamanho do *chunk* ou bloco, ou quando o arquivo é fechado. Isso é possível por causa da semântica adotada pelo HDFS que garante que apenas um cliente escreve um arquivo por vez (TANTISIRIROJ; PATIL; GIBSON, 2008).

Por sua vez o PVFS2 permite escrita concorrente de arquivos (desde que uma mesma região não seja escrita ao mesmo tempo), porém sem nenhum tipo de uso de *cache* do lado do cliente, o que dispensa o uso de mecanismos de sincronização e consistência de caches distribuídos (TANTISIRIROJ; PATIL; GIBSON, 2008).

3.12.3 Locks

Em se tratando de consistência, HDFS e MapR-FS funcionam de forma diferente. Para o HDFS, a semântica WORM dispensa um controle de consistência mais elaborado, uma vez que apenas um cliente escreve um bloco, o qual não será alterado posteriormente. Já para o MapR-FS, existe um mecanismo para alterações sem bloqueio (*lockless transactions*) onde as réplicas monitoram as alterações e as operações conflitantes são desfeitas para garantir a consistência (*roll-back*). Esse mecanismo *lockless* do MapR-FS é relativamente custoso, podendo causar impactos para escritas concorrentes concentradas em uma única região do arquivo.

Diferente do que acontece para os sistemas de arquivos para Big Data, mecanismos mais elaborados para gerenciamento de *locks* estão presentes em sistemas de arquivos distribuídos criados originalmente para HPC, como é o caso do Lustre (WANG et al., 2009). Contudo, esses sistemas de arquivos usam uma arquitetura desacoplada onde os nós de armazenamento de dados não são usados para o processamento das informações, o que descarta o uso eficiente da localidade dos dados. Nesse modelo o sistema de arquivos conta com a resiliência do hardware usado, em geral com custo mais alto e onde a falha é um caso de exceção.

Tanto o método flock (fornecido por sistemas de arquivos tipo BSD), como o fcntl

(do padrão POSIX) implementam *locks* do tipo *advisory*, isto é, fica a cargo da aplicação obedecer aos *locks* presentes, sem a possibilidade de ocorrer *dead locks* em nível de sistema operacional. Nem o PVFS2, nem o MapR-FS suportam *advisory locks* como os implementados com fcntl ou flock.

Embora a simplificação garantida pela inexistência de *locks* em sistemas de arquivos distribuídos como o PVFS2 e MapR-FS possibilite a criação de clusters com um alto número de nós, ficam prejudicadas as aplicações com semântica que demande escritas concorrentes em uma mesma região de um arquivo.

O Lustre usa uma solução híbrida para o sistema de bloqueio de arquivos. Com o Lustre, o modo de *lock* depende do nível de contenção de recursos (T. D. Thanh et al., 2008).

Os sistemas de arquivos distribuídos em geral que trabalham com *locks* e escalam até um número alto de nós dependem de uma semântica do tipo *write-once-read-many* e são desenhados para ambientes onde a falha da camada física é a regra. Por outro lado, os sistemas de arquivos distribuídos que não dependem de semântica *write-once-read-many*, em geral são desenhados partindo do princípio de que falhas da camada física são exceção (TANTISIRIROJ; PATIL; GIBSON, 2008)(SRIVAS et al., 2017)(T. D. Thanh et al., 2008).

3.12.4 A semântica WORM e a localidade dos dados

Dentro dos sistemas para processamento de Big Data o Hadoop se destaca como um dos mais utilizados. No Hadoop o processamento baseado na estratégia de map e reduce procura levar as tarefas de map para os nós onde residem os dados designados à cada tarefa. Esse processo chamado de *Data Locality Optimization* (otimização de localidade de dados) tem por objetivo diminuir a utilização da largura de banda disponível para o cluster. Com a semântica simplificada do HDFS, os processos de map não podem sobrescrever os dados no meio do arquivo de entrada gerando obrigatoriamente novos arquivos para armazenamento dos resultados. Os arquivos de resultado

de processos map do Hadoop são armazenados em disco localmente em cada nó (em file system local do nó, externo ao HDFS), gerando necessariamente tráfego de rede para passagem de dados para processos de reduce subsequente e/ou para armazenamento do resultado no HDFS. Para contornar essa limitação do MapReduce do Hadoop, o Spark (ZAHARIA et al., 2010) busca uma diminuição de arquivos intermediários usando a memória dos nós, estratégia cuja eficiência é proporcional à capacidade de RAM dos hosts. No Hadoop é frequente a ocorrência de casos em que o escalonador de tarefas observa que não existem recursos computacionais suficientes em nenhum dos nós com réplicas dos dados. Assim, a restrição que explora a localidade dos dados é relaxada e a tarefa é direcionada para nós que não possuem réplicas dos dados sendo processados. Isso vai ser frequente em clusters com uso concorrente para muitos fins ou por muitos usuários.

Como a semântica WORM do HDFS obriga a adequação das aplicações ao modelo MapReduce, onde cada map apenas lê os arquivos de entrada, a localidade dos dados pode ser conseguida usando qualquer um dos três nós com réplicas. Outro sistema de arquivos de Big Data é o MapR-FS que permite a reescrita do arquivo de entrada no processamento, porém, se isso ocorrer, a localidade dos dados só vai beneficiar o processo de reescrita no arquivo quando o processo for executado no nó com a cópia primária que é fixada no nó *master* da cadeia de replicação, reduzindo a localidade de dado para apenas um nó. O objetivo maior do MapR-FS é permitir o seu uso por aplicações tradicionais como um sistema de arquivos POSIX, porém a localidade de dados no processo de escrita não ocorre em todos os nós com réplicas, mesmo assim, a semântica mais elaborada do MapR-FS possibilitou o desenvolvimento de um banco de dados colunar, como o HBase do Hadoop, com um mecanismo que não precisa reescrever todos os dados para eliminar registros deletados.

Tanto o HDFS como o MapR-FS são desenhados para que o processamento ocorra buscando a localidade de dados, trazendo o processamento de map para o nó que armazena o dado, geralmente usando hardware de baixo custo onde a falha é algo esperado.

4 AWAREFS: BIG DATA, HPC OU APLICAÇÕES TRADICIONAIS NO MESMO SISTEMA DE ARQUIVOS

O sistema de arquivos proposto neste trabalho e descrito a seguir, chamado AwareFS (do inglês *Advanced Write Anywhere Read Everywhere File System*), permite que um mesmo bloco possa ser lido, gravado e regravado a qualquer momento e por qualquer um dos servidores com uma cópia do dado, provendo um variado sistema de locks e garantindo coerência dos dados entre as réplicas em um ambiente onde a falha seja a regra, de forma que a rede seja usada para trafegar os dados apenas nos casos onde o servidor solicitante realmente não tenha uma cópia do bloco em questão. Ainda que o processo de regravação seja através de um protocolo de escrita baseado em primário (TANENBAUM; STEEN, 2007), o papel de dono da cópia primária migra para o nó executando a escrita, evitando tráfego de dados para outro nó sempre que existir uma cópia local. O AwareFS deve ainda trazer benefícios como interface POSIX para possibilitar seu uso com aplicações tradicionais, além de uma maior distribuição no controle de metadados, reduzindo a ocorrência de ponto único de falhas com servidores concentrando esse trabalho.

O AwareFS é então um sistema de arquivos distribuído sem as limitações da semântica WORM, com gerenciamento distribuído de locks e com uma interface que também viabiliza aplicações tradicionais e de HPC. Além disso é um sistema de arquivos com uma arquitetura sem ponto único de falhas, que promove uso eficiente da localidade dos dados e que funciona com hardware de baixo custo.

A primeira versão do AwareFS, elaborada como parte deste trabalho, tem um foco voltado para o desempenho de operações de leitura e escrita, seguindo acessos sequenciais e randômicos, sem inicialmente provar capacidades de detecção e recuperação de falhas. Assim, nessa primeira versão já foram implementadas todas as operações voltadas à resiliência mas que causam impactos ao desempenho geral de E/S, como por exemplo a replicação de dados e a criação de pontos de recuperação

("*checkpoints*"), ficando para serem implementadas em uma próxima versão as capacidades de detecção e recuperação de falhas, como por exemplo, substituição de um servidor de dados.

4.1 Interface com o usuário

Seguindo o padrão de sistemas operacionais Unix (PATE; BOSCH, 2003), o AwareFS possui primitivas básicas do padrão POSIX (SPECIFICATIONS, 2008) como open, read, write, seek e close. A ideia é que o AwareFS seja o mais próximo possível de sistemas de arquivos convencionais. As primitivas se dividem entre operações para organização em diretórios e operações de arquivos.

Operações de diretórios:

- readdir(caminho, ...): essa função retorna cada uma das entradas do diretório especificado pelo parâmetro caminho;
- mkdir(caminho, modo): cria um diretório no caminho especificado com as permissões definidas pelo parâmetro “modo”;
- rmdir(caminho): remove o diretório especificado se ele estiver vazio.

Operações de arquivos:

- mknod(caminho, modo): cria um arquivo especificado por “caminho”;
- unlink(caminho): remove um arquivo;
- rename(antigo, novo): muda o nome do arquivo especificado por “antigo” para o nome definido em “novo”;
- open(caminho, flags): abre um arquivo com o modo especificado por “flags”;
- read(caminho, buffer, comprimento, início): grava em “buffer” um número de bytes (definido pelo parâmetro comprimento) lidos do arquivo especificado em “caminho”, a partir da posição especificada por “inicio”;

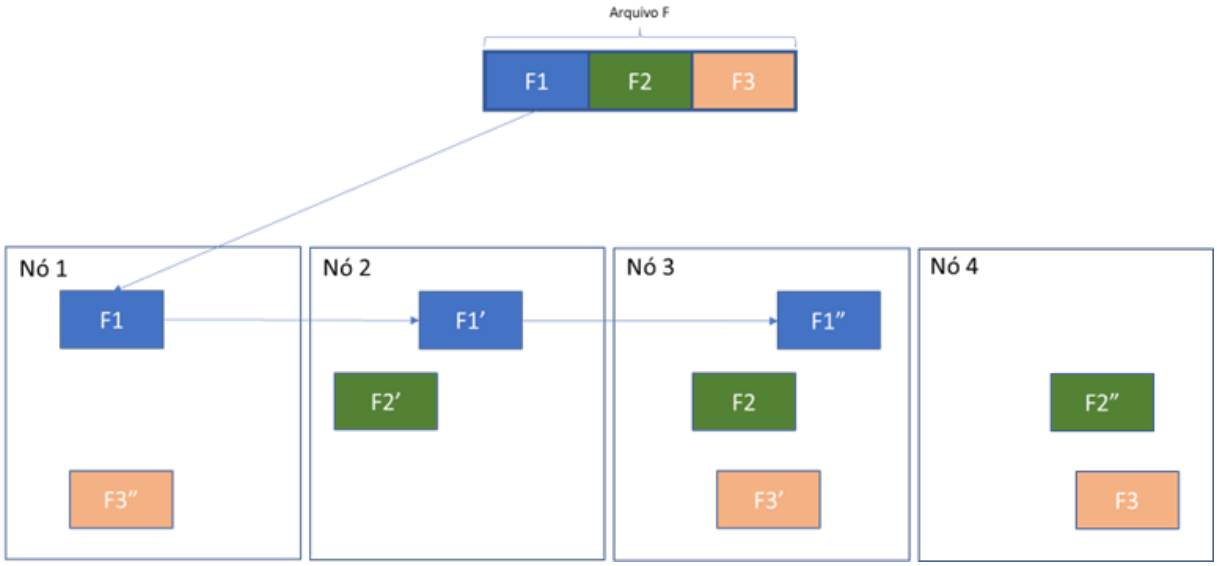

Figura 9 - Fragmentação e replicação de um arquivo de três chunks entre quatro DSs

- write(caminho, buf, comprimento, início) grava na posição “início” do arquivo especificado em “caminho” um número de bytes (definido pelo parâmetro comprimento) armazenados na área apontada por “buf”;
- truncate(caminho, len): corta o arquivo especificado por “caminho” na posição indicada pelo parâmetro “len”;
- flush(caminho): garante que todos os dados do arquivo indicado por “caminho” sejam persistidos em disco;
- release(caminho): usada em operações de close, essa operação encerra operações em andamento para o arquivo especificado.

4.2 Fragmentação e replicação de um arquivo

Os arquivos armazenados pelo AwareFS são divididos em partes menores chamadas chunks que são distribuídas nos nós de dados do cluster, que passam a gerenciar processos de leitura e escrita trocando informações com outros componentes do cluster para garantir consistência, coerência e desempenho. Cada chunk de um arquivo deverá ter três réplicas, cada uma armazenada em um nó diferente (Figura 9).

Para garantir consistência, em um dado momento, apenas um nó processa requisições de escrita para cada *chunk*: o nó dono do *chunk*. Considerando que o AwareFS usa um protocolo de escrita baseado em primário, o nó dono do *chunk* é aquele que está com a cópia primária do *chunk* naquele momento. Para diminuir o tráfego na rede de grandes volumes de dados, o papel de nó dono de um *chunk* pode ser transferido de um nó para outro. A organização dos dados e metadados, os processos de leitura e escrita, a replicação dos dados, o sistema de locks e o sistema de tolerância a falhas são descritos a seguir.

4.3 Organização dos dados em *containers*

Para garantir acesso distribuído eficiente e a redundância dos dados, o AwareFS divide os arquivos em partes chamadas *chunks* e agrupa-os em estruturas chamadas *containers* que são armazenadas nos servidores e replicadas para quantos servidores de cópia forem pré-definidos, sendo três o número default. Os *containers* têm um número máximo de *chunks* pré-definido, o que lhes garante um tamanho com ordem de grandeza várias vezes maior que o tamanho médio dos *chunks*. Para guardar a relação dos *containers* com seus respectivos servidores, o sistema de arquivos tem uma única estrutura que é a CL (Container List), ela é gerenciada pelo CS (Container Service). Embora a CL seja centralizada, outros servidores podem ter uma cópia idêntica dessa lista, o que permite que outro servidor assuma o papel de CS se necessário. Os servidores de dados (DS, do inglês Data Service) podem fazer *cache* da CL, diminuindo o número de consultas ao CS.

Um mesmo *container* é replicado em mais de um Data Service. A lista de DSs que possuem réplicas de um mesmo *container* é chamada “cadeia de replicação” do *container*. A Figura 10 ilustra um exemplo onde o arquivo F é dividido em três partes (três *chunks*), sendo que cada parte está armazenada em um *container* replicado três vezes, usando um conjunto de quatro Data Services. Essa divisão e organização dos dados garante ao AwareFS a possibilidade de manter cópias dos dados distribuídas em servidores diferentes, de forma que o dado possa ser lido e gravado de forma

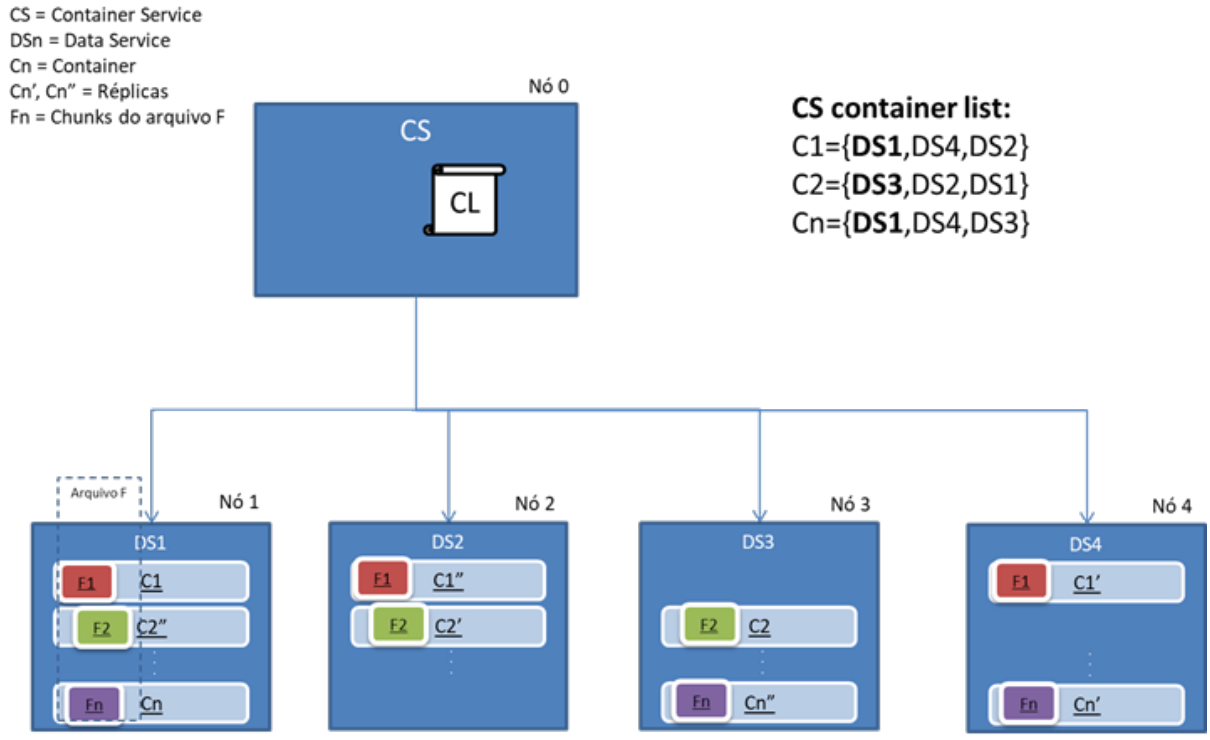

Figura 10 - Distribuição dos dados em *containers*

consistente em diversas localidades, característica importante para processamento baseado em MapReduce.

4.4 Arquitetura geral do AwareFS

Seguindo a ideia de usar uma arquitetura que elimine os pontos únicos de falha, o AwareFS é dividido de forma modular, em componentes que interagem entre si e podem ser executados em qualquer um dos nós do cluster.

Para um cluster AwareFS, em um mesmo nó físico é possível ter em execução tanto o controle de dados como o de metadados. Um nó físico pode ainda executar o componente cliente, responsável pelo processo de montagem do sistema de arquivos para leitura e gravação, como qualquer sistema de arquivos padrão POSIX. Na Figura 11 temos uma disposição onde um nó físico é dedicado aos serviços de metadados enquanto outros dois nós executam o serviço de dados e *lock* distribuído, juntamente com o componente cliente. A Figura 11 mostra os componentes responsáveis pelo

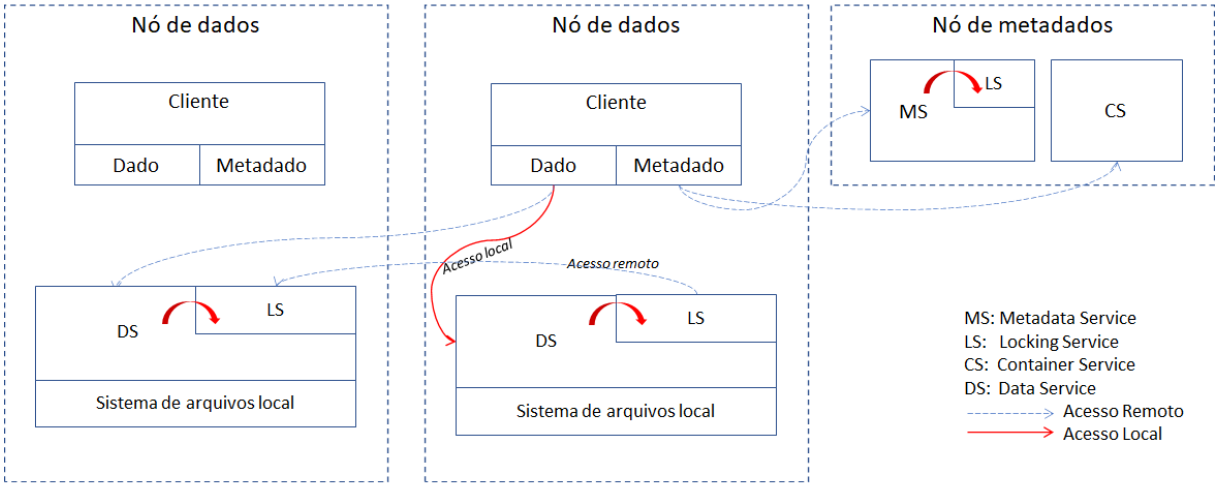

Figura 11 - AwareFS: Arquitetura geral

funcionamento do sistema de arquivos, são eles:

- **Client**: É o componente responsável por atender as solicitações de leitura e escrita da camada de aplicação. O clientAware é dividido em um controle específico para metadados e um outro responsável pelo acesso aos dados.
- **MS (Metadata service)**: É o serviço responsável por atender as solicitações de metadados iniciais, incluindo operações de *lock*. Operações iniciais de metadados são aquelas que obtém e gravam um identificador de arquivo a partir de um nome de arquivo com caminho ("*path name*"). O serviço de metadados negocia o acesso aos dados com o serviço de *lock* descrito a seguir.
- **LS (Locking service)**: É o serviço que atende as solicitações de *lock*, bem como conduz a negociação dos bloqueios com os demais LSs do cluster. O LS pode ser associado ao serviço de dados (DS) ou ao serviço de metadados (MS).
- **CS (Container service)**: O CS é o serviço responsável pela manutenção da lista de *containers* (CL: *container list*). O CS é o único serviço centralizado, contudo, por causa do tamanho pequeno e da frequência baixa de atualização da CL, o número de consultas a esse serviço tende a ser baixo, sem comprometimento de desempenho.

- **DS (Data service)**: É o serviço responsável por transformar as requisições de leitura e escrita em acessos ao sistema de arquivos local de cada nó do cluster. Operações de *lock* para garantir a consistência dos dados são direcionadas ao respectivo LS pelo DS. É o DS que faz a replicação dos dados para melhorar o desempenho e a resiliência.

Assim como acontece em outros sistemas de arquivos distribuídos baseados em clusters, o AwareFS distribui os serviços que o compõem em nós (servidores) diferentes. DS e LS são redundantes, de forma que se um nó de dados vier a falhar, o serviço passará a ser provido por outro nó. Embora haja um único nó de metadados, apenas uma lista inicial de objetos (arquivos e diretórios) é mantida pelo MS, sendo que as informações mais detalhadas de cada objeto são distribuídas nos nós de dados que compõem o cluster. Considerando que o CS é único mas tem seus dados replicados em outros pontos do cluster, sua recuperação pode ser implementada sem nenhuma complexidade. O MS, por sua vez, também não é um ponto único de falhas quando implementado com um banco de dados com características de alta disponibilidade, por exemplo, utilizando uma base de dados de tipo chave/valor implantada de forma redundante em dois ou mais servidores relativamente pequenos. Tanto CS como MS devem ser implementados com reinício automático, comandado pelos DSs e/ou clientes.

4.5 Semântica de leitura e escrita

Para o AwareFS, a semântica de leitura e escrita a ser oferecida é a seguinte:

Todos os dados ficam disponíveis para leitura tão logo o processo de escrita termine, garantindo que o dado tenha a versão definida pelo momento do fim da última escrita, independente do seu início.

A Figura 1 apresentada no capítulo 2 traz um exemplo em que um só *chunk* está envolvido, e tem-se que:

- A segunda leitura do client 1 teria C como resultado;
- A primeira leitura do client 2 teria A como resultado;
- A leitura do client 3 teria B como resultado.

A semântica, aqui oferecida, é garantida aos usuários do AwareFS adicionando uma informação de versão a todas as estruturas de dados e metadados. Numa operação de escrita e leitura concorrentes, por exemplo, a operação de escrita só atualizará os metadados depois de concluída todas as transações envolvidas, só aí as versões dos metadados serão alteradas para a versão dos *chunks* depois das alterações. A sincronização das operações de metadados será garantida por um conjunto de *locks* internos, transparentes para o usuário.

4.6 Organização dos metadados

Assim como em sistemas de arquivos tradicionais, a primeira parte da organização no AwareFS é a catalogação dos dados em pastas (diretórios) e arquivos (Figura 12). Cada diretório ou arquivo tem seus metadados, tais como data de acesso, tamanho, data de criação, etc., que são armazenados em estruturas chamadas inodes. Desta forma, no AwareFS para cada arquivo ou diretório tem-se um inode com os metadados inerentes. De forma simplificada, como os usuários referenciam os arquivos e diretórios pelo nome completo (i.e. *pathname*) o sistema de arquivos mantem um índice relacionando cada *pathname* com seu respectivo inode.

No AwareFS o índice relacionando os *pathnames* com seus respectivos inodes é controlado inicialmente por um serviço centralizado, o MS (Metadata Service), sendo que a maior parte dos metadados fica nos inodes que são armazenados e controlados de forma distribuída, assim como é feito para os *chunks*.

Para controlar o armazenamento distribuído dos inodes e *chunks*, o MS mantem um índice relacionando cada pathname com uma estrutura única chamada FID que contém o identificador do *container* usado para armazenamento, além do número do

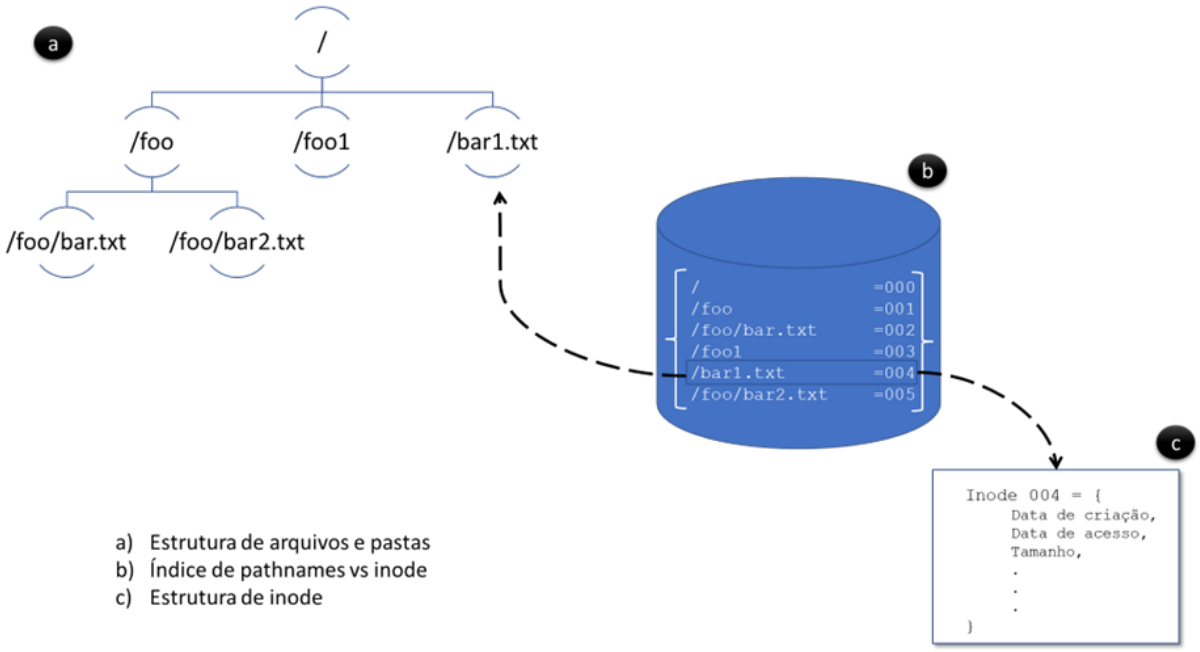

Figura 12 - Organização de arquivos e catalogação com inodes

inode em si e um número de versão, assim cada registro tem como campos:

- Pathname: o nome completo e único, referenciado pelo usuário, que indica o nome do arquivo ou diretório e seus "ancestrais", e.g. "/home/user1/controle/arq1.txt"
- FID: (identificador de arquivo, do inglês "*file indentifier*") que é uma estrutura com:
  - CID: identificador do *container* com o *inode* do arquivo;
  - Inode#: número do *inode* ou chunk. Para o AwareFS cada item armazenado em um *container* recebe um número único como identificador, seja ele um inode ou um chunk;

- versão: identificador modificado quando o *inode* é reutilizado. Seu objetivo é garantir que o dado refletido pelo *inode* seja o mesmo referenciado nos metadados. O número de versão é atualizado sempre no MS e no *inode* e a leitura ou escrita deve falhar se a versão não coincidir em ambos os lugares, o que pode acontecer em caso de falhas ou se a cópia do *inode* em uso estiver desatualizada. Esse mecanismo deve garantir que o dado apresentado para o usuário a partir dos metadados seja sempre consistente.

Os arquivos maiores podem ser divididos em *chunks* que também são identificados por estruturas FID e neste caso o Inode# é um número único usado para identificar o chunk. Dessa forma, além das informações básicas já descritas, o *inode* do arquivo possui um vetor com os FIDs de cada um dos *chunks* que compõem o arquivo. A Figura 13 mostra os vetores de FIDs para um arquivo F composto por três *chunks* e para um arquivo G com dois *chunks*. É importante ressaltar que o FID pode apontar para um *chunk* armazenado em qualquer *container* que eventualmente poderá estar em qualquer nó do cluster, aumentando assim as possibilidades de distribuição dos dados. A única exceção é o primeiro *chunk*, que fica no mesmo servidor que o *inode* do arquivo, para aumentar o desempenho de operações com arquivos pequenos, já que a leitura/escrita dos dados armazenados é feita usando o mesmo servidor que respondeu à leitura/escrita dos metadados armazenados no *inode*.

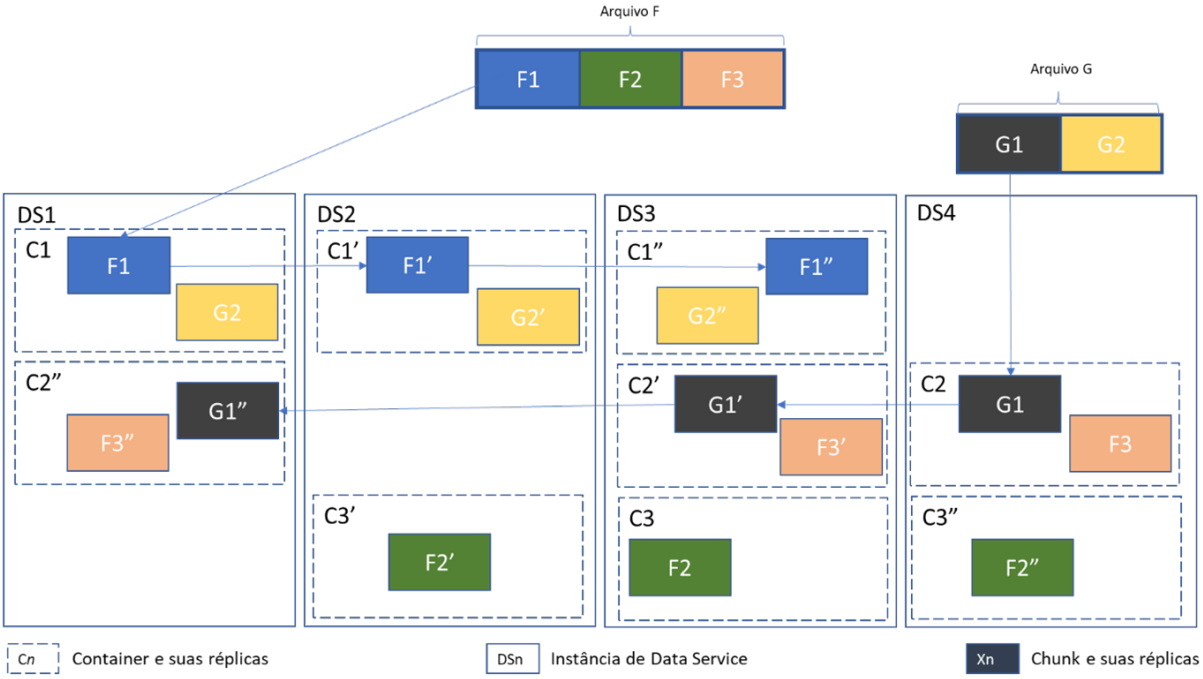

Figura 13 - Arquivos divididos em chunks, armazenados em *containers* e replicados

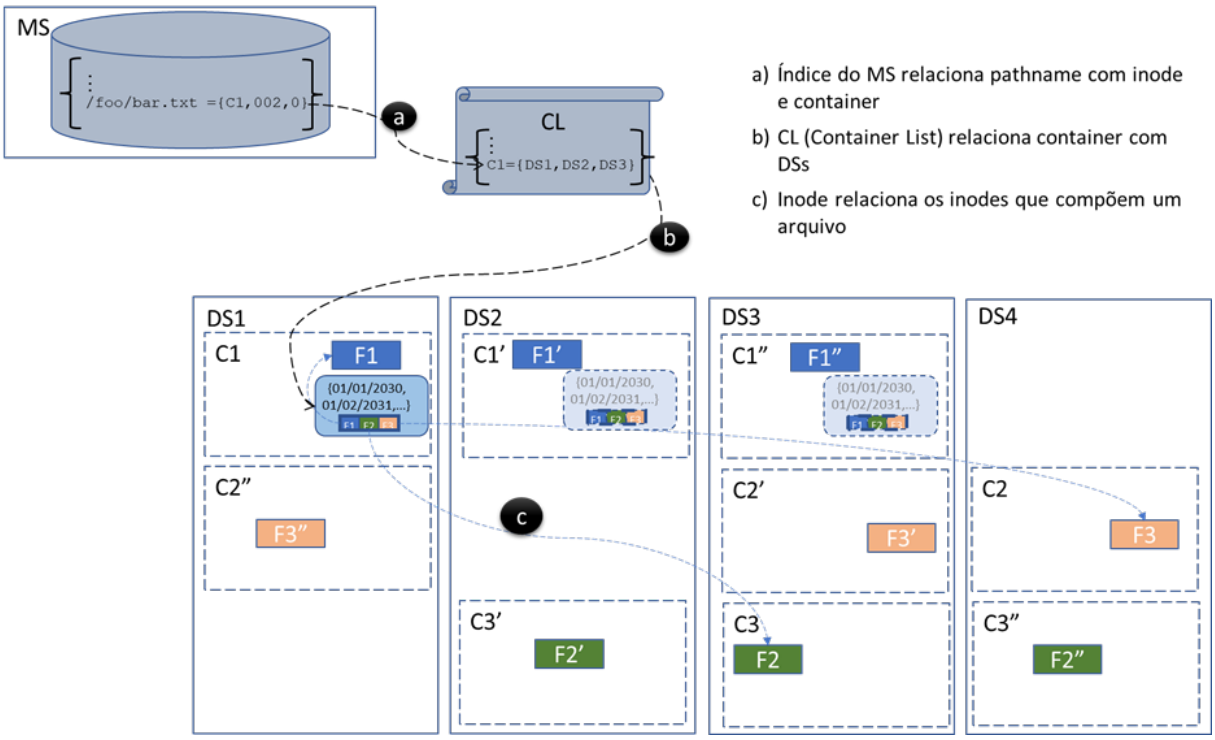

Figura 14 - Relacionamento de estruturas de metadados do AwareFS

A cadeia de estruturas que compõem os metadados usados pelo AwareFS é como segue (Figura 14):

- a) Base do MS com os FIDs de todos os pathnames que compõem a árvore do sistema de arquivos;
- b) CL, ou *container list*, com todos os DSs que armazenam cópias de cada *container*. Embora gerenciada pelo CS, é feito um cache dessa lista em todos os DSs e clientes, sendo que sua atualização só é necessária para um eventual primeiro acesso a dados em novos *containers*.
- c) Inodes armazenados de forma distribuída em *containers*.

Para criar um arquivo ou armazenar novos dados, uma solicitação é enviada ao Data Service que vai retornar um FID referente ao armazenamento depois de identificar um *container* com espaço disponível ou providenciar a criação de um novo *container*. Como exemplo, a criação do arquivo F na Figura 13 deve resultar na criação de um inode armazenado no *container* C1 (mesmo *container* do primeiro *chunk*) e seu FID é retornado como resultado da operação. Para acessos posteriores via *pathname*, o índice controlado pelo MS é atualizado com uma entrada relacionando o *pathname* do arquivo F com seu FID. No inode do arquivo F estará a lista dos FIDs dos chunks que o compõem, para que estes sejam usados para acessos aos dados. A Figura 15 descreve a sequência de passos para que um cliente do AwareFS possa ter acesso a um determinado chunk de um arquivo definido por seu *pathname*, são eles:

1. Com uma chamada ao MS, o cliente obtém o FID do inode do arquivo desejado usando o seu pathname.
2. Com a CL previamente armazenada, o cliente obtém o DS a ser utilizado usando o ID do *container*. Para operações de leitura, o DS poderá ser qualquer um dos DSs relacionados desde que possua uma réplica válida. Operações de escrita devem ser feitas no DS dono do *chunk*.

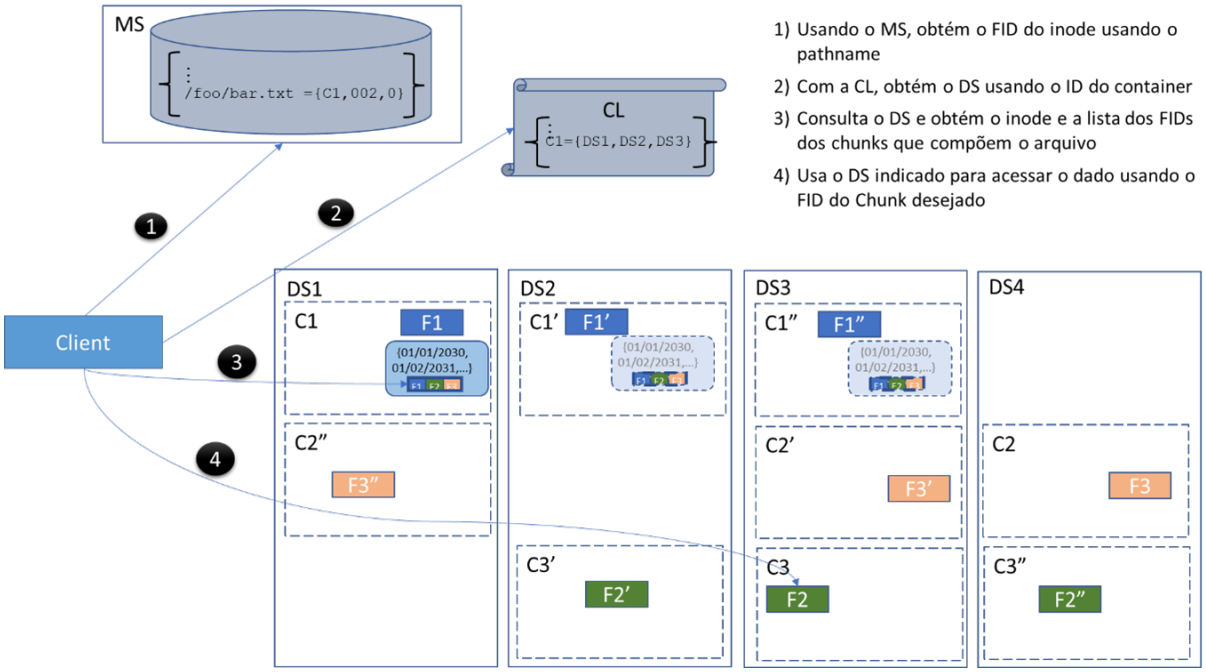

Figura 15 - Sequência de passos para o acesso de dados e metadados no AwareFS

3. O cliente então consulta o DS determinado e obtém o inode e a lista dos FIDs dos chunks que compõem o arquivo.

4. Com a informação da posição desejada dentro do arquivo, o cliente identifica qual é o *chunk* desejado (segundo *chunk* no caso da Figura 15) e usa o DS indicado para acessar o dado usando o FID de tal *chunk*.

4.7 Armazenamento dos dados

Como dito anteriormente, os dados são logicamente organizados em estruturas de controle chamadas "*containers*" que são armazenadas usando os recursos básicos do sistema operacional de cada máquina que compõe o cluster. Dessa forma, a primeira versão do sistema de arquivos aqui proposto deve fazer uso de componentes (diretórios e arquivos) dos sistemas de arquivos locais dos servidores.

Para armazenar os dados, cada *container* será um diretório do sistema de arquivos básico do servidor, nomeado da seguinte forma:

<CID>.<versão>: onde CID é o identificador único do *container* e versão é um

inteiro incrementado de forma sequencial.

Um *container* pode armazenar itens com dois tipos de informações: *inode* com metadados ou chunks com dados. Todas as atualizações dos dados armazenados no *container* são controladas por atributos que indicam se o item, seja ele *inode* ou *chunk*, está válido e qual o servidor está controlando suas atualizações, isto é, qual é o seu dono. Esses atributos ficam em um mapa armazenado em um arquivo único por *container* e sincronizado entre as cópias (veja item 4.8.1 abaixo).

O *inode* com metadados de um arquivo é armazenado em um arquivo de cabeçalho e, como dito no item 4.6, nele estará a lista dos FIDs referenciando arquivos com os dados propriamente ditos. Os arquivos de cabeçalho são referenciados doravante como inodes e os arquivos com os dados são referenciados como chunk. Os arquivos no sistema operacional dos nós do cluster são nomeados da seguinte forma:

h.<inode#>.<versão>.<idx#> para inodes, onde:

- inode# é o número do *inode*;
- versão é o número de versão do *container*, no momento da última atualização do conteúdo do arquivo de cabeçalho;
- idx# é o índice do *inode* na tabela de controle de atualizações do *container*.

c.<node#>.<versão>.<idx#> para chunks, onde:

- node# é o número do *chunk*;
- versão é o número de versão do *container*, no momento da última atualização do conteúdo do arquivo do *chunk*;
- idx# é o índice do *chunk* na tabela de controle de atualizações do *container*.

m.<CID>.<versão> para armazenar o mapa de controle do *container*, onde:

- CID é o identificador do *container*;

Configuração de arquivos e diretórios de um DS exemplo onde:

- Cada coluna é um diretório no sistema de arquivos do servidor do DS
- Cada célula é um arquivo do sistema de arquivos do servidor do DS
- O inode 102 e o chunk 103 foram alterados na versão 1 do container 1
- O chunk 205 foi alterado na versão 1 do container 2
- O container 3 foi criado e contém o inode 301 e o chunk 302

C1.0	C2.0	C1.1	C2.1	C3.0
m.1.0	m.2.0	m.1.1	m.1.1	m.3.0
h.101.0.1	c.201.0.1	h.101.0.1	c.201.0.1	c.302.0.1
h.102.0.2	c.202.0.2	h.102.1.2	c.202.0.2	h.301.0.2
c.103.0.3	h.203.0.3	c.103.1.3	h.203.0.3	
c.108.0.4	c.205.0.4	c.108.0.4	c.205.1.4	

Figura 16 - Exemplo de diretórios e arquivos criados no sistema de arquivos local de um DS

- versão é o número de versão do *container*.

4.8 Consistência e replicação

Tal qual o HDFS e outros sistemas de arquivos distribuídos, o AwareFS mantém múltiplas cópias do mesmo dado em servidores diferentes, ou seja, um mesmo *container* é replicado em pelo menos três servidores. O objetivo de manter várias cópias de um mesmo dado é garantir uma redundância que possibilite maior desempenho de leitura e escrita usando um método de armazenamento com tolerância a falhas. É claro, porém, que a semântica promovida pelo sistema de arquivos deve garantir que os dados sejam gravados e lidos de uma forma que mantenha a consistência dos dados. Assim como em sistemas de memória compartilhada, a semântica adotada pelo AwareFS impõe a forma como os dados são retornados em um processo de leitura, garantindo que a última atualização seja retornada independente de quantos processos estejam trabalhando com as várias cópias de um mesmo dado (veja mais sobre a semântica do AwareFS no item 4.2 acima) (ADVE; GHARACHORLOO, 1996).

Quanto à consistência, a estratégia adotada para o AwareFS é emular o comportamento de sistemas monoprocessados, utilizando métodos que garantam a consistência sequencial das atualizações dos dados, ou seja, o programador ou usuário observa o resultado esperado segundo a ordem em que as operações são executadas sobre o dado (ADVE; GHARACHORLOO, 1996).

Considerando que um mesmo dado deve ser armazenado em pelo menos três servidores, faz-se necessária também uma definição de mecanismos referenciados como “protocolos de coerência de *cache*” para garantir que a atualização de um dado seja propagada para todas suas réplicas e que as cópias sejam marcadas como inválidas enquanto estiverem desatualizadas para que não sejam retornadas assim em um processo de leitura (ADVE; GHARACHORLOO, 1996),(TANENBAUM; STEEN, 2007).

A seguir são detalhados os cuidados e métodos adotados no AwareFS para garantir a replicação dos dados, mantendo a consistência, em uma configuração que tolere falhas.

4.8.1 Controles de consistência e replicação

Ao armazenar um dado em um *container* o AwareFS busca replicar esse dado para todos os outros *containers* da cadeia de replicação (veja item 4.3 acima). Para que isso aconteça respeitando uma consistência sequencial os dados são divididos em partes menores chamadas *chunk*:

- **Chunk**: divisão lógica de um arquivo com tamanho máximo configurável (default: 64MB).

Para controlar a consistência são definidos e manipulados os seguintes atributos para cada chunk armazenado em um *container*:

- **Owner**: atributo com o identificador único do DS responsável pelas operações de escrita de um chunk. O papel de *owner* pode migrar de um DS para outro desde que ambos façam parte da cadeia de replicação do *container*.

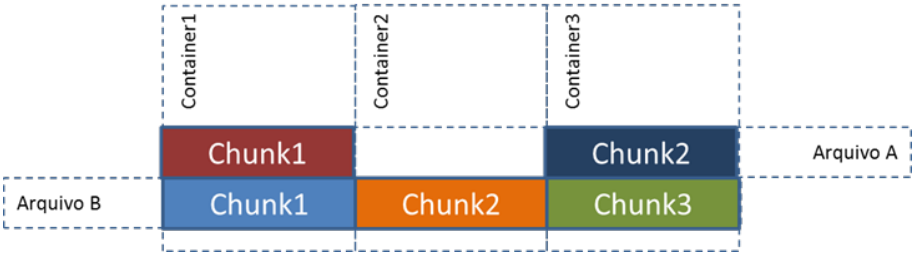

Figura 17 - Divisão de arquivos em chunks e organização em *containers*

- **Invalid**: atributo que indica se a cópia do *chunk* armazenada no *container* foi invalidada (completa ou parcialmente), ou seja, foi atualizada em um outro DS da cadeia de replicação. Esse atributo terá os valores 1 quando o chunk estiver completamente inválido, 0 (zero) quando estiver completamente válido ou um número negativo, indicando que o chunk tem regiões inválidas e o módulo do valor armazenado em Invalid indica quantas invalidações por região ocorreram.

Os parâmetros de *owner* e *invalid* de cada *chunk* são armazenados no mapa de controle do *container*, que é uma área de memória controlada pelo DS e persistida no arquivo com o mapa de controle do *container*, ou seja, o arquivo de tipo m. O papel do DS *owner* do *chunk* é central para o AwareFS, dado que a consistência sequencial é garantida usando um protocolo baseado em primário onde as atividades de escrita devem ser dirigidas a um único processo (TANENBAUM; STEEN, 2007). Para o AwareFS qualquer outra operação que dependa fortemente da ordem de execução (e.g. operações de bloqueio) também deve ser direcionada para o DS do nó com a cópia primária do *chunk*, ou seja, o *owner* do *chunk*. Em ambientes baseados em cópia primária, quando o dado é replicado em vários nós, todas as escritas são direcionadas a um só nó (Figura 18). Em ambientes para processamento distribuído baseados em map e reduce como o Hadoop, o sistema tenta direcionar as tarefas para nós do cluster que tenham os dados de entrada armazenados localmente. Desta forma, o AwareFS busca promover o uso do dado armazenado localmente em cada nó, não só para os dados de entrada sendo lidos pelos processos, mas também utiliza um protocolo de escrita local, transformando a cópia local em cópia primária para que não seja necessário transferir dados para um nó remoto ao regravar um dado, evitando

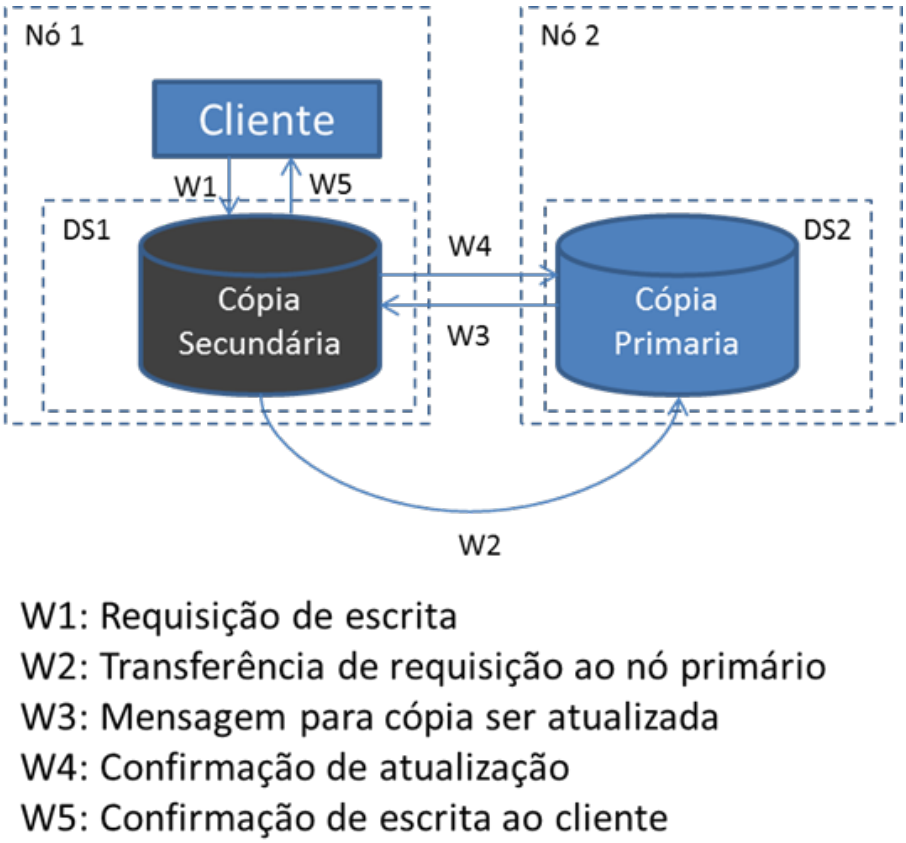

Figura 18 - Atualização de dado replicado em ambientes baseados em cópia primária

que todo o volume de trocas de mensagens e transferência de dados da Figura 18 se repita seguidas vezes (veja item 4.13).

Por outro lado, a flag *invalid*, que indica que a cópia do chunk está ou não inválida, é central no mecanismo para a garantia de coerência de cache que permite que as várias réplicas possam ser usadas para leitura. Cada DS tem uma flag de *invalid* para cada chunk e estas flags são armazenadas no mapa de controle do *container*, gerenciado pelo DS.

4.8.1.1 Invalidação por região

Considerando que um *chunk* tem tamanho relativamente grande (na ordem de megabytes), invalidar um *chunk* inteiro mesmo para pequenas reescritas seria ineficiente. Para que um DS possa receber apenas as regiões alteradas de um *container* um DS possui uma lista (InvalidRegList) com os seguintes dados de cada regravação parcial

de um *chunk*:

- CID: *container* ID do *chunk* alterado;
- Inode#: Número do *chunk*;
- NewVers: Versão do *chunk* depois de atualizado;
- OldVers: Versão do *chunk* antes da alteração;
- Offset: Posição do início da região alterada dentro do *chunk* (a partir do início do *chunk*);
- Length: Comprimento da região alterada dentro do *chunk*.

Assim, para os casos em que os chunks sofrem alterações parciais, o DS que for dono do *chunk* deve enviar aos demais DSs da cadeia de replicação uma mensagem com os dados a serem inseridos na InvalidRegList, solicitando que seja invalidada apenas a sub-região alterada e, nesses casos, a flag de Invalid do mapa de controle recebe o valor -1 indicando que o *chunk* está parcialmente inválido (apenas uma sub-região do *chunk* precisa ser atualizada).

Uma invalidação de *chunk* por região só pode ser de dois tipos:

1. Inicial: é o caso em que o *chunk* recebe uma primeira alteração depois de estar inteiramente atualizado, ou seja, depois de estar com a versão igual a da cópia primária armazenada no owner do *chunk*.
2. Subsequente: é o caso em que o *chunk* recebe uma atualização que não é inicial, porém foi recebida após a chegada ordenada de todas as invalidações por região que tenham ocorrido depois de uma invalidação por região do tipo inicial, sendo a ordem definida pela versão do *chunk*.

Se uma invalidação de *chunk* por região não se enquadrar em nenhum desses dois casos, será considerada uma invalidação fora de sequência e então o *chunk* todo será considerado inválido e deverá ser replicado inteiro, não por região.

Para evitar que cada mensagem de invalidação deva ser transmitida com garantia de entrega, o AwareFS verifica a cada mensagem de invalidação se a sequência dessas mensagens não foi quebrada, ou seja, se as mensagens de invalidação são iniciais ou subsequentes. Caso a sequência de mensagens de invalidação seja quebrada significa que as mensagens de invalidação por região não foram todas recebidas e então o *chunk* inteiro deverá ser retransmitido. Para fazer essa verificação cada *container* armazena uma lista chamada CInodes com o identificador de cada *chunk* e a versão armazenada para ele, ou seja, a versão corrente do chunk. Assim, a sequência de mensagens de invalidação é verificada da seguinte forma:

- A cada mensagem de invalidação recebida, o DS identifica se trata-se de uma invalidação por região inicial ou subsequente:
  - Inicial: o campo OldVers na mensagem é igual a versão atualmente indicada para o *chunk* em questão na lista CInodes;
  - Subsequente: o campo OldVers na mensagem é igual a o campo NewVers de uma mensagem presente na InvalidRegList para o *chunk* em questão.

Se a verificação falhar significa que houve perda na sequência de invalidates, então o *chunk* inteiro é invalidado (a flag de *invalid* no mapa é marcada com 1) e as entradas para esse *chunk* na InvalidRegList são eliminadas.

Na Figura 19, por exemplo:

- O nó 2 aceitou a região “write 2”, pois a versão indicada na mensagem de escrita (NewVers) era 2 e a versão corrente indicada para o *chunk* era 1. Assim a flag Invalid ficou com -1 indicando a existência de invalidação por região;
- O nó 2 aceitou a região “write 3”, pois a versão indicada na mensagem de escrita (OldVers) era 2, idêntica ao NewVers da última região aceita com a mensagem acima;
- O nó 3 não aceitou a região “write 3” pois a versão NewVers era 3, a versão corrente indicada para o *chunk* era 1 (e não 2) e não havia uma mensagem de

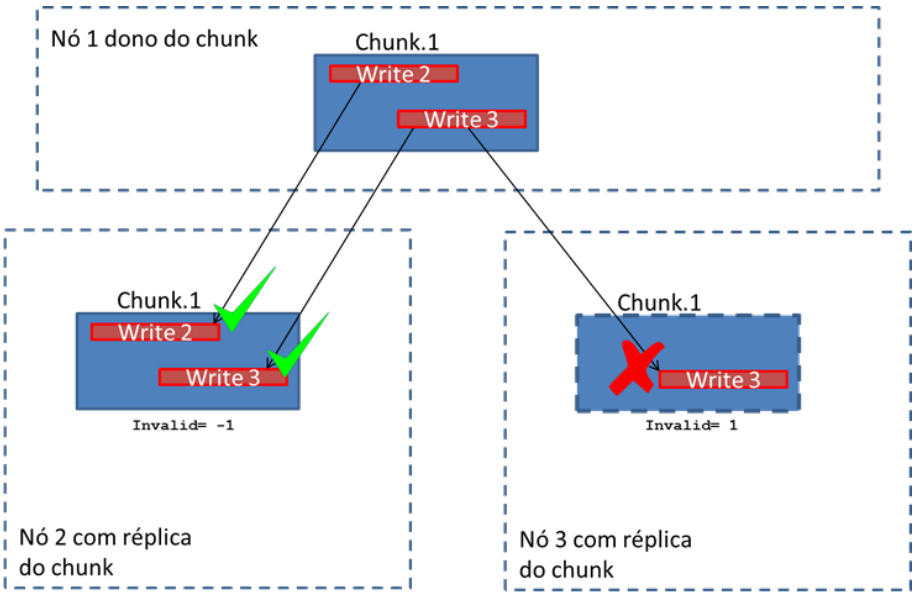

Figura 19 - Invalidação por região

atualização de região para a versão 2. Por isso, o *chunk* inteiro foi invalidado, mudando a flag de Invalid para 1.

4.8.1.2 Replicação

Para garantir que todo DS membro da cadeia de replicação tenha uma cópia válida do dado, um procedimento executado em segundo plano identifica as cópias locais dos *chunks* que tenham sido marcadas como inválidas (invalid=1) e requisita ao DS *owner* de cada chunk que envie os dados atuais armazenados na cópia primária. Se o *chunk* tiver sido marcado como parcialmente invalido (invalid=-1) é solicitado ao DS *owner* que os dados em cada uma das regiões armazenadas em InvalidRegList sejam transmitidos. Uma vez recebido os dados, a região correspondente é removida da InvalidRegList. Nos casos em que existirem várias alterações subsequentes de regiões com sobreposição, as regiões serão combinadas para diminuir o número de mensagens entre o DS de réplica e o *owner* do *chunk*, ou seja, será solicitada e recebida apenas a região resultante da combinação das regiões com sobreposição.

4.8.2 Controle da cadeia de replicação

Cabe ao CS garantir que a cadeia de replicação está completa, para esse fim o CS controla as seguintes atividades:

- **Cadastramento de um novo *container***: Sempre que um novo *container* for criado, o CS deve montar a cadeia de replicação, designando quais serão os DSs que terão cópias do *container* e qual será o *master* da cadeia de replicação. A distribuição inicial do papel de owner dos chunks entre os DSs é feita em "round-robin".

- **Remoção de um DS da cadeia de um *container***: O DS *master* possui controles que o permite determinar se um DS está indisponível. Assim, um DS é marcado como indisponível pelo DS *master* da cadeia de replicação da qual ele faz parte. Nesse momento o DS *master* envia mensagem ao CS solicitando que o DS seja "descadastrado" da cadeia do *container*.

- **Admissão de um DS na cadeia de um *container***: O CS mantém uma lista com os *containers* que estão descobertos (com menos réplicas que o mínimo estipulado). Sempre que um DS inicia suas atividades ele anuncia para o CS quais os *containers* que ele possui com suas respectivas versões. O CS usa essa informação para identificar se o DS é um bom candidato para recompor a cadeia de replicação, se esta estiver descoberta. O CS envia mensagem ao *master* da cadeia de replicação, indicando que o DS identificado deve ser admitido na cadeia. O DS *master* inicia o processo de ressincronização seguindo o descrito no item 4.8.3.3.

Quando um DS fica indisponível para mais de uma cadeia de replicação (i.e., quando mais de um *container* fica descoberto), o CS vai receber mensagens de mais de um DS *master*, solicitando a remoção do DS indisponível de sua cadeia. Se isso ocorrer, o CS pode remover proativamente o DS indisponível de todas as cadeias de replicação para diminuir a troca de mensagens.

É esperado que não sejam frequentes as atualizações da lista de *containers* (CL) e respectivas cadeias de replicação, o que possibilita que a CL seja armazenada em cache em cada DS. Assim, cabe ao CS invalidar os caches dos DSs sempre que necessário.

4.8.3 Replicação de dados baseada em pontos de sincronismo (Checkpoints)

Em situações em que o DS fica incomunicável as operações de escrita precisam ser honradas por outro DS com uma cópia válida do dado. Contudo, se isso ocorrer, o DS que ficou fora de operação não terá mais um mapa de controle que garanta a coerência ou consistência das operações. Assim, é necessário que haja um mecanismo para que o DS volte a um estado sincronizado com os demais membros da cadeia, ou para que ele seja definitivamente substituído por outro DS.

O AwareFS possui a figura do "DS *master* de replicação" para controlar o processo de eliminação de um DS da cadeia de replicação e/ou para controlar o processo de restauração do sincronismo necessário entre os DSs da cadeia de replicação. Um DS pode ser *master* de um *container* e não ser de outro.

4.8.3.1 Criação de pontos de sincronismo (checkpoints)

Como dito no item 4.7 acima, cada *container* tem seus dados armazenados em um diretório do sistema de arquivos básico de cada servidor da cadeia de replicação, sendo assim, a estratégia adotada para armazenamento de *checkpoints* (pontos de sincronismo) é bloquear a escrita ao diretório em um dado momento, simultaneamente para todos os DSs que constituem a cadeia de replicação, transferindo as escritas posteriores para um novo diretório. Caso haja uma nova escrita em um *chunk* presente no *checkpoint*, esta será feita com um mecanismo de "*copy-on-write*" (PATE; BOSCH, 2003), isto é, o diretório bloqueado ganhará um arquivo onde serão inseridos os dados necessários para reverter as alterações em uma situação em que seja necessário reestabelecer o ponto de sincronismo. Esse arquivo é chamado de "*undo log*". Com

esse mecanismo, é garantido que os dados contidos no diretório bloqueado estão replicados de forma consistente para todos DSs da cadeia de replicação.

A criação de um *checkpoint* é feita como segue:

1. O DS *master* da cadeia de replicação envia mensagem aos demais DSs da cadeia comandando a criação do *checkpoint*;
2. Os DSs respondem informando que as operações de escrita serão bloqueadas;
3. O DS *master* envia o novo número de versão do *container* para os demais DSs;
4. Os DSs (incluindo o *master*) mudam o diretório atual do *container* para read-only, criam o *undo log* e criam um outro diretório para o *container* com o número de versão indicada pelo *master*. O novo diretório possui um hardlink, para cada arquivo do diretório original;
5. Os DSs criam uma cópia dos mapas de controle do *container* no novo diretório;
6. Os DSs respondem para o *master* que a criação foi bem-sucedida e que as escritas podem seguir;
7. O *master* envia uma mensagem para os DSs da cadeia informando que o *checkpoint* foi criado e que as novas escritas estão liberadas.
8. A partir daí, enquanto houver o diretório do *checkpoint* (diretório bloqueado), todas as operações de atualização dos chunks vão ser precedidas por uma leitura da área afetada. O DS grava então no *undo log* os seguintes dados:
    - a. Inode#: Número do *chunk*;
    - b. NewVers: Versão do *chunk* depois de atualizado;
    - c. OldVers: Versão do *chunk* antes da alteração;
    - d. Offset: Posição do início da região alterada dentro do *chunk* (a partir do início do *chunk*);
    - e. Length: Comprimento da região alterada dentro do *chunk*.

- f. Buffer: Dados originalmente na área afetada.

Dessa forma, todos os DSs possuem o mesmo conteúdo para os mapas de controle do diretório do *checkpoint*, garantindo a consistência entre os DSs, para aquele *container*, naquele ponto do tempo e as novas escritas passam a ser feitas em um diretório a parte, sempre gerando um log de alterações que possibilita a reversão de operações para reestabelecer um ponto de sincronismo.

4.8.3.2 Checkpoints recorrentes

O DS *master* da cadeia de replicação comanda a criação de um novo *checkpoint* em intervalos fixos. Pelo menos um *checkpoint* é mantido para garantir que o sincronismo entre os DSs da cadeia de replicação de um *container* possa ser reestabelecido. Em cada DS, um procedimento em segundo plano se encarrega da remoção de *checkpoints* antigos.

4.8.3.3 Recuperação e retomada do sincronismo a partir de um checkpoint

Em situações de indisponibilidade de um DS, o processo de recuperação da falha desencadeado vai colocar outro DS no lugar do DS que falhou. Em um caso assim, o novo DS deverá sincronizar os dados armazendado no container a partir das outras réplicas na cadeia de replicação. O processo de sincronismo pode ser acelerado se o DS sendo admitido for o mesmo que foi declarado indisponível. Sempre que o sincronismo precisar ser retomado, usando as informações armazenadas nos demais DSs da cadeia de replicação, um DS pode reconstruir o diretório do *container*. Supondo o *container* C100 com a cadeia de replicação DS1, DS2, DS3, onde:

- DS1 é o *master*;
- DS1 e DS3 estejam sincronizados, gravando a versão B do *container* (diretório C100.B);

- DS2 esteja desatualizado, porém com um *checkpoint*, para a versão A (diretório C100.A);

O DS2 pode restabelecer o sincronismo através dos seguintes passos:

1. DS1 (*master* da cadeia) envia mensagem aos outros DSs da cadeia informando sobre o início do processo de reconstrução e bloqueando a atualização do *owner* dos *chunks*;
2. DS2 remove o diretório C100.B (caso esse exista) e aplica todas as alterações registradas no undo log de C100.A.
3. DS2 cria diretório C100.B.wrk para reconstruir a réplica da versão B do *container*;
4. DS2 cria o mapa de controle m.C100.B em C100.B.wrk com invalid=1 e owner=-1 para todos os *chunks*;
5. DS2 solicita para DS1 a lista de *chunks* em C100.B e o mapa de controle m.C100.B;
6. DS2 atualiza a flag de *owner* no mapa criado em C100.B.wrk para todos os *chunks* usando o mapa passado por DS1;
7. Para acertar os estado de *chunks* presentes na versão A (DS2->C100.A) e que não foram alterados, DS2 analisa o mapa DS1->m.C100.B e a lista de *chunks* em DS1->C100.B e identifica todos os *chunks* com versão igual ou anterior a A e invalid=0, sendo que para cada *chunk* identificado é feito:
    - a. Criação de hardlink em C100.B.wrk apontando para o arquivo pré-existente em C100.A;
    - b. Atualização da flag de *invalid* (invalid=0) no mapa de controle sendo construído (DS2->m.C100.B);
8. Os passos 5 e 7 acima são repetidos para os demais DSs da cadeia, ou até que não haja mais nenhum chunk com invalid=1;

9. DS2 renomeia o diretório C100.B.wrk para C100.B e envia mensagem ao *master* da cadeia (DS1) informando o final do processo;

10. DS1 envia mensagem aos outros DSs da cadeia informando sobre o final do processo de reconstrução e desbloqueando a atualização do *owner* dos *chunks*.

Ao final do processo acima, o DS2 tem as flags de owner e invalid com os valores corretos para todos os *chunks* exceto para chunks cujo *owner* era o próprio DS2 e que estavam com invalid=1 em todos os demais DSs, nesses casos o *chunk* é marcado como corrompido, pois seu último estado não poderá ser recuperado.

Com o mapa de controle em um estado consistente, o processo em execução em segundo plano para atualização de cópias invalidadas pode ser retomado normalmente.

4.9 Tolerância a falhas

Sistemas de arquivo para ambientes onde “a falha é a regra” é imprescindível que todos os componentes sejam redundantes e que haja mecanismos para identificação de falhas e substituição automática desses componentes. Assim, a seguir destacamos as características de tolerância a falhas para o DS que garantem o funcionamento contínuo da cadeia de replicação. Inicialmente no AwareFS, o MS (Metadata Service) será centralizado, implementado de forma simplificada, sem uma preocupação com sua redundância, uma vez que ele será responsável por armazenar apenas o pathname do arquivo (nome com diretórios) e o FID do arquivo (informação de CID,Inode#,IDUnico).

4.9.1 Detecção de falhas

Os componentes do sistema de arquivo distribuído (veja Figura 11) interagem via troca de mensagens, o que faz com que, além de falhas de hardware e software dos nós do cluster, falhas de rede causem indisponibilidade de componentes de software. Considerando que a indisponibilidade de componentes como o DS pode prejudicar o

acesso apenas à parte dos dados, é importante que haja mecanismos para detecção também de falhas parciais, de forma a viabilizar o pronto reestabelecimento do sistema.

Podemos dividir as falhas consideradas no desenvolvimento do AwareFS em transientes, intermitentes e permanentes (TANENBAUM; STEEN, 2007).

- **Falhas transientes**: São as falhas que ocorrem por poucos instantes e desparecem (e.g. falhas de comunicação em rede). Para esse tipo de falha, em toda troca de mensagens entre componentes, por exemplo entre DS e CS, é esperado um tempo de time-out e a mensagem é reenviada se não houver resposta nesse período.

- **Falhas intermitentes**: São falhas que aparecem, desaparecem e reaparecem sem que a causa-raiz seja clara e/ou facilmente identificada. Para esse tipo de falha, a redundância dos componentes do AwareFS deve garantir o funcionamento contínuo do sistema, sendo que o componente falho será substituído de forma automática.

- **Falhas permanentes**: São as falhas que, uma vez iniciadas, só terminam se o componente causador for substituído. Assim como para falhas intermitentes, no caso do AwareFS o componente em falha deve ser substituído automaticamente.

A detecção de falhas no AwareFS é feita através da verificação da saúde do processo de trocas de mensagens entre seus componentes. Além disso, o CS pode enviar uma mensagem de verificação a qualquer componente para identificar se ele está operacional.

A seguir é apresentada uma descrição do mecanismo de detecção de falha de cada componente:

- **Detecção da falha do CS**: Todos os DSs trocam mensagens com o CS para criação de *containers* e reorganização da cadeia de replicação de um *container*.

Uma falha na troca de mensagens com o CS indicará a falha dele. O MS poderá receber mensagem para verificar se o CS está em execução.

- **Detecção de falha do DS**: Para identificar a falha de um DS (que não seja o *master* da cadeia), o DS *master* da cadeia de replicação possui as seguintes variáveis de controle:
  - **failCount**: para cada erro de comunicação entre um DS e outro (e.g. time-out na troca de mensagens entre DSs), o *master* é informado da falha e incrementa esse contador;
  - **lastContact**: todo DS armazena o timestamp do último contato com sucesso com cada DS da cadeia. Para cada erro de comunicação entre um DS e outro, o *master* é informado sobre quando foi o último contato bem-sucedido entre o DS reportando o erro e o DS irresponsivo.

  Com esses controles, o DS *master* poderá classificar um DS como indisponível caso haja um determinado número de falhas de comunicação ou se o DS estiver sem responder por um determinado período. O caso de um DS *master* da cadeia de replicação é tratado no item 4.9.2.2.

4.9.2 Recuperação automática

Uma vez detectada a situação de falha, os outros componentes do cluster devem substituir o componente classificado como indisponível por um remanescente. A responsabilidade pelo controle da substituição de um componente é como segue:

- **Substituição do CS**: Com o CS indisponível, cabe ao MS iniciar uma nova instância do CS.
- **Substituição do DS comum**: Assim que o DS *master* identificar que um DS da cadeia está indisponível, ele iniciará um processo de troca de mensagens com o CS para que um novo DS seja criado (ou para que um DS existente seja eleito) para compor a cadeia de replicação (veja item 4.8.2).

- **Substituição do DS *master***: Cabe aos demais componentes da cadeia de replicação promover o processo de eleição para designar um novo *master* para a cadeia de replicação.

A seguir são descritos os processos necessários para substituição de cada componente.

4.9.2.1 Substituição do CS

Embora o CS (Container Service) seja um elemento central na arquitetura, a CL (*Container List*) é armazenada em cache em todos os DSs do cluster, aumentando o desempenho das operações e diminuindo o impacto em caso de perda de comunicação. Em caso de falha do CS, qualquer DS poderá enviar mensagem ao MS (Metadata Service) solicitando que um novo CS seja iniciado. A única tabela necessária para o reestabelecimento do CS é a CL que raramente é atualizada e é armazenada em um *container* específico, replicado como os demais *containers*.

4.9.2.2 Falha do DS master

Caso o DS *master* da cadeia de replicação fique indisponível, os demais DSs da cadeia não poderão continuar os trabalhos normalmente, uma vez que toda a coordenação de sincronismo, necessária para a replicação do *container*, é feita pelo DS *master*. Assim, tão logo um DS identifique que o DS *master* está irresponsivo, uma eleição para indicar um novo *master* será iniciada. O processo de eleição de um novo *master* para a cadeia de replicação de um *container* segue um algoritmo de eleição em anel (TANENBAUM; STEEN, 2007), com os seguintes passos:

- Qualquer DS, ao detectar que o *master* está irresponsivo (e.g. ao receber timeout enviando mensagens por n vezes consecutivas) envia para o próximo DS responsivo uma mensagem ELEIÇÃO (próximo DS responsivo, pois um DS que não responda deve ser ignorado). Cada DS adiciona seu próprio ID e seus dados

e passa mensagem ao próximo. Cada DS adiciona os seguintes dados sobre si em uma mensagem de eleição:

  - ID do DS;
  - Identificador do rack do DS;
  - Total de *containers* dos quais o DS em questão é *master*;
  - Número de *chunks* que o DS tenha marcado como inválidos para o *container* cuja cadeia de replicação precisa de um novo *master*.

- Quando a mensagem retorna ao DS que iniciou a eleição, então ele faz com que uma nova mensagem COORDENADOR comece a circular. A mensagem COORDENADOR tem o ID do DS que cumprir melhor os seguintes critérios de eleição, nessa ordem de precedência:

  1. O DS que seja mais próximo da maioria dos outros DSs (informação baseada no identificador do rack do DS);
  2. O DS que já seja *master* de menos *containers* (ou cadeias de replicação);
  3. O DS que tenha menos *chunks* invalidados do *container* da cadeia de replicação em questão.

- Depois que a mensagem COORDENADOR circular, todos voltam a trabalhar com o novo DS *master*.

- O novo *master* solicita ao CS que atualize as informações da cadeia de replicação.

- O novo *master* inicia o processo para substituição do DS *master* antigo (item 4.9.2.3 abaixo).

4.9.2.3 Substituição de um DS

Comunicação com o CS: Uma vez que o DS *master* tenha identificado que um DS deve ser substituído, a exclusão de um DS da cadeia de replicação de um *container*

é feita através de envio de mensagem para o CS. Se a cadeia de replicação estiver descoberta, ou seja, com menos DSs que um mínimo pré-estipulado, o DS *master* solicitará ao CS, que seja designado um novo DS para compor a cadeia. O novo DS será sincronizado com os outros DSs da cadeia usando o processo de retomada de sincronismo através de *checkpoint*, (item 4.8.3.3 acima). Se o CS designar um DS que nunca teve dados do *container*, o sincronismo será retomado da mesma forma, já que todos os *chunks* serão marcados como inválidos, fazendo com que os dados disponíveis sejam recebidos dos demais DSs. **Transferência do papel de *owner***: Tão logo um DS seja declarado como indisponível, o DS *master* passa o papel de *owner* para outro DS ainda disponível na cadeia de replicação de cada *container*. Para decidir qual DS deve assumir o papel de *owner* dos *chunks* do DS indisponível, o DS *master* possui o seguinte controle:

- invalidCount: uma matriz com o número de *chunks* invalidados que um DS possui para cada outro DS que seja *owner* do *chunk* (*chunks* invalidados que o DSi tem do DSj). Esse contador é atualizado a cada mensagem de invalidate que o DS *master* recebe.

Assim, para cada *container*, considerando que o DSj esteja indisponível, para cada *chunk* cujo *owner* era DSj, o DS *master* arbitra como novo *owner* o DSi que possua, nessa ordem de precedência:

1. Menor número de *chunks* invalidados cujo *owner* seja o DS indisponível (menor invalidCountij - índice de desatualização);
2. Menor número de outros chunks sob sua responsabilidade (número de chunks onde owner=DSi - carga atual por *ownership*), isso é obtido no mapa de controle do *container*;

Reestabelecimento de um DS: Para evitar problemas causados por um DS que ficou sem comunicação e que consequentemente não saiba que perdeu a condição de *master*, na inicialização do DS, ou depois de um tempo ocioso pré-estipulado, ou seja,

um tempo sem receber nenhuma mensagem, cada DS entra em contato com o CS perguntando se ele ainda é componente das cadeias de replicação que ele tem informação. Com a resposta do CS, o DS identifica que foi removido e indica ao CS que pode ser readmitido. O DS reestabelecido passa a ser uma opção para que o CS o indique como componente da cadeia de replicação, se esta ainda estiver descoberta.

4.10 Controle de locks

Liao lembra em (LIAO, 2010) que o padrão POSIX define uma série de características comumente respeitadas por diversos sistemas de arquivos distribuídos, contudo o respeito dessas características muitas vezes implica em perda de desempenho de leitura e escrita concorrente, especialmente quando se trata de atomicidade das operações e na coerência de cache:

Atomicidade: implica que os dados de uma operação de escrita sejam inteiramente disponíveis ou completamente indisponíveis a uma operação de leitura.

Cache (*client-side file cache*): é o recurso pelo qual os dados são armazenados no client de forma a diminuir comunicações entre cliente e servidor. Contudo esse tipo de recurso demanda um controle que garanta que as atualizações feitas em *cache* sejam propagadas para o servidor e demais caches, de forma a garantir que todos os acessos concorrentes tenham uma visão coerente dos dados compartilhados.

Cabe ao sistema de controle de *locks* garantir que as operações de leitura e escrita sejam feitas de forma atômica, garantindo coerência entre as diversas cópias dos dados. Para que um processo de leitura ou escrita possa ser feito, é preciso que um lock específico para a atividade seja cedido e controlado, de forma a impedir que outra atividade com o mesmo dado, porém efetivada por outro agente (cliente ou transação), possa levar a um estado de incoerência entre as cópias armazenadas nos diversos caches.

Para o AwareFS, o controle de *locks* é feito pelo serviço de *locks* (LS) no mesmo servidor onde residem os serviços de dados ou metadados (respectivamente DS e

MS).

O AwareFS segue o mesmo modelo de *locks* usado pelo DLM (*distributed lock manager*) do VMS e aprimorado pelo Lustre (WANG et al., 2009). Nesse modelo são usados *locks* com modos e tipos diferentes. É o modo do lock que vai indicar a forma de bloqueio que o lock deve proporcionar, por exemplo, se é um lock para acesso exclusivo ou se é um lock para escrita concorrente. Por outro lado, é o tipo do lock que definirá qual recurso que o lock deve proteger, ou seja, qual é o tipo de objeto cuja integridade está sendo garantida pelo lock. Os modos e tipos possíveis para o AwareFS são os seguintes:

Modos:

- NL Null: Lock que indica o interesse no recurso, mas não impede que outros *locks* sejam concedidos para o mesmo recurso.
- EX Exclusive: Lock exclusivo, usado para escrever no diretório em um processo de criação de arquivo. Também usado para bloquear um chunk na transferência do papel de cópia primária.
- PW Protective Write (normal write): Lock usado na escrita de dados em um arquivo.
- PR Protective Read (normal read): Lock usado na leitura de dados em um arquivo.
- CW Concurrent Write mode: Lock de escrita concorrente, usado na atualização de metadados durante um processo de abertura de um arquivo para escrita.
- CR Concurrent Read mode: Lock de leitura de metadados, usado durante um processo de “lookup”, para permitir a leitura de metadados de diretórios e links presentes no caminho especificado.

Tipos:

- chunk lock: Lock usado pelo DS para garantir a consistência dos dados em um *chunk* durante o processo de leitura/escrita.
- flock: Advisory Lock requisitado pelo usuário, associado ao metadado do arquivo, usado na função flock.
- metadata lock (inode bit): Lock usado pelo MS para proteger metadados.
- Path lock: Usado pelo MS para garantir consistência da base central de referência aos inodes pelo respectivo *pathname*.

Seguindo o modelo de *locks* por extensão (Extent Locks), no caso do AwareFS os *locks* de dados são controlados no nível de extensões de um *chunk*, ou seja, duas operações de escrita em um mesmo trecho de um *chunk* não são feitas de forma concorrente por servidores diferentes, contudo operações de escrita em trechos diferentes poderão ser executadas em paralelo.

Para o AwareFS os *locks* podem ser solicitados pelo MS (para garantir a consistência dos metadados) ou pelo DS (para garantir escrita concorrente e consistente simultaneamente em diferentes chunks e inodes).

Para garantir a distribuição do gerenciamento de *locks*, existe um LS associado ao MS e um LS associado a cada DS.

4.10.1 Locks de dados

Para garantir a coerência entre escritas e leituras de um mesmo *chunk*, o AwareFS conta com um mecanismo de bloqueio que tem por objetivo:

1. Garantir que seja válido apenas o dado referenciado pelo metadado;
2. Impedir que um *chunk* desatualizado seja lido durante ou depois que uma escrita tenha atualizado o metadado;

3. Impedir que mais de uma escrita seja feita em uma mesma região.

Os *locks* de dados servem para bloquear o trabalho com dados do arquivo acessado. Esse tipo de *lock* pode ser concedido por região dentro de um mesmo *chunk*, com cálculo e controle feitos pelo LS associado ao DS que tem o *chunk*.

Internamente, de forma transparente para o usuário, o serviço de dados (DS) faz uso de *locks* nas seguintes condições:

- Para impedir que um trecho de um dado *chunk* seja lido durante o processo de atualização, o DS enfileira uma solicitação de *lock* de escrita (PW) para a região afetada.
- Como último passo no processo de escrita, para atualizar a versão do *chunk* no FID salvo no header do arquivo, o DS enfileira uma solicitação de *lock* de escrita concorrente (CW) no *inode*.

4.10.2 Locks de path

Segundo o especificado por Ritik Malhotra em (MALHOTRA, 2014) sistemas de arquivos baseados em pathnames (como os desenvolvidos com a biblioteca FUSE (SINGH, 2014)), precisam de mecanismos que garantam a consistência das estruturas de acesso usadas para consultas feitas a partir do *pathname* de um arquivo ou diretório. Especialmente em sistemas concorrentes, a intercalação de operações nos metadados pode quebrar facilmente a consistência do acesso a um item, por exemplo:

- Thread 1 :
- write(‘/a/b’;DataChunk1)
- Thread 2 :
- rename(‘/a’,’/c’)
- Thread 1 :

- write(‘/a/b’;DataChunk2)

Para evitar esse tipo de inconsistência, o MS do AwareFS trabalha em conjunto com uma instância do LS para controlar os *locks* necessários. Os *locks* de path são concedidos em ordem lexicográfica e removidos na ordem inversa.

4.10.3 Locks de metadados

Além da referência feita pelo pathname do arquivo, o AwareFS possui metadados para todos os arquivos que precisam ter sua consistência garantida ou o acesso aos dados pode ser comprometido. Para essa garantia, são submetidos e controlados *locks* de metadados que são *locks* necessários para alterar de forma consistente qualquer item armazenado no *inode* (arquivo h.*).

Os *locks* de metadados são controlados pelo LS junto do DS dono do inode.

Em situações de falha, *locks* de metadados são revogados caso haja mudança do DS dono do *inode*. Para garantir a consistência das operações os *locks* devem ser renovados ou as operações devem falhar.

4.10.4 Locking Service - LS

Como citado no item 4.4, todo o trabalho de gerenciamento de *locks* do AwareFS é feito de forma distribuída. Cada LS responde pelos *locks* associados pelos dados e metadados sob o controle do DS ao qual ele está associado.

Para fazer o gerenciamento dos *locks* o LS possui as seguintes estruturas de dados:

- aw_lock_request: Estrutura para registro de solicitação de *lock* composta pelos seguintes dados:
  - RequestorID: Identificador do componente (DS ou clientAware) que requisitou o *lock*;

  - ResourceID: Identificador do recurso ao qual o *lock* se refere (FID ou PathName);
  - LockType: Identificando qual o tipo de *lock* (*chunk*, flock, metadata ou path);
  - ReqLock: Modo de *lock* requisitado (e.g. CW);
  - Extent: Offset e comprimento da região ao qual o *lock* se refere;
  - GranterID: Identificador do DS que concedeu o *lock*;
  - Timestamp: momento da requisição do *lock*, usado para ordenação e no processo de expiração de requisições.

- Granted_Llist: Lista de solicitações de *locks* atendidas;
- Waiting_Llist: Lista de solicitações de *locks* que ainda não foram concedidos.

4.10.5 Solicitação de locks

O processo de solicitação de *locks* no AwareFS é feito pelo método interno SetLK que recebe uma estrutura aw_lock_request como parâmetro e segue o procedimento padrão usado por sistemas Linux baseados no POSIX (P; MARCO; others, 2007), onde são usados os seguintes passos:

1. **Verificação de conflitos**: Com base no modo de *lock* solicitado, o LS verifica primeiramente se o *lock* é compatível com os *locks* já concedidos (Granted_Llist). Em seguida é verificado se o *lock* é compatível com os *locks* solicitados que ainda não foram concedidos (Waiting_Llist). As verificações de compatibilidade são feitas com base na matriz da Figura 3, levando em consideração a região para a qual o *lock* foi solicitado. Se não houver conflitos o lock_request entra para a fila Granted_Llist, uma mensagem é enviada ao solicitante indicando que o *lock* foi concedido (com detalhes sobre a região do *lock* efetivamente concedido) e o processo termina com sucesso.
2. **Tratamento de conflitos**: Para cada conflito é passada uma mensagem para o componente detentor do *lock* conflitante (clientAware, DS ou MS) para que este

possa persistir dados em memória e liberar o *lock*. O lock_request entra então para a fila Waiting_Llist e uma mensagem é enviada ao solicitante indicando que o *lock* está sendo processado.

4.10.6 Cancelamento de locks

Assim como o proposto nas versões iniciais do Lustre (WANG et al., 2009), os *locks* usados no AwareFS são retidos pelo componente solicitante e só são revogados em duas condições:

- Se não forem renovados depois de um tempo pré-estipulado (referenciado como "tempo de *lease*"), isto é, depois de um período o LS envia uma mensagem para o detentor de um *lock* informando que o *lock* foi revogado. Operações de escrita em curso falham e processo de escrita precisará ser refeito. Essa situação tende a ser rara, pois o tempo de lease deve ser bem maior que o tempo de escrita de um chunk completo.
- Se uma solicitação conflitante surge, o LS envia mensagem solicitando que o detentor do *lock* de forma que este possa persistir dados em memória e liberar o *lock*.

Para viabilizar o cancelamento de *locks*, o LS segue os seguintes passos:

1. Uma mensagem é enviada ao detentor do *lock* com as informações da solicitação de *lock* sendo revogada;
2. A solicitação revogada é então removida da Granted_Llist ou da Waiting_Llist;
3. As solicitações na Waiting_Llist são então reprocessadas da mesma forma descrita no item 4.10.5;
4. Se for encontrada na Waiting_Llist uma solicitação que possa agora ser concedida, está é movida para a Granted_Llist e uma mensagem é enviada ao solicitante indicando que o *lock* foi concedido e o processo termina com sucesso.

4.10.7 Exemplo – Leitura e escrita concorrentes

Com o AwareFS, uma situação em que uma mesma região de um arquivo /home/foo/bar/B.txt precisa ser lido pelo processo C1 e escrito pelo processo C2 teria a seguinte sequência de passos:

1. C2 enfileira no LS do MS um pedido de *lock* de path (CR) para /home, /home/foo, /home/foo/bar e /home/foo/bar/B.txt. O MS retorna a estrutura FID de /home/foo/bar/B.txt.
2. C2 consulta a CL para identificar o DS com o *container* identificado pelo CID retirado da estrutura FID.
3. O DS com o *container* é consultado para encontrar o inode de /home/foo/bar/B.txt. O clientAware envia mensagem ao DS solicitando consulta (mensagem CheckHDR).
4. O DS identifica o *chunk* a ser acessado com o offset indicado na requisição de escrita.
5. O DS identifica o dono do *chunk* e o processo continua com este outro DS.
6. O DS dono solicita *lock* de escrita (PW) para região alterada ao LS local (LS do dono do *chunk*). Supondo que esta mesma região esteja sendo lida pelo processo C1 (*lock* tipo PR), o LS envia uma mensagem para ele solicitando que o *lock* seja liberado. O processo só segue se o *lock* tipo PW for concedido ao DS;
7. O DS dono atualiza o chunk com o buffer passado pelo processo clientAware;
8. O DS dono do *chunk* manda mensagem para que os outros DSs da cadeia invalidem a região nas suas cópias do *chunk*;
9. A versão do *chunk* é atualizada no inode quando o *lock* for concedido;
10. O *lock* PW é liberado;

11. O *chunk* fica livre para ser replicado e uma nova solicitação de *lock* eventualmente feita por C1 pode agora ser concedida.

4.11 Acesso de leitura aos metadados e dados

Para identificar a localização dos dados e conseguir acesso de leitura ou escrita a estes, o processo segue os passos descritos abaixo. Considerando por exemplo um acesso ao arquivo /home/foo/bar (sem, no entanto, considerarmos os *locks* necessários nos metadados de /home, /home/foo e /home/foo/bar), teremos os seguintes passos (Figura 20):

1. O serviço cliente (clientAware) obtém o FID do arquivo /home/foo/bar (composto por CID, Inode#, IDUnico) ao serviço de metadados, o MS;
2. clientAware consulta a CL (Container List) para identificar um servidor DS com o *container* identificado pelo CID. Se o clientAware estiver em execução no mesmo servidor que um DS com uma cópia do *container*, este DS será o primeiro a ser consultado para acesso ao inode;
3. O DS (Data Server) designado, ou seja, o DS com o *container*, é consultado para encontrar o *inode* indicado anteriormente na estrutura FID.
4. Se não é um FID de inode continua no passo 7 abaixo.
5. O metadado (*inode*) de /home/foo/bar, armazenado em um arquivo tipo head (veja item4.7), possui um vetor de estruturas FID que indica que ele é composto, por exemplo, dos chunks B0, B1 e B2, todos com tamanhos fixos, sendo que o último pode ter um tamanho menor. Para cada ocorrência do vetor de estruturas FID, é armazenado também o lko (do inglês "*last known owner*") que é um inteiro indicando o último DS que foi owner desse FID, ou zero caso não tenha havido um owner anteriormente.
6. Com o offset indicado na requisição de leitura, é identificado, por exemplo, que o chunk a ser acessado é o B1. A estrutura FID de B1 é usada em uma nova

consulta a CL, para identificar um DS com uma cópia do *container* armazenando os dados do *chunk* B1 (passos 2 e 3). Se o processo solicitante da leitura residir no mesmo servidor que um DS com cópia, esse será preferencialmente o DS a ser utilizado.

7. Com o número do *chunk* (inode# na estrutura FID) e o *container* ID, o DS indicado verifica então os seguintes itens, verificações de disponibilidade:

    - a. Se o *chunk* está disponível: o DS verifica se existe no *container* um registro para o *chunk* especificado.
    - b. Se o *chunk* de B1 está completamente invalidado: isso é feito checando a flag de invalidação armazenada no mapa de controle do *container*, na posição apontada pelo índice do *chunk*.
    - c. Se o *chunk* é o corrente: isso é feito verificando se o identificador único (versão), especificado no FID recebido, é o mesmo que o armazenado no vetor de controle de versão do mapa de controle do *container*.
    - d. Se o *chunk* está válido, isso é feito verificando se temos zero na posição referente ao *chunk* no vetor de controle de invalidação do mapa de controle do *container*.

8. Se as verificações do passo 7 falharem, o DS retorna erro e o processo volta ao passo 6 para consultar um novo DS com uma cópia do *container*. Se nenhum DS disponível possuir um arquivo válido, compatível com a solicitação de leitura, um erro de E/S é retornado.

9. O DS passa ao clientAware o conteúdo do *chunk*, a partir do offset solicitado.

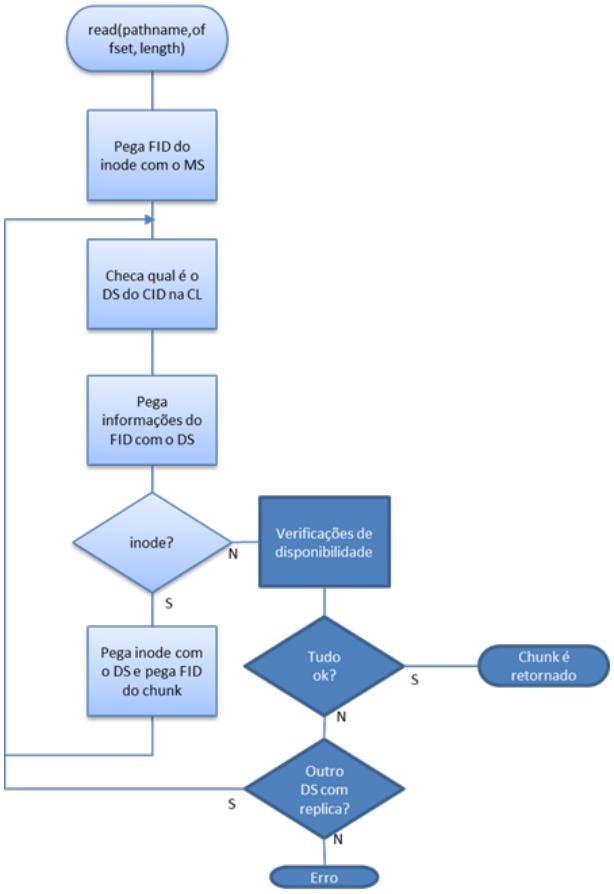

Figura 20 - Leitura de chunks

4.12 Processo de escrita

O AwareFS, como outros sistemas de arquivos distribuídos, usa um protocolo de consistência baseado em primário, onde requisições de escrita são encaminhadas para uma das cópias. Nesse modelo a consistência sequencial é garantida direcionando todos os comandos de atualização para o nó com a cópia primária que ordena as operações em um esquema FIFO ("*first in, first out*"). No caso do AwareFS, a cópia primária é aquela em poder do DS owner do *chunk*. Como dito no item 4.8.1, com o protocolo de escrita local do AwareFS, o papel de owner pode migrar de um DS para outro.

Para garantir a coerência entre leituras e escritas, todos os dados são acessados por outros clientes só depois da atualização dos metadados que é o último passo dentro do processo de escrita. Enquanto uma atualização não é finalizada e suas cópias invalidadas a versão do dado não é atualizada e os metadados continuam apontando para a versão anterior. Leituras e escritas de dados e metadados são organizadas com um sistema de obtenção de locks.

Toda escrita no AwareFS é feita de forma transacional, ou seja, o processo só é dado como terminado com sucesso se todos os passos envolvidos tiverem sido executados sem erros e se um mínimo de cópias do dado estiver disponível.

Como fizemos para a leitura, considerando, por exemplo, uma escrita em um arquivo /home/foo/bar existente (ainda sem considerarmos os locks necessários nos metadados de /home e /foo), teremos os seguintes passos:

1. O serviço clientAware recebe uma solicitação de escrita contendo o nome completo do arquivo ("/home/foo/bar"), um buffer com os dados a serem gravados e um offset indicando a posição dentro do arquivo;

2. O serviço de metadados (MS) é consultado pelo serviço clientAware e retorna o FID do arquivo (informação de CID, Inode#, IDUnico);

3. O clientAware consulta a CL (Container List) para identificar o servidor DS com o

container identificado pelo CID. Se o clientAware estiver em execução no mesmo servidor que um DS com uma cópia do *container*, este DS será o primeiro a ser consultado para acesso ao inode;

4. O DS com o *container* é consultado para encontrar o inode indicado anteriormente na estrutura FID;

5. Se a estrutura FID está referenciando um *chunk* o processo continua no passo 7 abaixo.

6. Se o FID é de metadados o clientAware envia mensagem ao DS solicitando consulta (mensagem CheckHDR). Com o offset indicado na requisição de escrita, é identificado o *chunk* a ser acessado. A estrutura FID do *chunk* inicial da escrita e seu respectivo lko (identificador do último DS que tenha sido dono desse *chunk*) são recuperados. Caso o lko seja igual a zero, a estrutura FID é usada em uma nova consulta a CL, para identificar um DS com uma cópia do *container* armazenando o *chunk* (passos 3 e 4 acima). Do contrário o processo segue para o próximo passo.

7. Para atualizar o *chunk*, o DS precisa inicialmente identificar qual DS é o atual dono do *chunk*. Se o DS atualmente sendo consultado for o dono do *chunk*, o processo de escrita segue, senão um novo DS com uma réplica é consultado (processo volta ao passo 3 acima).

8. DS solicita lock de escrita (PW) para região alterada ao LS local que encaminha para o LS dono do *chunk*;

9. O DS dono atualiza o *chunk* com o buffer passado pelo processo clientAware (o processo para atualizar o *chunk* precisa fazer o "*copy-on-write*", atualizando o undo log dos *checkpoints* existentes);

10. O DS dono do *chunk* identifica na CL (lista de *containers*) quais são os outros nós que compõem a cadeia de replicação do *container* e manda mensagem para que os outros DSs invalidem a região nas suas cópias do *chunk*;

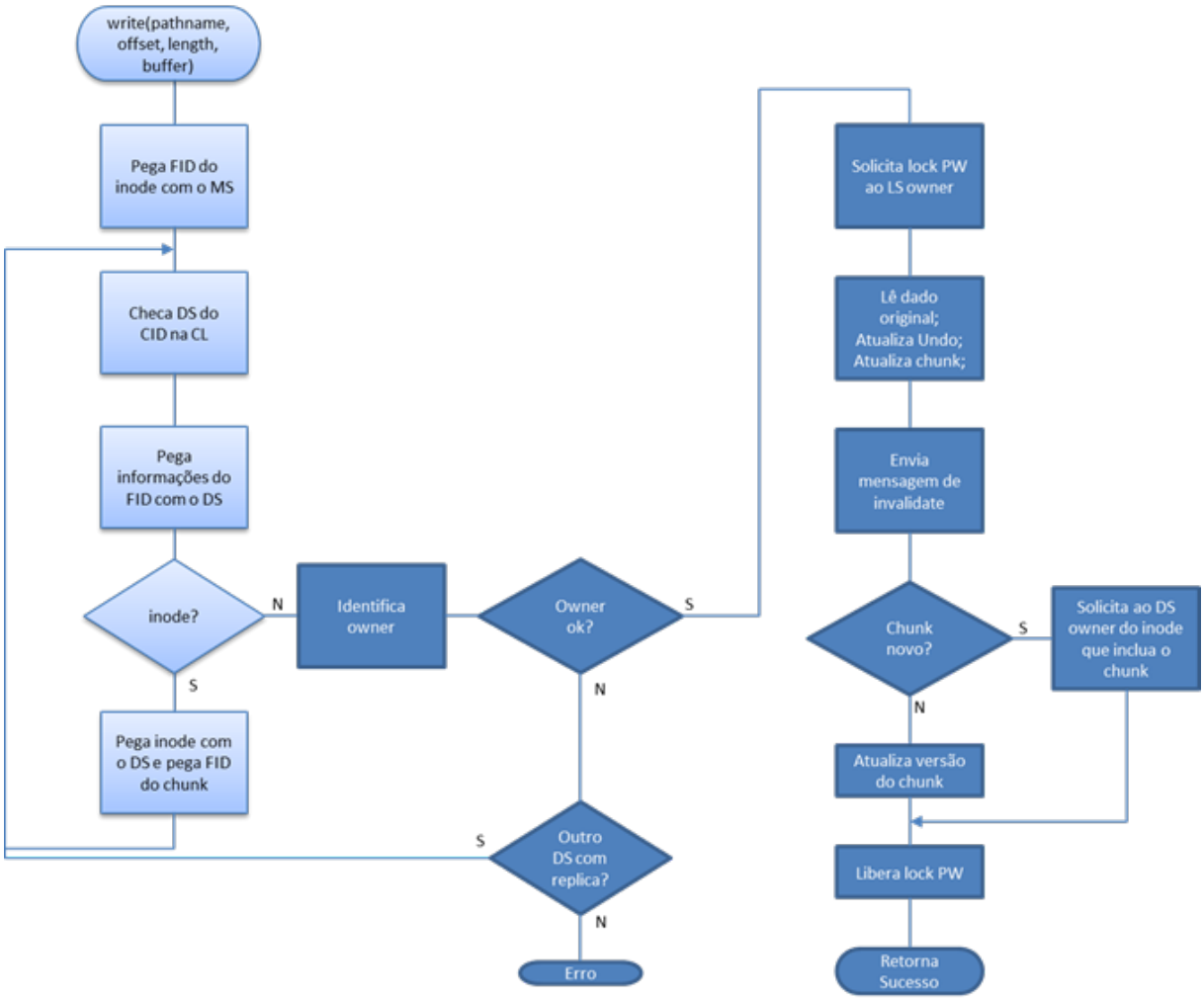

Figura 21 - Processo de escrita

11. Se for uma reescrita de um *chunk* já alocado, a versão do *chunk* é atualizada no mapa de controle do *container*. Se for uma escrita em um novo *chunk*, o identificador (FID) do *chunk* é inserido no inode;

12. Libera lock PW;

13. O *chunk* fica livre para ser replicado;

4.13 Transferência do papel de nó dono do chunk

Seguindo o descrito no item 4.12 percebe-se que os dados deverão trafegar pela rede para serem gravados primeiro no nó dono do *chunk*, mesmo para um caso em

que a solicitação de escrita parta de outro nó que seja membro da cadeia de replicação. Com esse comportamento, seguindo o exposto na Figura 21, um processo sendo executado no nó 2 ou no nó 3 deve transferir os dados para o nó 1 pela rede, mesmo possuindo uma cópia local do *chunk*. Para evitar esse tráfego de dados pela rede e garantir um processo de escrita mais rápido, o AwareFS transfere o papel de nó dono da cópia primária do *chunk* (owner do *chunk*) para o nó iniciando o processo de escrita, desde que não haja outro processo de escrita em curso para o mesmo *chunk*. Considerando um caso em que:

- a cadeia de replicação do *container* seja DS1, DS2, DS3;
- o processo clientAware esteja em execução no nó 2;
- o owner do inode de metadados é o nó 2 (DS2);
- o owner do *chunk* seja o nó 1 (DS1);

O processo de escrita descrito no item 4.12 é complementado com a execução dos seguintes passos (entre os passos 7 e 8 do processo):

- O DS dono do *chunk* verifica na CL se o nó onde o clientAware reside faz parte da cadeia de replicação. Como o clientAware reside no nó 2, ou seja, no mesmo nó que o DS2 que também é componente da cadeia de replicação, o DS1 verifica se o *chunk* não está sendo atualizado por outro processo (obtendo um lock exclusivo, modo=EX) e, se for esse o caso, passa o papel de dono do *chunk* para o DS2 seguindo um protocolo de comprometimento de duas fases como segue:
  - a. O DS1 envia uma mensagem NewOwnerReq para DS2 e DS3 indicando que o novo dono do *chunk* deve passar a ser o DS2;
  - b. DS2 e DS3 enviam mensagens NewOwnerCommit para DS1 concordando com a transação;
  - c. DS1 envia uma mensagem GlobalCommit para DS2 e DS3 que então atualizam o mapa de controle do *container* indicando que o owner do *chunk* agora é DS2;

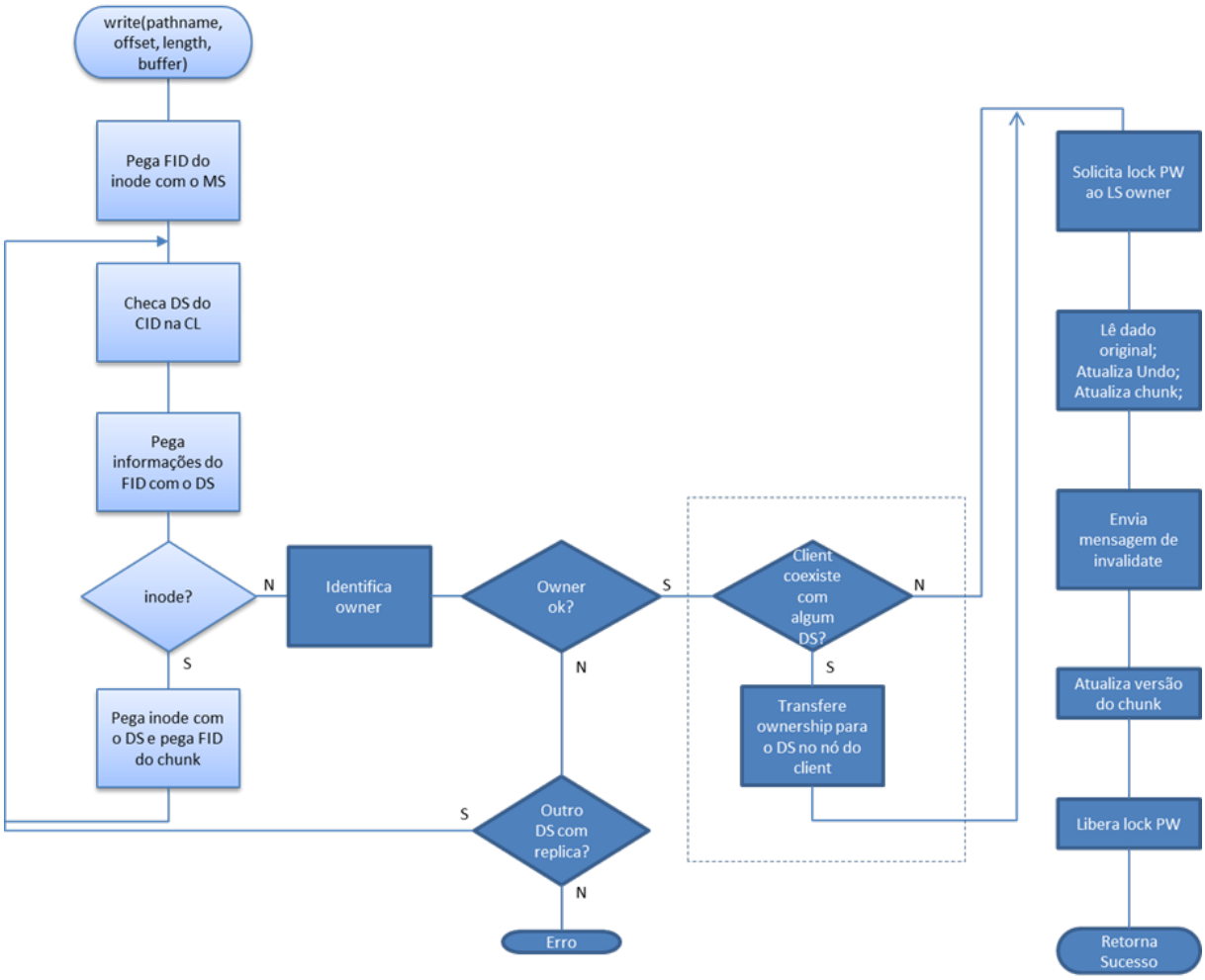

Figura 22 - Escrita com transferência do papel de nó dono do chunk

– d. O DS1 envia então resposta ao clientAware solicitante (no nó 2) indicando que o processo de escrita deve ser redirecionado para o DS2;

5 IMPLEMENTAÇÃO

Diferente do Hadoop e seu HDFS que foi desenvolvido basicamente em Java, para o AwareFS a linguagem base escolhida, principalmente por questões de desempenho, foi o C++14. Considerando que os componentes são distribuídos, podendo usar diversos hosts conectados via rede TCP/IP, a comunicação entre os componentes se divide em dois modelos:

- RPC: Para comunicação síncrona entre clientes e serviços, como:
  - Comunicação entre clientAware e DS;
  - Comunicação entre clientAware e CS;
  - Comunicação entre DS e CS;
  - Comunicação entre DS, clientAware e MS;
- MPI: Para comunicação assíncrona entre DSs e entre CS e DSs.

Para viabilizar o desenvolvimento ágil dos serviços baseados em RPC (i.e., "*Remote Procedure Call*"), foi escolhida inicialmente a ferramenta Thrift (RAKOWSKI, 2015) como framework de implementação de código C++.

Originalmente desenvolvida pelo Facebook, o Apache Thrift é uma ferramenta para auxiliar o desenvolvimento de código distribuído em diversas linguagens. Para usar o Thrift, o desenvolvedor cria uma interface em um código próprio chamado IDL (*interface definition language*) com uma sintaxe semelhante ao C. Para o código IDL, o desenvolvedor conta com os tipos básicos como inteiro e *string*, que podem ser combinados em estruturas (como a *struct* do C). Também é na IDL que o desenvolvedor declara os métodos de interface dos serviços RPC. Uma vez criado o código IDL, o desenvolvedor pode usar o Thrift para gerar o código necessário para “serializar” e “desserializar” os dados e promover as chamadas RPC, tudo isso na linguagem que ele desejar, a partir de um leque que vai desde Python, Java até Go e C++. Seguindo

a arquitetura proposta no item 4.4, a implementação do AwareFS será feita com os seguintes componentes:

- **Container**: Estrutura e métodos para controlar o armazenamento dos dados no AwareFS.
- **MS (Metadata service)**: É um serviço para controlar as referências entre pathname de um arquivo e seu FID, o identificador que definirá o acesso ao dado usando os demais serviços.
- **LS (Locking service)**: É o serviço que atende as solicitações de lock para garantias de consistência.
- **DS (Data service)**: É um serviço com métodos para tratamento dos dados armazenados nos *containers*. Cada instância de DS, controla uma lista de objetos do tipo TContainer.
- **CS (Container service)**: Trata-se do serviço para controle de *containers* e gerenciamento do cluster de DSs.
- **FuseClientAware**: É o serviço de sistema de arquivos em modo usuário (baseado no FUSE(VANGOOR et al., 2019)) para atender requisições de leitura e escrita iniciadas pelo cliente do AwareFS usando uma interface padrão POSIX.

A seguir são descritos maiores detalhes sobre a implementação do protótipo do AwareFS, criado para avaliar a estratégias propostas, destacando vantagens e desafios a serem observados na materialização de um sistema de arquivos com as características buscadas.

5.1 Processamento concorrente

Para aumentar a eficiência do processamento, todos os componentes do AwareFS executados nos servidores usam um modelo de programação concorrente baseado em múltiplas threads que manipulam os mesmos dados de controle em um esquema

de memória compartilhada. A execução de várias threads em paralelo traz maior desempenho e melhor aproveitamento dos recursos computacionais porém requer artifícios específicos como utilização de *locks* e primitivas de sincronização (e.g., mutex) (WILLIAMS, 2012).

Todos os serviços do AwareFS fazem uso de múltiplas threads que são programadas usando o OpenMP(CHANDRA, 2001) como ferramenta para facilitar o desenvolvimento. As diversas threads são indicadas no código com diretivas “omp parallel”, “omp single”, “omp section” e “omp parallel for”. Outros recursos amplamente do OpenMP usados são o sincronismo entre threads com a diretiva “omp critical” e o controle de *locks* com as funções que manipulam estruturas omp_lock_t e omp_nest_lock_t.

Além dos controles obtidos com o OpenMP, duas situações demandaram o uso da biblioteca de gerenciamento de threads do C++, são elas:

- **Criação de thread e execução paralela em segundo plano**: Dentro das estruturas do OpenMP não foi possível encontrar uma construção que possibilitasse a criação de uma thread com execução imediata em segundo plano, concorrente com a thread de origem. Para esse tipo de necessidade usou-se o método detach() da classe std::thread do C++14.
- **Bloqueio condicional de thread**: No caso do DS do AwareFS, a replicação de dados é feita por uma thread separada que só deve ser executada, só deve “acordar”, se houver algum dado que precisa ser replicado. Para isso foi usada a classe std::condition_variable, fazendo que o evento de invalidate notifique a thread de replicação que mais um chunk ou inode foi invalidado e precisa ser replicado.

Esses recursos de processamento concorrente são muito importantes e amplamente usados para garantir que as diversas funções de cada componente possam ser executadas em paralelo, aumentando o desempenho e diminuindo a latência no atendimento das solicitações.

5.2 Arquivo de log

Por conta da complexidade inerente de um sistema de arquivos distribuído como o AwareFS, é muito importante o uso eficaz de logs para registrar as atividades e mensagens de depuração. Para esse fim, foi usada a biblioteca Apache log4cxx (WHAT. . . , 2008) que possui diversas facilidades, em especial a capacidade de definir a "verbosidade" do log nos níveis TRACE, DEBUG, INFO, WARN, ERROR e FATAL, exatamente como o log4J que é amplamente usado com Java,

5.3 Parametrização e arquivos de configuração

Todos os componentes do AwareFS podem ser configurados de diversas formas, adequando o seu funcionamento às características do cluster usado. Para o tratamento de parâmetros, seja via linha de comando ou arquivo de configuração, foi usada biblioteca program_options do conjunto de bibliotecas Boost do C++ (SCHäLING, 2011). Dessa forma, todos os componentes mostram um texto de ajuda se forem executados com a opção "–help", sendo que cada componente possui parâmetros diferentes que podem ser especificados via valor default, linha de comando ou arquivo de configuração.

5.4 Serialização de dados (Marshalling)

Para armazenamento e comunicação de dados de controle, todos os componentes do AwareFS usam um processo comum de serialização, ou seja, cada componente traduz as informações em memória para um formato que possa ser armazenado e transferido entre sistemas computacionais diferentes (RAKOWSKI, 2015). No desenvolvimento do protótipo do AwareFS, o Apache Thrift foi usado como ferramenta para facilitar a programação de serialização e desserialização de dados.

O Protocol Buffers da Google também foi estudado como ferramenta de Marshalling durante as pesquisas que precederam o desenvolvimento do protótipo do Awa-

reFS, porém o Thrift foi selecionado como ferramenta por conta de suas funcionalidades, mesmo perdendo ligeiramente em desempenho de desserialização (SUMARAY; MAKKI, 2012). Armazenamento em formato JSON foi uma funcionalidade do Thrift que foi muito utilizada para facilitar a depuração durante o desenvolvimento.

5.5 Controles e comunicação síncrona com Thrift

Operações entre os componentes do AwareFS que precisem acontecer de forma síncrona são feitas com chamadas de RPC que são construídas usando o Apache Thrift. Todos os métodos que podem ser chamados por um componente remoto são declarados usando a linguagem de definição de interface (IDL) do Thrift.

Como cada componente do AwareFS possui características diferentes, atendendo mais ou menos requisições por unidade de tempo, dois tipos diferentes de servidores Thrift foram usados, o TThreadedServer para o CS e para o MS e o TNonblockingServer para o DS.

5.6 Controles e comunicação assíncrona com MPI

Funções relacionadas com a resiliência dos dados e componentes do cluster AwareFS são feitas com um processo de troca de mensagens entre os serviços do AwareFS. O único componente do AwareFS que não se comunica por troca de mensagens é o clientAware, uma vez que basicamente ele é usado para atender requisições de usuário e aplicações.

No protótipo do AwareFS, o MPI é usado como framework para troca de mensagens em todo o processo de criação da cadeia de replicação de cada *container*, no processo de admissão de um DS no cluster, na criação de checkpoints dos *containers* e no processo de replicação de dados.

Um comunicador (MPI_COMM_WORLD) é usado para comunicação do CS com os DSs e vice-versa. Cada cadeia de replicação de *container*, composta por três pro-

cessos DS, compõe um comunicador que é usado para a comunicação entre os DSs da cadeia de replicação. Também, para viabilizar o processamento de mensagens de diferentes tipos, são geradas duas novas cópias de cada comunicador de cadeia de replicação. Dessa forma, cada cadeia de replicação usa dois comunicadores: um para processamento de replicação e outro para requisições de criação de checkpoints. O comunicador global MPI_COMM_WORLD (incluindo CS e DSs) é usado para mensagens de criação de novos *containers* e novas cadeias de replicação, mensagens de inicialização de réplicas e mensagens de processamento de pedidos de checkpoint (no protótipo esses pedidos são iniciados pelo CS).

Todas as mensagens são enviadas de um processo para outro usando MPI_Send e recebidos com MPI_Improbe e parâmetros MPI_ANY_SOURCE e MPI_ANY_TAG. É a tag da mensagem MPI que vai indicar qual parte do código deve ser acionada, sendo que o pay-load da mensagem contém os dados a serem processados, tudo devidamente serializado.

Na implementação 4.0.1 do OpenMPI, todo o processo de conexão e desconexão de um processo a um comunicador MPI apresentava alguns problemas que foram resolvidos (bugs 3458 e 6446), porém voltaram a causar problemas em versões posteriores como a 4.0.2. Isso somado à complexidade intrínseca dos processos de detecção e recuperação de falhas motivou a implementação do protótipo do AwareFS com apenas procedimentos de organização do cluster, criação de checkpoints e replicação dos dados, que são aspectos cuja inexistência impactaria nas avaliações de desempenho.

5.7 Detalhes de implementação dos components

Mesmo com o uso de ferramentas como Thrift, OpenMP e OpenMPI, o desenvolvimento do protótipo do AwareFS demandou diferentes estratégias e técnicas cujo entendimento é necessário para melhor avaliação dos resultados obtidos a partir da ideia proposta. A seguir são detalhados aspectos específicos de cada componente do

AwareFS, trazendo a devida dimensão do trabalho desenvolvido.

5.7.1 Classe TContainer

A classe TContainer não é derivada de nenhuma outra e foi criada com o objetivo de descrever o estado de um *container* e a interface para manipulação do *container* e seus dados.

A identidade e o estado de um *container* são descritos com variáveis para identificador, versão do *container*, freeze (para identificar se o *container* parou de receber alterações), vetor de registro de *inodes/chunks* (lista CInodes), caminho do arquivo com o mapa, caminho do diretório com os dados do *container* e mapa do *container* (ContainerMap).

Como dito no item 4.7, cada item armazenado em um *container* é referenciado como um nodo. A lista usada para controle de *inodes/chunks* chamada CInodes tem uma entrada por nodo com uma variável chamada índice que é a posição que o nodo ocupa no mapa do *container*. Essa lista é necessária pois tal posição pode variar entre as réplicas de um *container*, uma vez que a posição de um nodo pode não ser mantida quando este é replicado (Figura 23). Isso acontece em situações de escrita simultânea nas várias réplicas de um *container*. O mapa do *container* (cmap) é um objeto do tipo ContainerMap que é usado para armazenar vetores com os identificadores dos *inodes/chunks*, o tipo dos *inodes/chunks* (c = chunk, h = inode ou header), uma flag indicando se os *inodes/chunks* foram invalidados e a versão dos *inodes/chunks*. Esse mapa tem um limite flexível com um default de 768 entradas que é suficiente para armazenar até o máximo de 256 entradas que um *container* vai ocupar. O número é o triplo do máximo de *containers* pois um mesmo *container* que inicia a gravação de um chunk pode receber de forma assíncrona as réplicas de outros *containers*. ContainerMap é um objeto definido com o Thrift para possibilitar serialização e desserialização.

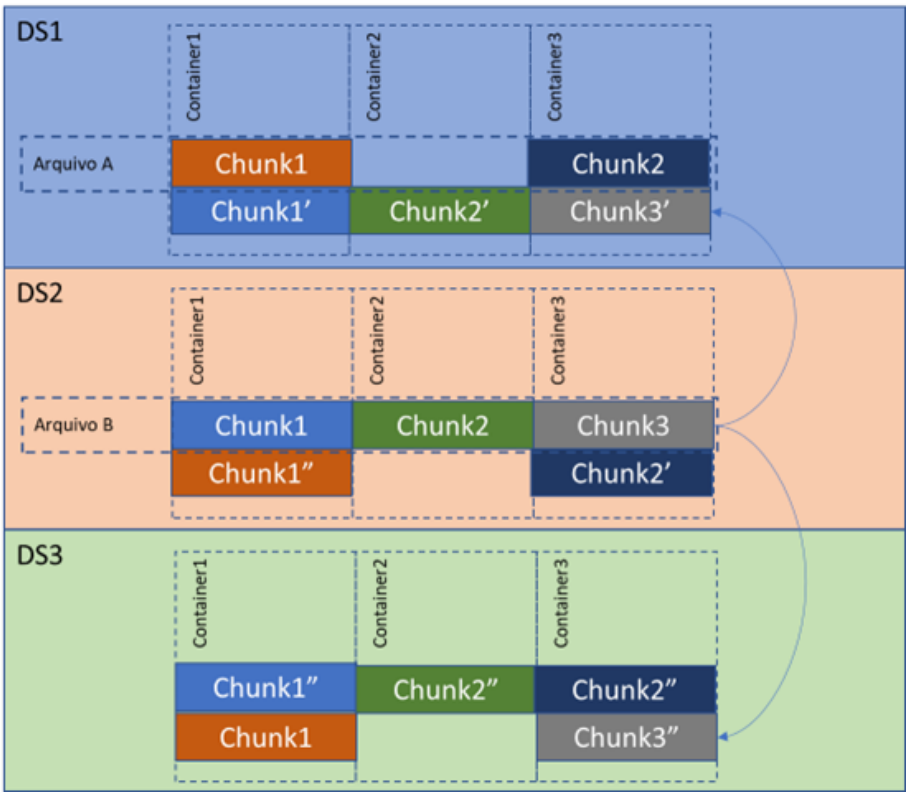

Figura 23 - Posição do nodo variando entre replicas do *container* durante escritas concorrentes de dois arquivos

5.7.2 Container Service

O serviço de controle de *containers* do AwareFS tem como objetivo gerenciar a CL, criando e adicionando entradas para novos *containers* que sejam necessários. Ele também é responsável pela organização e gerenciamento do cluster de DSs. O CS executa em apenas um nó do cluster, fazendo uso de múltiplas threads. Seu funcionamento é baseado em uma classe TContainerService e em um serviço Thrift chamado ContainerService.

5.7.3 Locking Service

O serviço de gerenciamento de *locks* do AwareFS tem como função controlar os bloqueios necessários para garantir a consistência dos dados em um uso concorrente onde solicitações de leitura e atualização de dados e metadados precisam ser atendidas em paralelo. Para implementar esse papel, uma classe TLockingService compatível com uma execução em múltiplas threads foi criada para atender e controlar

solicitações de bloqueio. Cada solicitação de bloqueio é representada por um objeto Thrift chamado aw_lock_request com as informações descritas no item 4.10.4. No protótipo do AwareFS, diferente do CS e do DS, o LS não tem um processo próprio e é instanciado dentro de cada processo de DS.

Uma função secundária do LS é controlar os ponteiros para os serviços Thrift dos DSs do cluster e *locks* necessários para que várias threads possam se comunicar com os DSs de forma concorrente.

A classe TLockingService tem como dados principais as filas de *locks* concedidos e de solicitações em espera, possuindo os seguintes dados:

- Granted_Llist: lista com um objeto aw_lock_request para cada solicitação de bloqueio atendida. Essa lista é um *container* C++ do tipo std::deque ("double ended queue") para permitir inserções e remoções com maior desempenho.
- Waiting_Llist: lista com um objeto aw_lock_request para cada solicitação que não pôde ser atendida. Também é implementada com a classe C++ std::deque.
- dsVect: vetor contendo um ponteiro do tipo Thrift DataServiceClient para cada DS do cluster que tenha relacionamento com o DS associado ao LS em questão.
- DS_lock_vec: um vetor contendo um lock do tipo omp_lock_t para cada DS do cluster que tenha relacionamento com o DS associado ao LS em questão.
- wList_lock: um lock to tipo omp_nest_lock_t para controlar o uso concorrente da lista Waiting_Llist.
- gList_lock: um lock to tipo omp_nest_lock_t para controlar o uso concorrente da lista Granted_Llist.

5.7.4 Data Service

Para servir requisições de dados dos clientes e de outros componentes do AwareFS, uma instância do Data Service é executada em cada servidor do cluster. Cada

DS é responsável por controlar o armazenamento de *chunks* e inodes usando os discos locais de cada servidor. Assim como o CS, o DS também faz uso de várias threads de execução e seu funcionamento é baseado em uma classe TDataService e em um serviço Thrift chamado DataService. Todo objeto da classe TDataService possui um objeto da classe TLockingService para fazer o controle de bloqueios relacionados aos *chunks* e inodes que armazena.

Semelhante ao Container Service, o Data Service também tem seu funcionamento baseado em uma classe TDataService e um serviço Thrift chamado DataService.

5.7.4.1 Criação de *container* e cadeia de replicação

A criação de *container* e sua cadeia de replicação é feita no método Poll() que trata todas as mensagens entre DS e CS através do comunicador MPI_COMM_WORLD. A primeira mensagem tratada pelo método Poll é a AWFS_CSDS_RCID que serve para o CS comandar a criação de um *container* com um identificador específico. O método Poll cria uma nova instância TContainer, incluindo as entradas na lista Cmap e newCIDs. O CS envia em seguida a mensagem AWFS_CSDS_STRP para o DS *master* da cadeia de replicação para que ele coordene a inicialização da cadeia e das replicações. O DS *master* dispara o processo de inicialização mandando as mensagens AWFS_DSDS_CRRP e AWFS_DSDS_CRCH para todos os membros da cadeia de replicação. O método Poll trata a mensagem AWFS_DSDS_CRRP, criando o objeto TContainer especificado e atualizando a lista Cmap e também trata a mensagem AWFS_DSDS_CRCH criando um comunicador MPI específico para a cadeia de replicação que é armazenado nos vetores MPIcommVec. Como último passo no método Poll, apenas o DS *master* da cadeia usa o comunicador recém-criado específico da cadeia de replicação para enviar a mensagem AWFS_DSDS_SCCM para todos os membros da cadeia de replicação comandando a criação sincronizada do mapa do *container*, necessária para que todas as réplicas de um objeto TContainer tenha uma visão consistente sobre qual DS tem o papel de owner de um nodo e se o nodo foi invalidado (veja 4.8.1).

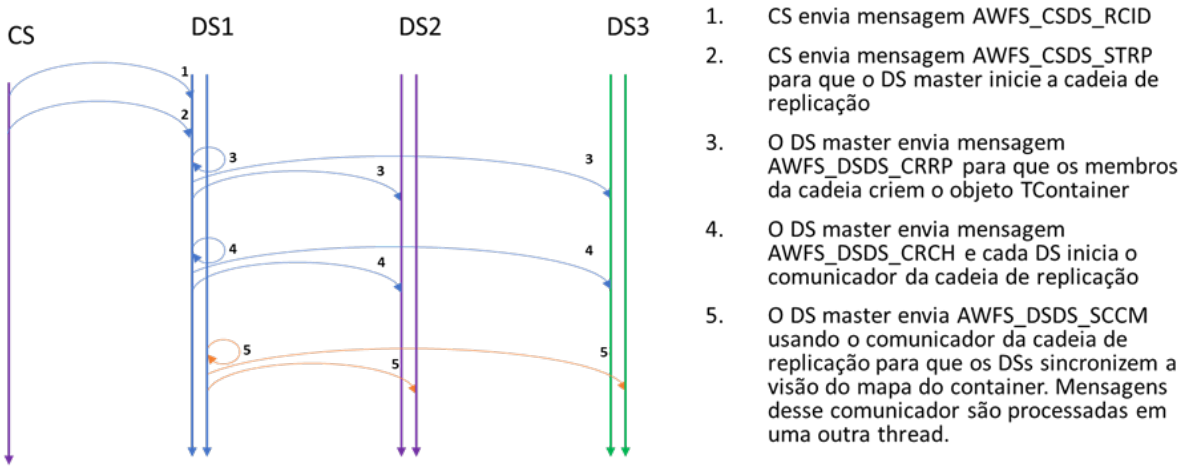

Figura 24 - Troca de mensagens para criação de um novo *container* e sua cadeia de replicação

5.7.4.2 Criação de Checkpoints

No protótipo do AwareFS é o CS quem comanda a criação de novos checkpoints, enviando uma mensagem AWFS_CSDS_CKPT para o DS *master* da cadeia de replicação do *container*. O DS *master* da cadeia, ao receber AWFS_CSDS_CKPT, envia uma mensagem AWFS_DSDS_SCCM para todos os membros da cadeia. Executando em uma thread separada, processando apenas mensagens de uma única cadeia de replicação através de comunicador MPI específico, é o método PollChain2() que vai processar a mensagem AWFS_DSDS_SCCM para criar o mapa de *container* de forma sincronizada entre os DSs membros da cadeia (passo 2 da Figura 25). Ao tratar essa mensagem o DS *master* faz um broadcast (MPI_Ibcast) do mapa da sua cópia do *container*, sincronizando na cadeia de DSs a visão de qual DS é dono de qual nodo. Em seguida o DS *master* envia uma mensagem AWFS_DSDS_CRSN para que todos os membros da cadeia de replicação criem um checkpoint do *container* sendo tratado. Ao receber uma mensagem AWFS_DSDS_CRSN o método PollChain2() chama o método freeze() do objeto TContainer interrompendo atualizações no *container*, esperando que o mesmo seja feito por todas as réplicas usando a função MPI_Allreduce como forma de sincronismo. O método CheckPoint() do objeto TContainer é então chamado para persistência dos dados de controle do *container* e criação de um novo

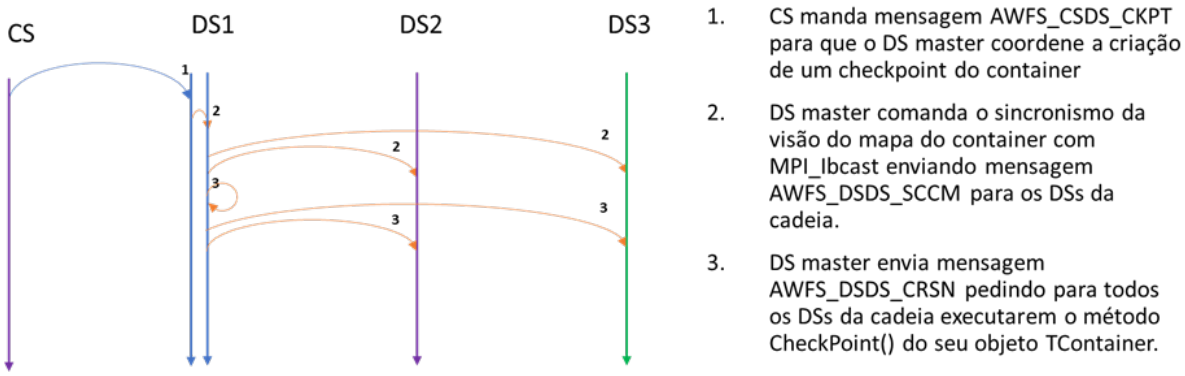

Figura 25 - Troca de mensagens para criação de checkpoints de um *container*

diretório com os dados do *container* como descrito em 4.8.3.1. As atualizações no *container* são liberadas chamando o método Melt() do objeto TContainer.

5.7.4.3 Replicação de dados

Para fazer a cópia dos dados entre o DS dono de um chunk (ou *inode*) e os demais DSs na cadeia de replicação o DS dedica uma thread para cada cadeia de replicação, ou seja, uma thread para cada comunicador armazenado em MPI-commVec. A replicação de dados é feita de forma assíncrona com uma sequência das mensagens AWFS_DSDS_REPC, AWFS_DSDS_REPW, AWFS_DSDS_REPS e AWFS_DSDS_GETC.

O processo de replicação inicia depois que o DS dono da cópia primária faz uma operação de escrita e pede a invalidação da réplica por uma chamada síncrona (chamada RPC com Thrift) do método Invalidate(). Então o DS pede para o objeto TContainer marcar o nodo como inválido e notifica a thread de consumeReplicaRequests que existem inodes/chunks inválidos que precisam ser atualizados. A thread com o método consumeReplicaRequests() do DS entra em execução, identifica quais inodes/chunks estão inválidos e envia uma mensagem AWFS_DSDS_REPC para o DS que é o owner do nodo. Ao receber uma mensagem AWFS_DSDS_REPC o DS obtém o identificador do nodo a ser replicado e responde com uma mensagem AWFS_DSDS_REPW. O DS desatualizado recebe a mensagem e executa uma thread

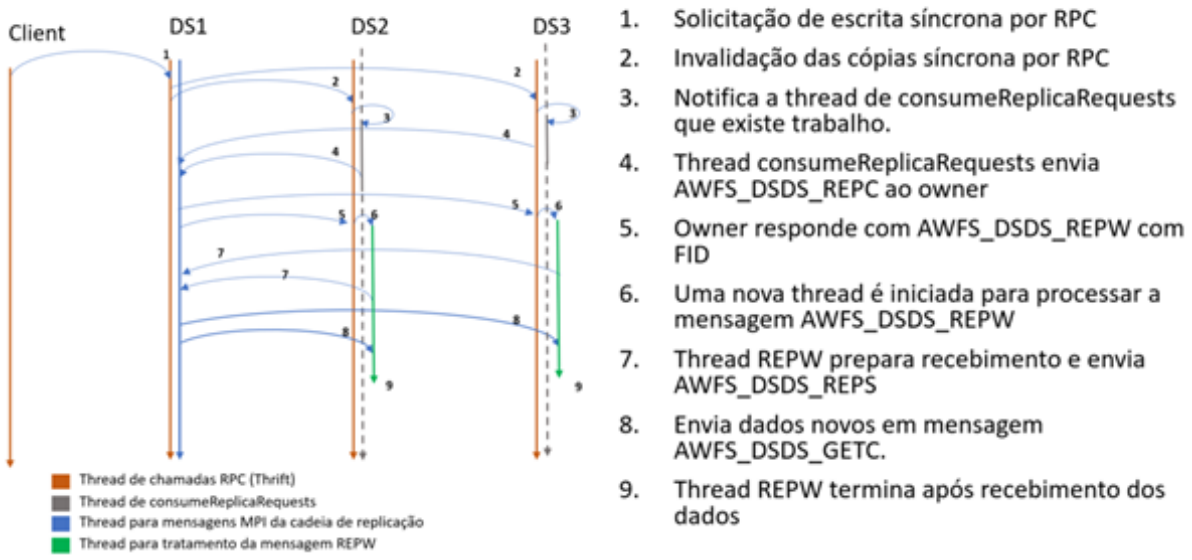

Figura 26 - Troca de mensagens e acionamento de threads para replicação de dados

em segundo plano para proceder a replicação que é feita enviando uma mensagem AWFS_DSDS_REPS para o DS owner e iniciando em seguida o recebimento de uma mensagem AWFS_DSDS_GETC que enviará de fato os dados desatualizados. A execução da thread em segundo plano é importante para liberar a thread da cadeia de replicação para atender outras solicitações de replicação.

Essa estratégia de trocas de mensagens se fez necessária pois o MPI exigia que a chamada de MPI_Recv já estivesse iniciada antes da chamada MPI_Send para envio dos dados. Assim, como descrito na Figura 26, as mensagens AWFS_DSDS_REPC, AWFS_DSDS_REPW e AWFS_DSDS_REPS têm curta duração e organizam o processo de replicação criando as estruturas necessárias e atualizando dados de controle sendo que a transferência dos dados ocorre na mensagem AWFS_DSDS_GETC.

5.7.4.4 Transferência do papel de nó dono do chunk

Como descrito em 4.13, é preciso uma transação atômica para alterar de forma consistente em todos os nós da cadeia de replicação a indicação de qual DS da cadeia está representando o papel de nó dono de um *chunk* ou inode. Para esse fim, depois

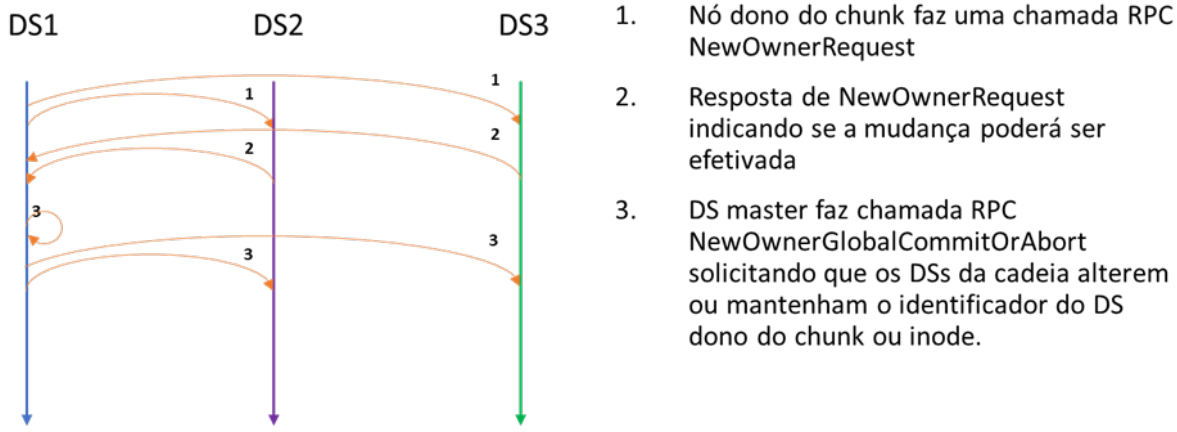

Figura 27 - Chamadas RPC para mudança do papel de dono do chunk ou inode

de obter um lock exclusivo do *chunk* ou *inode*, o AwareFS segue um processo de comprometimento em duas fases (*two phase commit*) usando o método SendNORQ ("*Send New Owner Request*") no qual o DS requisitando a mudança do papel de dono executa a primeira fase chamando o método RPC NewOwnerRequest para os outros DSs da cadeia de replicação. O método NewOwnerRequest responde concordando com a operação de mudança ou discordando caso o novo dono não possa assumir esse papel. Com as respostas dos DSs ao método NewOwnerRequest, o método SendNORQ segue para a segunda fase executando o método NewOwnerGlobalCommitOrAbort, onde todos os DSs da cadeia de replicação, igualmente, alteram ou não a identificação de dono do nodo (*chunk* ou *inode*) para o identificador do novo dono.

5.7.5 ClientAware - o cliente básico

Um cliente básico chamado ClientAware foi criado para implementar interações básicas com o AwareFS, além de possibilitar testes básicos de leitura e escrita. Escrito em C++11, o ClientAware implementa as seguintes funções:

- **Open**: usada para abrir um arquivo para escrita, a função Open recebe uma string com o nome do arquivo a ser aberto e retorna uma estrutura inodeHdr (citado anteriormente em 0) para ser passada para a função Write.
- **Create**: essa função tem como objetivo criar um arquivo dentro do AwareFS co-

locando nele os dados copiados de um arquivo presente no sistema de arquivos padrão da máquina cliente. Ela recebe como parâmetro duas strings, uma com o nome do arquivo a ser criado no AwareFS e outra com o nome do arquivo local com os dados a serem copiados.

- **Read**: função criada para ler um arquivo do AwareFS e gravar em um arquivo do sistema de arquivos local. Recebe duas strings como parâmetro, uma com o nome do arquivo a ser lido no AwareFS e outra com o nome do arquivo local a ser criado com os dados lidos.

- **List**: A função List tem como objetivo listar as informações sobre inode (nodo de cabeçalho) e chunks de um arquivo. Ela recebe como parâmetro o nome de um arquivo presente no AwareFS e lista o identificador FID do seu inode seguido pelo identificador do DS que é dono desse inode e da lista de DSs que possuem réplicas desse inode. Em seguida as mesmas informações são listadas para cada chunk que compõe o arquivo.

- **Write**: A função write escreve um buffer em uma posição informada (offset) de um arquivo no AwareFS. Ela recebe como parâmetro a estrutura inodeHdr do arquivo, um inteiro com a posição inicial a ser escrita no arquivo (offset), o número de bytes a serem escritos e o buffer com os dados a serem gravados.

- **main**: A função main do ClientAware tem por objetivo inicializar as estruturas para comunicação RPC via Thrift e fazer chamadas as funções Create, Read, Write e List, segundo o que for especificado na linha de comando. A saída de ajuda do ClientAware é a exposta na Figura 28.

O ClientAware recebe como parâmetros indicações de arquivo de entrada e saída, operação a ser executada e parâmetros específicos da operação, além de configurações básicas do AwareFS como tamanho do cluster, host com o CS e portas TCP/IP.

As operações possíveis são ‘c’ (create), ‘r’ (read), ‘w’ (write) e ‘l’ (list), sendo que para a operação ‘w’ é possível usar a opção ‘–pattern’ para passar uma string que

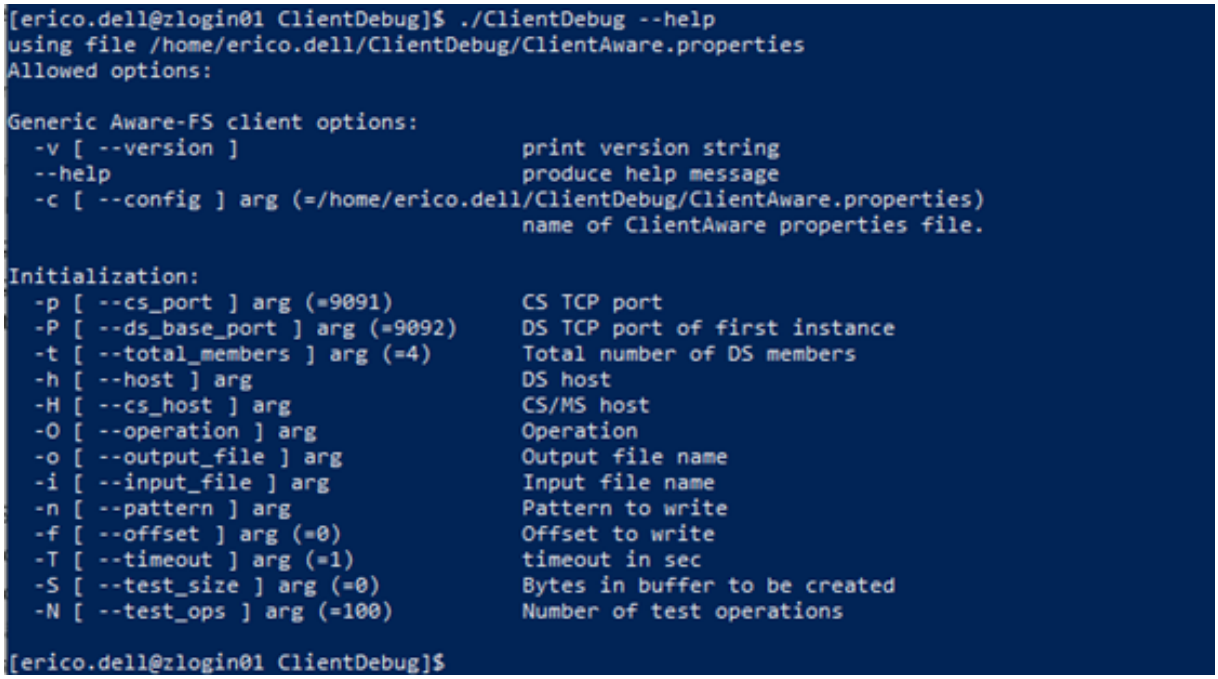

Figura 28 - Ajuda do cliente básico ClietAware

será usada como um padrão a ser escrito no arquivo para fins de testes de escrita.

Caso o nome de arquivo de entrada (especificado em '–input_file') seja igual a "shutdown", o ClientAware passa uma mensagem para que o CS finalize a execução de todos os serviços que compõem o cluster AwareFS.

5.8 O cliente FUSE do AwareFS

Considerando a importância da compatibilidade com a interface POSIX que os sistemas operacionais possuem para interagir com aplicações e com o usuário final, o protótipo usado para avaliação do AwareFS foi construído com uma interface baseada na biblioteca FUSE, do inglês "*file system in user space*" (FUSE, 2013). A biblioteca FUSE possibilita a criação de interface para sistemas de arquivos sem a necessidade de desenvolvimento de código para execução em modo privilegiado.

A interface FUSE do AwareFS torna a interação com usuário transparente, seguindo o padrão tradicional de sistemas Unix, possibilitando o uso de comandos triviais como ls para listar o conteúdo de um diretório, rm para remover um arquivo ou até

mesmo cat para criação de arquivos.

Cada cliente AwareFS FUSE é executado em várias linhas de execução concorrentes, beneficiando o desempenho de acessos simultâneos ao sistema de arquivos. Para cada arquivo acessado, uma estrutura de dados é criada para armazenar referências para o MS, para o CS e para os diversos DSs do cluster, além de outras variáveis de controle e um vetor de buffers, formando assim um contexto devidamente inicializado para acessar o arquivo e que é reutilizado enquanto houver arquivos em uso.

O cliente AwareFS FUSE conta com um recurso de "bufferização" de dados para acelerar a resposta às requisições de leitura e escrita. Todos os acessos aos dados são feitos usando uma comunicação com um DS e, independentemente do tamanho solicitado para leitura, o DS vai retornar todo o conteúdo do *chunk* em questão, iniciando no ponto da leitura. Todo esse dado é armazenado em um buffer no contexto de acesso ao arquivo, de forma que leituras subsequentes para pontos adjacentes do *chunk* não resultarão em comunicação com um DS pela rede. De forma análoga, escritas subsequentes e adjacentes são acumuladas no buffer do contexto e uma comunicação com um DS para persistir o dado só ocorre ao alcançar o final do *chunk* ou se a nova solicitação de escrita não for para um trecho adjacente. Qualquer DS na cadeia de replicação do *container* em questão pode atender uma requisição de leitura e no caso de requisições de escrita, o último DS que foi identificado como owner é indicado pela variável lko presente nos metadados, uma vez que apenas o DS dono da cópia primária atenderá requisições de escrita. Para casos em que o *chunk* está sendo criado e por isso lko=0, qualquer DS da cadeia de replicação pode ser utilizado, dando preferência para o DS que coexista com o cliente FUSE em um mesmo host.

Outras operações de metadados como rmdir, rename e unlink são transformadas em solicitações tipo RPC e transmitidas ao MS.

6 EXPERIMENTOS E RESULTADOS OBTIDOS

Considerando que o AwareFS é um sistema de arquivos distribuído, que aproveita a localidade do dado para viabilizar um processamento mais eficiente dos dados, são elencados as seguintes avaliações para a presente pesquisa que demonstram vantagens sobre sistemas de arquivos voltados para Big Data como é o caso do HDFS:

- **Metadados distribuídos**: Como as referências de arquivos estão distribuídas nos vários DSs do ambiente, será possível verificar a eficiência do acesso concorrente de dados, determinando se o controle de metadados influencia na escalabilidade do sistema de arquivos distribuído.
- **Reescrita de dados**: Uma vez que o AwareFS possibilita reescrita de dados, é relevante avaliar como esse tipo de escrita ocorre e seu desempenho. É importante comprovar que o protocolo de escrita local utilizado pelo AwareFS traz benefícios quando comparado com protocolos de escrita remota.
- **Acesso randômico**: Considerando a semântica WORM do HDFS, o AwareFS flexibiliza o seu uso para aplicações que precisem atualizar espaços intermediários dos arquivos, viabilizando seu uso também com aplicações mais tradicionais.
- **Sistema de controle de locks**: Sistemas como o HDFS não permitem acessos de escrita concorrentes já que a semântica imposta impede esse perfil de escrita. Para o caso do AwareFS, a avaliação dos resultados deve comprovar que o controle de *locks* proposto viabiliza operações de leitura e escrita, mostrando como o uso do recurso de *locks* pode ser uma alternativa para evitar as restrições impostas pela semântica WORM.

Avaliações de acesso randômico envolvendo solicitações concorrentes incluindo reescrita também foram feitas, já que o AwareFS visa atender também aplicações tradicionais e de computação científica, demonstrando o desempenho, o dinamismo e a

flexibilidade do sistema de armazenamento proporcionado. A seguir são destacadas vantagens do AwareFS inicialmente descrevendo pontos chave de sua interface com o usuário, que cobre aspectos importantes do padrão POSIX. Posteriormente outras avaliações são feitas utilizando ferramentas de benchmark amplamente utilizadas na avaliação de sistemas de arquivos distribuídos que suportam o padrão POSIX. O objetivo é comprovar o benefício do protocolo de escrita local empregado pelo AwareFS bem como a escalabilidade horizontal do AwareFS possível com sua arquitetura, seu controle distribuído de *locks* e seu gerenciamento de metadados distribuído.

6.1 Perfis de escrita

Em (GLUSTER. . . ,) os autores do GlusterFS elencam algumas alternativas para a avaliação de um sistema de arquivos distribuído, citando que para acessos em arquivos grandes, comum em cargas de trabalho de Big Data, as formas de medir o desempenho depende das características do acesso:

- **Acessos sequenciais em arquivos grandes**: o ideal é observar as taxas de transferência (megabytes por segundo), avaliando a taxa de transferência de dados. Especialmente ao lidar com grandes volumes de dados sequenciais, medir o volume de dados lido/gravado por unidade de tempo é apropriado, por ser comum em cenários de streaming, como leitura ou gravação de arquivos grandes.
- **Acessos randômicos em arquivos grandes**: o ideal é observar a eficiência em termos de quantas operações de E/S são executadas por segundo, ou seja, a latência e a capacidade de manipular operações de E/S de maneira rápida e eficiente. Acessos randômicos ocorrem em aplicações com padrões de acesso imprevisíveis e não lineares aos dados armazenados, por exemplo quando várias aplicações leem e gravam o mesmo arquivo.

Quando o tamanho do arquivo é pequeno (até poucos quilobytes) o ideal é observar o número de arquivos manipulados por unidade de tempo. Nessa primeira versão

do AwareFS, optou-se por um foco na manipulação de arquivos grandes e o serviço de metadados (MS) foi desenvolvido de forma simplificada.

Dessa forma, as avaliações descritas nas seções a seguir terão medidas em termos de MB/s para acessos sequenciais e medidas em IOPS (operações de E/S por segundo) para acessos randômicos, sempre buscando medir o desempenho de leitura e escrita, sequencial e randômica, em processos concorrentes, que são características dos perfis de E/S de cargas de trabalho de Big Data, de aplicações científicas e de aplicações tradicionais.

6.2 Protocolo de escrita local

Como descrito no item 4.8.1, sistemas de arquivos baseados em primário concentram operações de escrita em apenas um dos nós. No caso do MapR-FS, por exemplo, é o nó *master* da cadeia de replicação que atenderá as operações de escrita propagando em seguida para os outros nós com réplicas. O AwareFS por sua vez, tenta transferir o papel de "*chunk owner*", dono da cópia primária do *chunk*, para o nó solicitando a escrita, caso nele resida uma réplica do *chunk* em questão. Dessa forma, como citado na introdução do capítulo 4, temos um benefício por tirar vantagem da localidade do dado. O processo de transferência do papel de cópia primária é a base do protocolo de escrita local do AwareFS e seu benefício foi avaliado comparando-se os mesmos processos de escrita usando um sistema de arquivos com protocolo de escrita remota (foi usado o MapR-FS) e usando o AwareFS e seu protocolo de escrita local. A avaliação foi feita através de duas configurações para os processos de escrita:

- **Escrita concentrada**: Todos os arquivos foram criados usando o primeiro nó do cluster, sendo que cada *chunk* tem réplicas em outros dois nós. Esse tipo de teste mostra o quanto o processo de escrita pode ser impactado em casos onde só um nó é usado como ponto de entrada para o cluster. Mesmo tendo o mesmo nó como ponto de entrada para todos os *chunks*, suas réplicas ficam distribuídas de forma aleatória por todo o cluster, seguindo a cadeia de replicação de cada

container.

- **Escrita distribuída**: Todos os arquivos criados previamente no processo de escrita concentrada são então reescritos usando outros nós do cluster, fazendo com que um *chunk*, cuja cópia primária estava inicialmente no primeiro nó do cluster, seja reescrito por outro nó causando a migração da cópia primária de um nó pro outro, observando os benefícios do protocolo de escrita local do AwareFS quando comparado com uma escrita semelhante trafegando o dado pela rede no caso de um protocolo de escrita remota.

Usando como exemplo um conjunto de quatro arquivos em um cluster de apenas quatro nós, a Figura 29 mostra como seria a distribuição de escrita e suas respectivas cópias primárias e secundárias, onde cada célula representa uma réplica de um *chunk* e cada coluna representa a combinação de *chunks* de um arquivo (F1, F2, F3 e F4), sendo que as cores mais vivas indicam as cópias primárias e as cores mais pálidas indicam as cópias secundárias. A parte inferior da Figura 29 mostra a nova configuração depois que as cópias prmárias migram para o nó onde o processo de escrita está ocorrendo. Observando a Figura 29, considerando por exemplo as gravações feitas no primeiro *chunk*, o teste de escrita concentrada vai fazer com que a cópia primária seja criada no primeiro nó, fazendo com que o DS1 seja o *owner* inicial do *chunk*. A Figura 30 ilustra o comportamento de escrita para um *chunk* com as mudanças do papel de *owner*. A Figura 30 – (a) mostra que no teste de escrita concentrada os *chunks* são gravados no DS1 e replicados para os DSs 2 e 3. A Figura 30 – (b) mostra as solicitações de escrita sendo iniciadas nos quatro nós com um protocolo de escrita remota, obrigando o dado a ser enviado pela rede até o DS1. A Figura 30 – (c), mostra que em um protocolo de escrita local, os dados são persistidos sem trafegar pela rede, desde que haja uma cópia local para o *chunk* modificado, por isso apenas as solicitações originadas no quarto DS vão causar movimentação de dados pela rede. Essa sequência de testes concentrados e distribuídos foi executada em laboratório usando ambiente e procedimentos descritos a seguir.

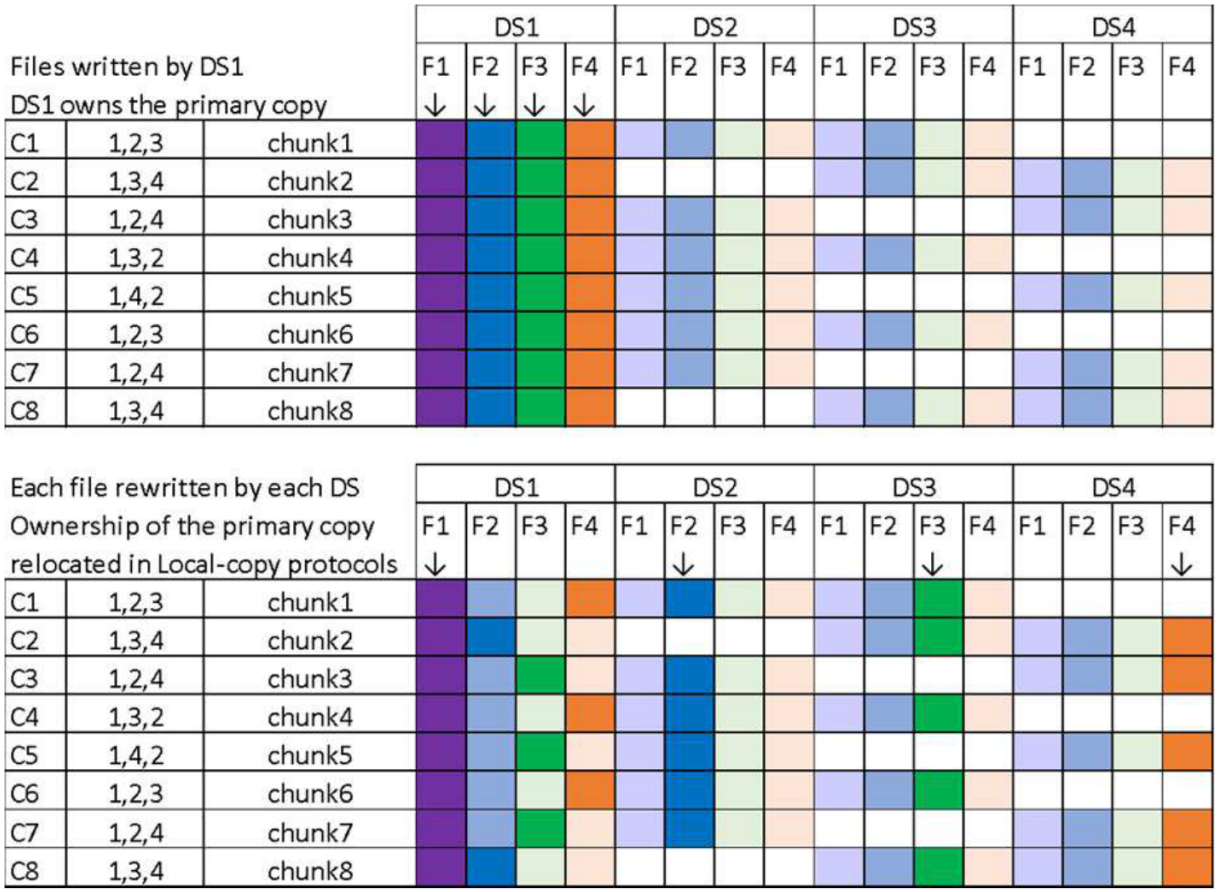

Figura 29 - (acima) distribuição de *chunks* no teste de escrita concentrada e (abaixo) distribuição de *chunks* no teste de escrita distribuída

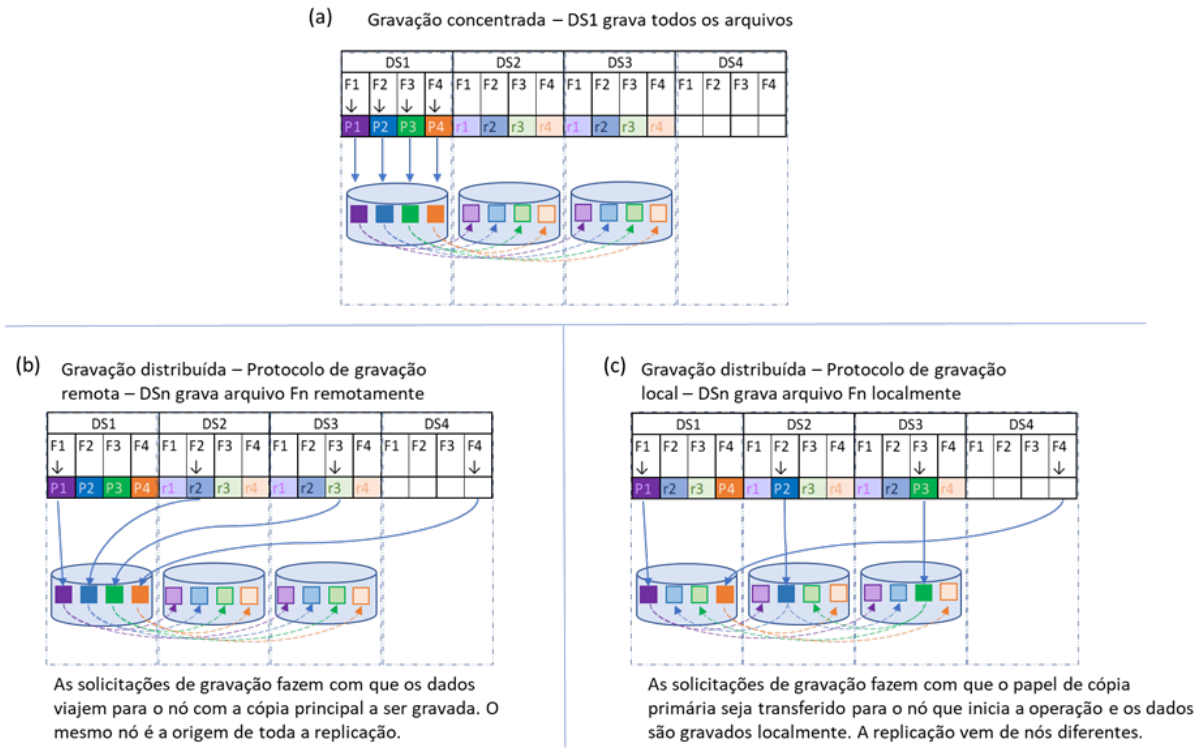

Figura 30 - Comportamento de solicitações de escrita

6.2.1 Ambiente de execução

Para os testes foi usado um cluster Dell EMC vxRail com 4 nós, cada um com 1xIntel Xeon Silver 4110 @2.10GHz, 8 cores p socket (16 processadores logicos), 127.62 GB RAM, 4 discos SAS Flash de 1,09TB, e 2x interfaces de rede 10GbE. O sistema hypervisor foi um VMware ESXi versão 7.0.3. O gerenciador de cluster foi um VMware vSphere com sistema de armazenamento vSAN 7.0.3.

Foram usadas 6 máquinas virtuais sendo que cada nó físico ficou encarregado de dois nós virtuais com 6 vCPUs, 32 GB de RAM e discos de 500 GB, usando Linux CentOS versão 7.7.1908.

6.2.2 Benchmark e ajustes para comparação

Usando o IOR (SHAN; SHALF, 2007), os seis arquivos de 1GB cada foram criados usando o primeiro nó do cluster de seis nós. O IOR foi iniciado com parametrização de escrita sem *page-caching*, usando seis *tasks* (uma em cada nó), com escrita de blocos

de 128Kbytes. Esse processo foi feito inicialmente usando o MapR-FS e os tempos foram coletados. Foram desabilitados os recursos de compressão e "*write-back cache*" do MapR-FS para que as medidas não fossem impactadas pelas otimizações proporcionadas por essas capacidades e representassem também o tempo inerente da transferência de dados para a cópia primária usando a rede.

Considerando que o MapR-FS usa devices de bloco para armazenamento e o AwareFS usa arquivos normais do sistema operacional, os discos apresentados para o MapR-FS foram loop-devices do Linux (pseudo devices que usam arquivos do sistema operacional como armazenamento), possibilitando uma comparação melhor dos tempos obtidos com os dois sistemas de arquivos distribuídos.

Para avaliar o tempo gasto durante a leitura e gravação nos discos, configuramos os DSs do AwareFS para também não completarem operações de escrita antes de gravar dados nos discos. Isso é fundamental para entender como o AwareFS pode se comportar em situações de E/S que devem ser efetivadas nos discos para garantir a persistência dos dados.

6.2.3 Resultados – Eficiência dos protocolos

Tanto os testes de escrita concentrada como os testes de escrita distribuída foram executados usando o mesmo cluster de seis nós, primeiro com o sistema de arquivos com protocolo de escrita remota e depois com o sistema de arquivos com protocolo de escrita local. Foram observados resultados onde o sistema de arquivos com protocolo de escrita remota apresentou medidas relativamente mais baixas, mas é preciso salientar que isso ocorreu também devido aos recursos desabilitados para viabilizarem as comparações. Em (E. C. Da Silva; L. Matsumoto Sato; E. T. Midorikawa, 2021) os autores destacam que o processo de transferência do papel de cópia primária tem um custo computacional que não ocorre quando um protocolo de escrita remota é utilizado.

Para o caso do protocolo de escrita remota, a Figura 31 mostra que o teste de es-

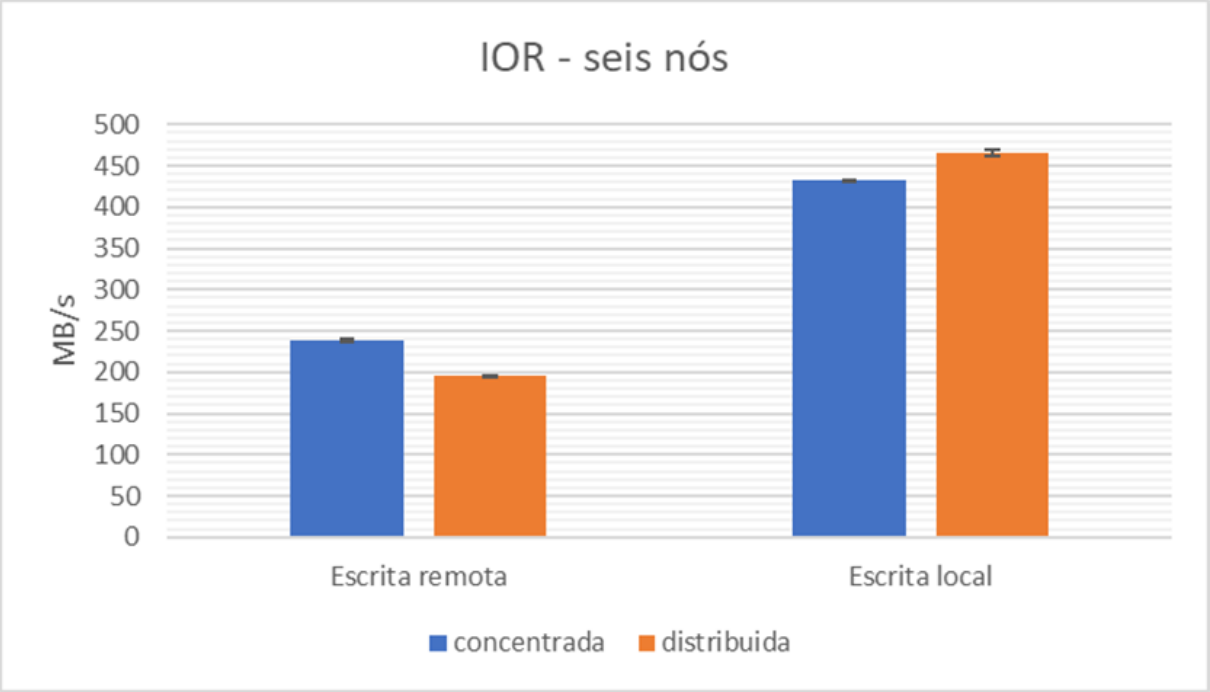

Figura 31 - Testes de escrita com IOR

crita distribuída (reescrevendo os *chunks* com cópias primárias em outros nós) apresentou uma perda de desempenho de cerca de 18% em comparação ao teste de escrita concentrada. Por outro lado, o protocolo de escrita local permitiu que o mesmo desempenho fosse alcançado tanto no teste de escrita concentrada como no teste de escrita distribuída, apresentando até um ganho da ordem de 8% devido à melhor distribuição de overhead por todo o cluster. Barras de erro foram adicionadas no gráfico da Figura 31 com base em valores na Tabela 3, considerando que o teste foi executado 100 vezes, medindo-se a taxa de escrita em MB/s, calculando-se a média aritmética, o desvio padrão e os intervalos de confiança de 95%, utilizando-se distribuição normal. Comparando os resultados obtidos com o protocolo de escrita remota do MapR-FS com os obtidos com o protocolo de escrita local do AwareFS, observou-se um ganho de desempenho de cerca de 140% no teste de escrita distribuída, que é razoável considerando o ganho de 80% observado no teste de escrita concentrada e as diferenças de configuração impostas ao MapR-FS para viabilizar a comparação entre os dois sistemas de arquivo, como por exemplo o uso de persistência no sistema de arquivos do Linux.

Tabela 3 - IOR - Intervalos de confiança

Teste	Protocolo	Taxa (MB/s)	Desvio Padrão	Int. de confiança
Concentrado	Escrita remota	238,978586	10,16399	[236,9865 : 240,9707]
Concentrado	Escrita local	431,830808	7,730645	[430,3156 : 433,3460]
Distribuído	Escrita remota	195,54899	2,975474	[194,9658 : 196,1322]
Distribuído	Escrita local	465,216869	20,08817	[461,2797 : 469,1541]

Observando como ficou a realocação das cópias primárias, observou-se que 34 dos 96 *chunks* tiveram sua cópia primária realocada, o que indica que cerca de 35% dos dados que necessariamente deveriam ser transferidos pela rede para o nó 1 foram reescritos localmente.

6.3 Desempenho e escalabilidade horizontal

O AwareFS foi pensado para atender perfis de uso diferenciados, podendo aumentar seu desempenho e capacidade de armazenamento com a adição de novos nós com hardware de baixo custo. Para avaliar como o AwareFS e sua arquitetura modular pode escalar desempenho e capacidade, uma série de testes foi executada para avaliar o desempenho em condições de leitura randômica, escrita randômica, escrita sequencial e leitura sequencial. Os mesmos testes foram executados utilizando requisições de 4KB, 128KB e 1MB, em tamanhos de cluster diferentes, incrementando o número de nós, buscando observar diferenças no desempenho à medida que o tamanho do cluster aumentava.

6.3.1 Ambiente de execução

Nesse conjunto de testes usamos de 8 a 36 nós de um cluster HPC onde os nós eram servidores Dell PowerEdge C6620 com uma arquitetura com 8 regiões NUMA, 2 processadores Intel Xeon Platinum 8480+ de 2 GHz com 56 núcleos por soquete, 512 GB de RAM a 4800 MHz e rede 1x1GbE interface. O sistema operacional foi RedHat Linux 8.6 com um sistema de arquivos XFS de 407GB em discos NVMe.

6.3.2 Benchmark

Para observar o comportamento do AwareFS sob diferentes perfis de uso, a ferramenta de benchmark utilizada foi o fio (AXBOE, 2014), que também possui diversas opções para configurar diferentes perfis de E/S, sendo especialmente útil para testar acesso aleatório em arquivos grandes com sua capacidade de executar operações durante um tempo especificado (LEE et al., 2021).

6.3.3 Leitura e escrita sequencial

Para avaliar o desempenho do AwareFS com a leitura e gravação de arquivos grandes, a ferramenta fio foi usada para ler um arquivo de 1 GB sequencialmente do início ao fim e depois reescrever o arquivo da mesma maneira. Um arquivo foi criado para cada thread de fio de cada DS no cluster e utilizado por ele nos testes, sendo executadas 4 threads de fio por DS. O benchmark fio para leitura foi executado em paralelo em todos DSs, permitindo observar como o AwareFS poderia ser capaz de gerenciar diversas operações concorrentes de I/O. Em seguida outra execução do fio foi feita para reescrever os arquivos, também sequencialmente de forma concorrente. Como cada DS era responsável por ler e depois escrever arquivos criados por ele mesmo, todos os *caches* disponíveis no FUSE e no sistema operacional foram utilizados em todo o seu potencial, aproveitando as vantagens obtidas com a localidade dos dados. Isso foi feito para avaliar um cenário real onde os dados são criados e manipulados por cada nó do cluster, como acontece com uma estrutura de processamento distribuído de Big Data como o Hadoop. Para avaliar a escalabilidade do AwareFS, executou-se os mesmos testes aumentando os nós disponíveis no cluster e aumentando proporcionalmente o número de arquivos, o tempo de execução e o número total de contêineres. Como cada nó executou 4 threads de testes fio, aumentou-se o número de arquivos gerenciados simultaneamente pelo AwareFS até um número igual ao número de nós no cluster multiplicado por 4. Após medições de velocidade, a média aritmética, o desvio padrão e intervalos de confiança de 95%, foram calculados utilizando distribuição

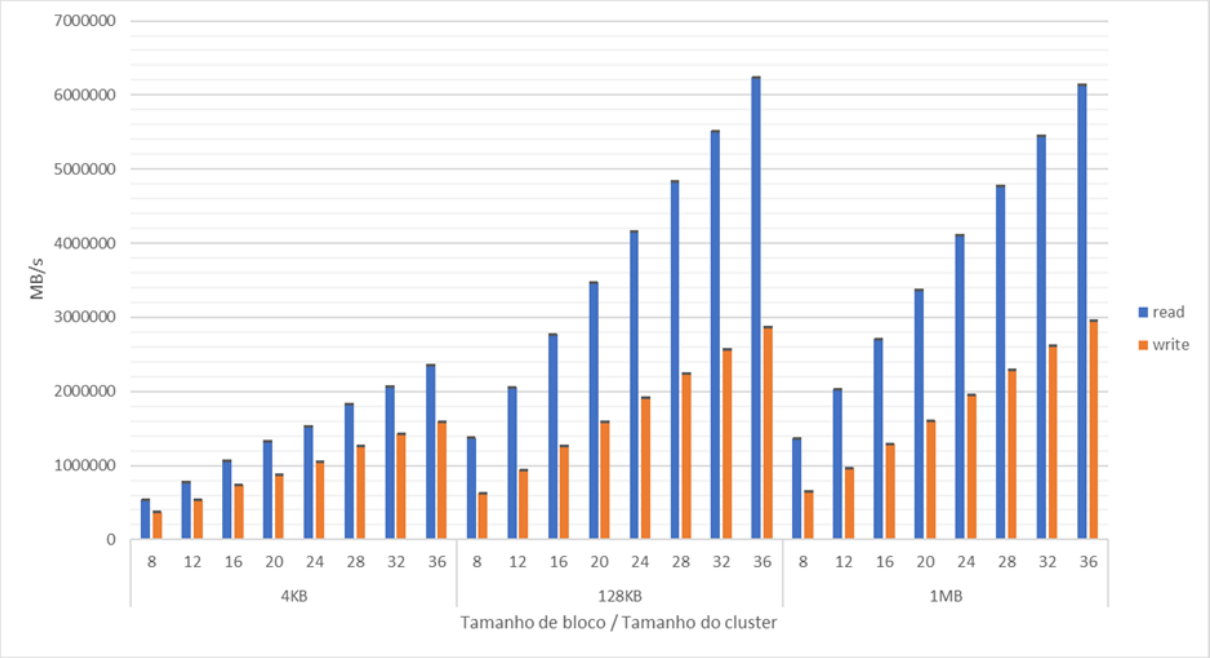

Figura 32 - Escrita e leitura sequencial - fio variando tamanhos de bloco e cluster

normal. O fio faz medições em intervalos de meio segundo, assim os tamanhos das amostras variaram de 3.833 medições para o cluster de 8 nós até 77.697 medições para o cluster de 36 nós.

A Figura 32 mostra claramente uma tendência de aumento do desempenho medido, conforme o tamanho do cluster e os números de *containers* e arquivos crescem. Detalhando mais os testes executados, a Tabela 4 mostra como escrita e leitura foram avaliadas, sempre com valores calculados usando uma significância de 5% para distribuição normal.

6.3.4 Leitura randômica

Um acesso completamente randômico, ou seja, em posições aleatórias do arquivo seria ineficiente para um processamento distribuído como o observado em cargas de trabalho de Big Data ou em aplicações científicas que procurassem explorar a localidade dos dados na execução das tarefas de forma distribuída. Contudo, elaborando a semântica imposta pelo sistema de arquivos para permitir acessos aleatórios dentro de um mesmo *chunk*, novos usos e algoritmos podem ser utilizados, viabilizando a ex-

Tabela 4 - Leitura e escrita sequenciais variando tamanhos de blocos e do cluster

				Leitura				Escrita			
Nós	Bloco	Arquivos	Conteineres	Taxa (MB/s)	Desvio Padrão	Amostras	Intervalo de confiança	Taxa (MB/s)	Desvio Padrão	Amostras	Intervalo de confiança
8	4KB	32	381	537881	13293	3812	[537459.02 : 538302.98]	372619	15572	3817	[372124.99 : 373113.01]
12	4KB	48	329	781600	16878	8603	[781243.35 : 781956.65]	538636	25314	8606	[538101.18 : 539170.82]
16	4KB	64	442	1069850	24938	15310	[1069454.98 : 1070245.02]	744060	34511	15321	[743513.54 : 744606.46]
20	4KB	80	656	1333728	28522	23930	[1333366.63 : 1334089.37]	879805	42590	23939	[879265.49 : 880344.51]
24	4KB	96	780	1536531	30938	34472	[1536204.41 : 1536857.59]	1055361	50320	34501	[1054830.03 : 1055891.97]
28	4KB	112	1018	1832242	38143	46944	[1831896.96 : 1832587.04]	1270539	60194	46961	[1269994.58 : 1271083.42]
32	4KB	128	1274	2065040	47450	61317	[2064664.43 : 2065415.57]	1433682	69087	61311	[1433135.14 : 1434228.86]
36	4KB	144	1545	2356984	52482	77616	[2356614.78 : 2357353.22]	1591445	76898	77615	[1590904.01 : 1591985.99]
8	128KB	32	381	1376645	53816	3833	[1374941.31 : 1378348.69]	636252	46846	3840	[634770.32 : 637733.68]
12	128KB	48	329	2059389	81113	8630	[2057677.67 : 2061100.33]	946459	71406	8627	[944952.21 : 947965.79]
16	128KB	64	442	2771280	108067	15345	[2769570.15 : 2772989.85]	1266526	94511	15357	[1265031.22 : 1268020.78]
20	128KB	80	656	3472820	133905	23981	[3471125.23 : 3474514.77]	1593582	118429	23980	[1592083.07 : 1595080.93]
24	128KB	96	780	4160973	160470	34533	[4159280.52 : 4162665.48]	1914162	141385	34538	[1912670.91 : 1915653.09]
28	128KB	112	1018	4841961	187058	47012	[4840270.09 : 4843651.91]	2240996	164060	47012	[2239512.98 : 2242479.02]
32	128KB	128	1274	5515430	212605	61373	[5513747.97 : 5517112.03]	2572483	187557	61393	[2570999.38 : 2573966.62]
36	128KB	144	1545	6244219	241437	77724	[6242521.64 : 6245916.36]	2878191	211472	77697	[2876704.04 : 2879677.96]
8	1MB	32	381	1370181	58824	3832	[1368318.53 : 1372043.47]	654142	47869	3834	[652626.78 : 655657.22]
12	1MB	48	329	2036779	87957	8630	[2034923.28 : 2038634.72]	969416	71465	8628	[967908.05 : 970923.95]
16	1MB	64	442	2710075	119294	15350	[2708187.82 : 2711962.18]	1291671	95191	15343	[1290164.78 : 1293177.22]
20	1MB	80	656	3374811	146136	23984	[3372961.54 : 3376660.46]	1612400	118761	23991	[1610897.21 : 1613902.79]
24	1MB	96	780	4112316	176037	34537	[4110459.44 : 4114172.56]	1956471	142107	34547	[1954972.49 : 1957969.51]
28	1MB	112	1018	4779510	204071	47015	[4777665.36 : 4781354.64]	2290769	165710	47000	[2289270.88 : 2292267.12]
32	1MB	128	1274	5454174	234093	61406	[5452322.47 : 5456025.53]	2617110	188361	61392	[2615620.01 : 2618599.99]
36	1MB	144	1545	6144762	262182	77724	[6142918.80 : 6146605.20]	2966524	210976	77719	[2965040.74 : 2968007.26]

ploração da localidade dos dados. Para avaliar o desempenho de acessos randômicos dentro de um mesmo *chunk*, mediu-se o desempenho de operações aleatória no AwareFS utilizando, simultaneamente em cada nó, um conjunto de threads de execução do benchmark fio e agregando os resultados obtidos (LEE et al., 2021). Configuramos todas as execuções do fio utilizando "*randomread*" ou "*randomwrite*" para executar leituras ou gravações variando o deslocamento de forma aleatória, porém sempre dentro de uma área de 64 MB do arquivo (o tamanho de um *chunk*), mudando para uma área diferente somente após a leitura ou gravação de 64 MB de dados. Assim como feito para os testes de acesso sequencial, o tempo de execução do fio também cresceu à medida que o tamanho do cluster utilizado foi aumentado. Para avaliação de desempenho de leitura randômica, as quatro threads de fio em cada nó foram executadas no modo "randomread", executando operações em *chunks* de um arquivo criado em um nó vizinho, eliminando assim os efeitos dos *caches* de sistema operacional. Esse mesmo modelo de avaliação foi repetido configurando o fio para executar operações de 4KB, 128KB e 1MB, possibilitando uma comparação do desempenho em termos de IOPS para requisições de tamanhos diferentes (Figura 33). As avaliações de leitura aleatória tiveram duração variando desde 60 segundos para um cluster de 8 nós,

até 270 segundos para o cluster de 36 nós. As medidas foram coletadas pelo fio em intervalos de meio segundo. O número de amostras variou de 442 medições o cluster de 8 nós até 68.912 medições o cluster de 36 nós. Para incluir barras de erros nos gráficos, a média aritmética das medições de IOPS, o desvio padrão e os intervalos de confiança de 95% foram calculados usando distribuição normal. As linhas de tendência nos gráficos da Figura 33 indicam um claro crescimento de desempenho à medida que o tamanho do cluster usado no benchmark também cresce. É natural que a aleatoriedade das solicitações de leitura afete a eficácia do buffer existente no cliente FUSE do AwareFS. Em sua configuração inicial, o cliente FUSE do AwareFS utiliza um mecanismo de leitura do tipo “read ahead” criando um buffer para leituras subsequentes, onde todos os dados desde o offset de leitura até o final do *chunk* do arquivo são extraídos do DS e armazenados em buffer, fazendo com que solicitações de leitura subsequentes sejam atendidas sem comunicação com qualquer DS para os casos em que os dados necessários já estiverem armazenados no buffer local.

6.3.5 Escrita randômica – avaliação do protocolo de escrita

Como descrito em 4.8, 4.12 e 6.1, o protocolo de escrita local baseado em primário do AwareFS promove a migração do papel de cópia primária de um DS para outro, para os casos onde a solicitação de escrita se origine no mesmo nó onde reside um DS com uma cópia válida do *chunk*. Com essa estratégia, o dado não precisa trafegar pela rede e a localidade do dado é explorada, ou seja, a cópia local é utilizada para atualizações.

Considerando que em operações de acesso randômico é mais usual uma avaliação do número de operações de E/S por unidade de tempo Para avaliar a eficiência do protocolo de escrita local do AwareFS com escritas randômicas, as avaliações consistiram em três diferentes passos.

- a) Passo 1 (move disabled): Foi desabilitada a capacidade de movimentação do papel de “*owner*” do *chunk*, isto é, do papel de cópia primária, e uma avaliação

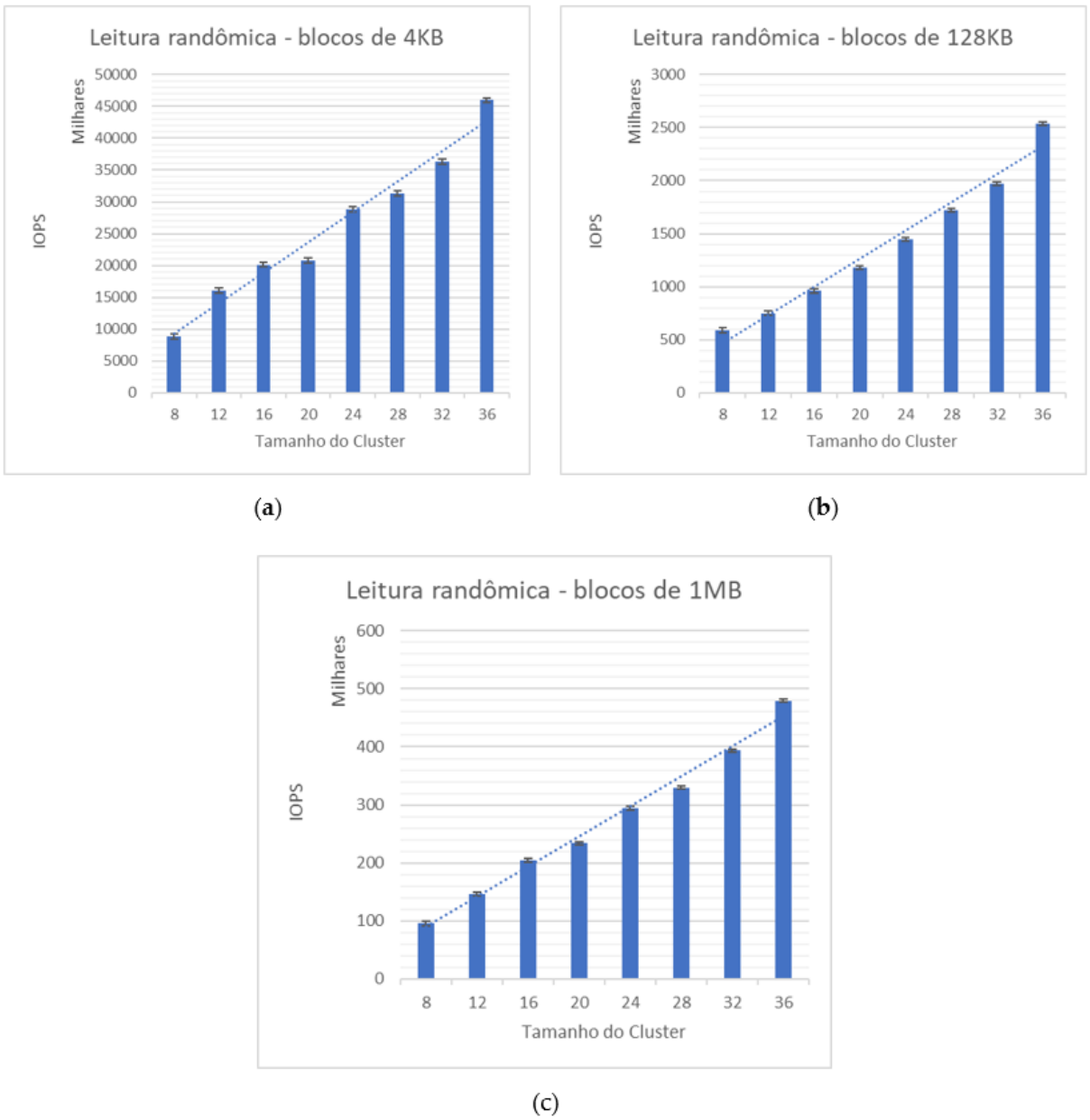

Figura 33 - Leitura randômica de um arquivo criado em um nó vizinho com tamanhos de bloco de (a) 4KB, (b) 128KB e (c) 1MB

de desempenho foi feita executando o fio no modo “randomwrite”. Dessa forma foram coletadas medidas de IOPS sem explorar a localidade do dado, obrigando a movimentação do dado a ser escrito até o nó usado na criação do *chunk*, aquele que inicialmente é o *owner* do *chunk*.

- b) Passo 2 (move enabled): A capacidade de movimentação do papel de “*owner*” do *chunk* foi reabilitada e uma nova avaliação de desempenho foi feita reexecutando o fio no modo “randomwrite”, repetindo os mesmos offsets de escrita. Assim foram coletadas novas medidas de IOPS, dessa vez explorando a localidade do dado, já que o papel de *owner* do *chunk* migrou do nó vizinho para o nó iniciando as requisições de escrita, sempre que uma cópia local estava disponível.

- c) Passo 3 (move enabled – 2nd round): O processo do passo 2 é repetido e novas medidas de IOPS foram coletadas para observar o desempenho do processo de escrita depois da migração do papel de owner do *chunk*. Esse terceiro passo foi para observar o desempenho sem o overhead próprio do processo de migração do *owner*.

Para permitir comparações, os três passos são executados em sequência, repetindo as mesmas solicitações de escrita geradas aleatoriamente no passo 1.

Considerando esses três passos executados em um cluster de 12 nós, com as características descritas em 6.3.1, utilizando solicitações de 128KB, observamos claramente na Figura 34 um ganho de 9% comparando o passo 2 (*move enabled*), que é quando o dado não precisa ser movimentado pela rede, com o passo 1 (*move disabled*), que não aproveita a localidade dos dados. Também observamos que o passo 3 tem um desempenho ainda maior, sendo cerca de 13% melhor que o observado no passo 1, já que o papel de cópia primária já estava com o nó iniciando as escritas.

O número de IOPS observados com requisições maiores é menor por conta do tempo gasto na movimentação/escrita de dados requerida por solicitação, resultando em menos requisições de E/S por segundo, ou seja, um número menor de IOPS. Executando os mesmos três passos, no mesmo ambiente com requisições de escrita

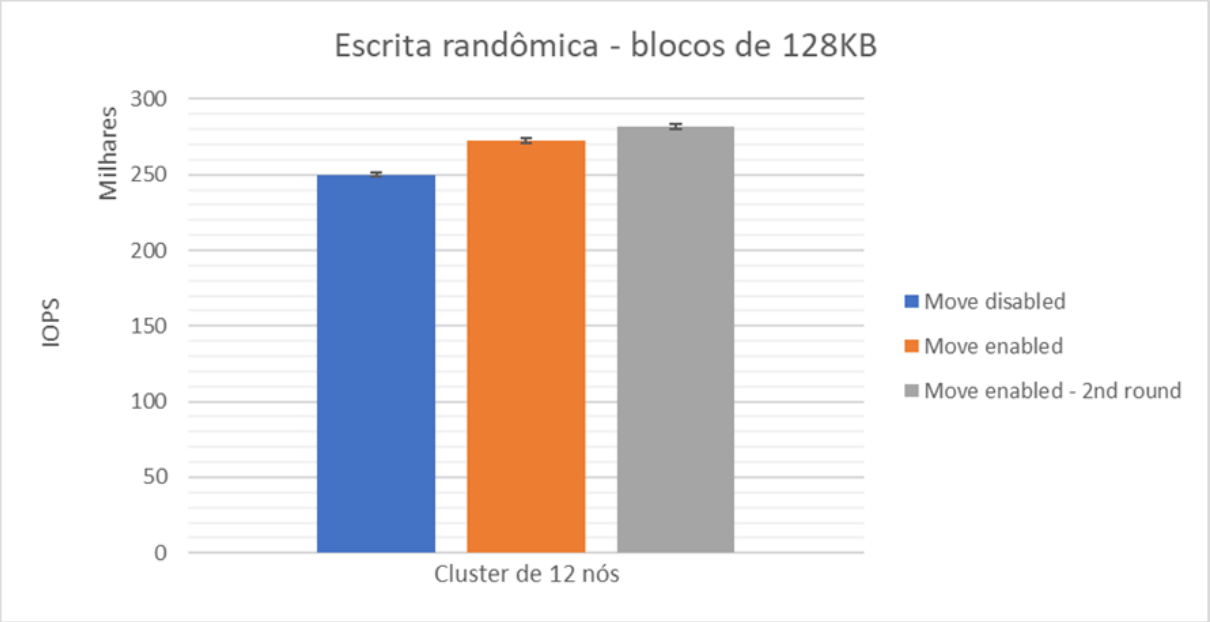

Figura 34 - Influência da migração do papel de cópia primária em escritas randômicas com cluster de 12 nós e blocos de 128KB

de blocos de 4KB, foi observado um comportamento ainda de ganho a cada passo, porém com diferenças mais acentuadas ao final, Figura 35.

O número de IOPS medido no terceiro passo para requisições de 4KB foi cerca de 43% maior que o observado no primeiro passo. Isso acontece pois o número de operações executadas é maior por trabalhar com blocos menores, melhorando a taxa de resposta por operação graças ao protocolo de gravação local que evita comunicação com outros DSs pela rede. Além disso, com um tamanho de escrita menor, aplicações que requeiram escritas randômicas concentradas em um mesmo *chunk* podem se beneficiar do esquema de cache implementado no cliente FUSE do AwareFS.

Repetindo o processo de avaliação de três passos com requisições de 1MB, o ganho de desempenho foi de 17% comparando o uso do protocolo de escrita local (terceiro passo) com medidas coletadas no passo 1, que não move o papel de cópia primária antes da escrita. Como o uso de blocos de 128KB garantiu uma melhora de 13%, blocos de 1MB demonstraram um ganho maior por conta do tamanho das transferências pela rede que ocorreram.

Para avaliar se esse comportamento, observado para o cluster de 12 nós, se man-

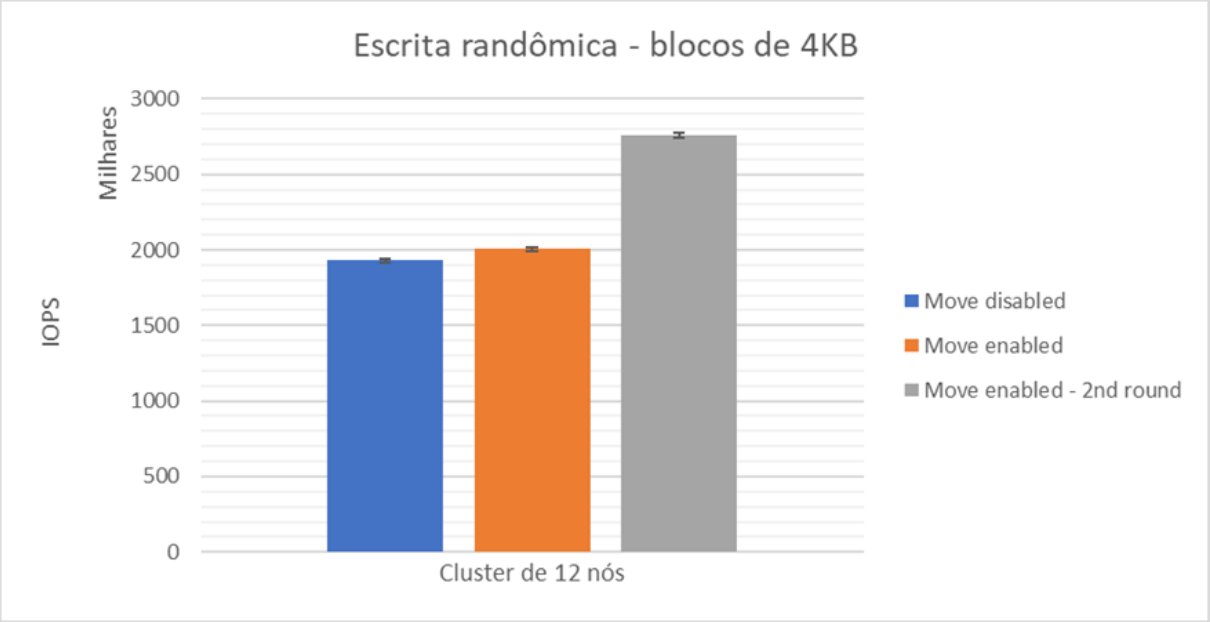

Figura 35 - Influência da migração do papel de cópia primária em escritas randômicas com cluster de 12 nós e blocos de 4KB

teria com outros tamanhos de clusters, o processo de avaliação de três passos foi repetido para tamanhos de cluster variando de 8 a 36 nós, variando também o tempo de execução do benchmark. O fio foi iniciado com quatro threads por nó, impondo ao AwareFS uma carga variando de 32 até 144 arquivos de 1GB sendo atualizados pelas execuções de fio gerando requisições concorrentes. A mesma avaliação foi feita com requisições de blocos de 4KB, 128KB e 1MB.

O gráfico da Figura 36 mostra que o desempenho geral de gravações aleatórias aumentou junto com o tamanho do cluster, observando comportamento semelhante com os três tamanhos de bloco (4 KB, 128 KB e 1 MB). O comportamento observado com um cluster de 12 nós, também foi observado para clusters variando de 8 a 36 nós.

Comparando os DSs que inicialmente eram *owner* dos *chunks*, ou seja, eram donos da cópia primária, com os DSs que terminaram como *chunk owner* após todas as etapas de avaliação, temos uma taxa de mudança de propriedade que diminuiu conforme mostrado na Tabela 5. Mesmo com essa queda, o desempenho do medido utilizando o protocolo de gravação local ainda foi superior ao observado sem tal protocolo. Mesmo para clusters maiores ficou claro o benefício do protocolo de escrita

Tabela 5 - Número total de *chunks* versus *chunks* cujo papel de cópia primária migrou

Nós no cluster	Duração (s)	Total de chunks	Troca de owner	Taxa de troca
8	60	512	59	11.5%
12	90	768	55	7.2%
16	120	1024	84	8.2%
20	150	1280	80	6.3%
24	180	1536	76	4.9%
28	210	1792	73	4.1%
32	240	2048	49	2.4%
36	270	2304	50	2.2%

local, mostrando como o desempenho de gravação pode se beneficiar da localidade dos dados, quando o processo de migração do papel de cópia primária do AwareFS é empregado.

6.4 Operações de metadados e arquivos pequenos

Ainda que o gerenciamento de arquivos pequenos não tenha sido o foco central na primeira implementação do AwareFS, algumas avaliações foram feitas e são descritas aqui para demonstrar a funcionalidade. Dessa forma, o benchmark MD-Workbench (MD-Workbench, 2023) foi utilizado para uma verificação inicial do desempenho alcançado com o AwareFS no gerenciamento de arquivos pequenos. O MD-Workbench é um benchmark paralelo, baseado em MPI, para avaliação de desempenho de metadados e E/S com arquivos pequenos. Diferente de outros benchmarks de metadados, o MD-Workbench gera padrões de acesso que não são facilmente armazenáveis em cache.

Um cluster foi criado na nuvem Microsoft Azure, com cinco máquinas virtuais do tipo *Standard_B2ms* com 2 vCPUs e 8GiB de RAM e 2 discos SSD de 16GiB. Após ajustes no processo de instalação do MapR para ignorar requisitos mínimos de CPU e memória, foi instalado o MapR-FS nesse ambiente, apenas para que o mesmo benchmark pudesse ser executado em outro sistema de arquivos distribuído de Big Data, para que os resultados verificados pudessem servir como base de comparação para melhor avaliação dos valores observados com o AwareFS.

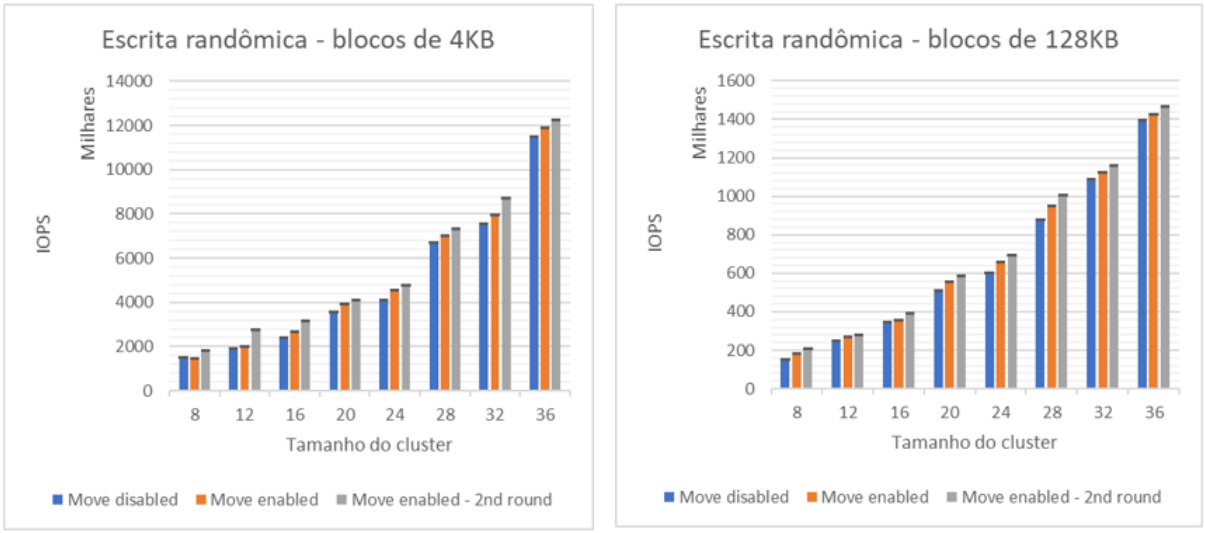

Figura 36 - Influência da migração do papel de cópia primária em escritas randômicas para diferentes tamanhos de bloco e diferentes tamanhos de cluster (a) utilizando blocos de 4KB; (b) utilizando de blocos de 128 KB; (c) utilizando blocos de 1 MB

Tabela 6 - Resultados do benchmark MD-Workbench para o MapR-FS

MapR-FS					
min (s)	**max (s)**	**média (s)**	**desvio padrão**	**Objetos (Arquivos ou diretórios)**	**IOPS**
0,26	0,52	0,38	0,1	200	1542,0
0,26	0,53	0,40	0,1	200	1520,8
0,26	0,57	0,40	0,1	200	1398,7
0,25	0,54	0,38	0,1	200	1492,6
0,25	0,52	0,38	0,1	200	1540,2
0,26	0,56	0,40	0,1	200	1424,4

O MD-Workbench foi aplicado, tanto no AwareFS como no MapR-FS, parametrizado para executar avaliações manipulando 10 arquivos por diretório, usando 4 diretórios criados com 40 arquivos cada. Todos os arquivos com um tamanho de 3901bytes. Cada avaliação é constituída de três fases: criação de diretórios e arquivos, benchmark propriamente dito e limpeza. Na fase de benchmark primeiro os metadados de um arquivo são obtidos (função stat do padrão POSIX), depois esse arquivo é aberto, lido, fechado e removido, finalmente um novo arquivo é criado, 3901 bytes são gravados nele e por fim ele é fechado. A fase de benchmark foi repetida para cada um dos 10 arquivos manipulados por diretório. Todo o processo de avaliação foi distribuído por 5 ranks de MPI, um para cada nó. A Tabela 6 e a Tabela 7 mostram os valores obtidos para seis execuções do benchmark tanto para o MapR-FS como para o AwareFS.

A Tabela 6 mostra que o MapR-FS apresentou valores de IOPS consideravelmente altos, com tempos abaixo de meio segundo por processo MPI, mesmo instalado em um ambiente com muito menos recursos que o recomendado pelos desenvolvedores.

O MapR-FS conta com um controle de metadados distribuído em que pelo menos três nós do cluster respondem por operações de metadado, além de leitura e escrita de arquivos pequenos. O AwareFS, na sua primeira versão ainda usa um único nó para responder por operações de metadado, incluindo criação e remoção de arquivos. Para essa avaliação de metadados, o AwareFS foi instalado com o MS (Metadata Service) compartilhando o primeiro nó do cluster com o CS (Container Service) e com um DS (Data Service). Isso ocorreu pois os outros ambientes não comportavam a instalação do MapR-FS, mas esse fato afetou o desempenho observado para o

Tabela 7 - Resultados do benchmark MD-Workbench para o AwareFS

AwareFS					
mínimo (s)	**máximo (s)**	**média (s)**	**desvio padrão**	**Objetos (Arquivos ou diretórios)**	**IOPS**
11,39	13,20	12,69	0,7	200	60,6
12,96	15,26	14,08	1,1	200	52,4
13,77	16,19	15,07	0,9	200	49,4
14,83	17,72	16,68	1,1	200	45,1
11,88	14,32	12,78	1,0	200	55,9
12,07	15,91	14,32	2,0	200	50,3

AwareFS já que o nó na Azure contava com recursos limitados, impedindo a resposta de mais requisições concorrente pelo MS. A Tabela 7 mostra valores cerca de 30 vezes piores para o AwareFS que os alcançados com o MapR-FS. Esse resultado se justifica, pois, como dito anteriormente, essa primeira versão do MS foi criada com foco na manipulação de arquivos grandes (maiores que 64MB), usuais no contexto de BigData e computações científicas. A otimização de operações de metadados e armazenamento de arquivos pequenos é relevante para aplicações com manipulação de arquivos pequenos, devendo ser abordada em trabalhos futuros.

7 CONCLUSÃO E TRABALHOS FUTUROS

Garantir a proximidade dos dados aos recursos computacionais traz benefícios já explorados por tecnologias de Big Data, que utilizam hardware de baixo custo em clusters projetados para condições em que a falha de um componente não é uma exceção. A localidade dos dados quando explorada, acelera o processamento evitando que os dados trafeguem pela rede, dispensando o uso de componentes conectividade, hardware e software, que tenham um custo mais elevado. Ainda que existam sistemas de arquivos distribuídos que explorem a localidade dos dados em operações de leitura, cargas de trabalho usam protocolos de escrita remota onde atualizações são direcionadas para o nó responsável pela cópia primária, sem explorar a localidade dos dados em requisições de escrita. Neste trabalho foi apresentado o AwareFS, um sistema de arquivos distribuído horizontalmente escalável que explora a localidade dos dados proporcionando leitura e escrita, sequencial e randômica, consistente e coerente, usando um sistema de gerenciamento de locks distribuído. O protocolo de escrita local baseado em primário do AwareFS garante a coerência, explorando a localidade dos dados também durante a escrita. Associado ao sistema de gerenciamento de locks distribuído, os mecanismos de reescrita proporcionam maior desempenho sem prejuízo para a consistência dos dados.

Um amplo trabalho de avaliação foi feito usando benchmarks de mercado, com perfis de leitura e escrita, sequencial e randômica, provando capacidades de concorrência e eficiência do AwareFS, ressaltando os benefícios comprovados do protocolo de escrita local. Os resultados obtidos comprovaram a eficiência com perfis de E/S comuns em Big Data e também com características mais dinâmicas como os tradicionalmente possíveis com ferramentas de computação científica. Tais resultados comprovam a hipótese apresentada

Conclui-se que as principais contribuições deste trabalho são as proposições de mecanismos e estratégias baseadas na exploração da localidade de dados em escri-

tas de dados através da mudança da cópia primária e uso de locks distribuídos nas operações de leitura e escrita na concepção e implementação de um sistema de armazenamento eficiente capaz de cobrir requisitos de aplicações cientificas, sem deixar de cobrir requisitos de E/S próprios de cargas de trabalho de Big Data. A compatibilidade com o padrão POSIX, ainda que parcialmente implementada no protótipo, torna possível o uso do AwareFS também com aplicações que utilizam armazenamento compartilhado consideradas tradicionais, ou seja, que não usam MPI.

7.1 Trabalhos futuros

Mesmo com a robustez considerável apresentada pelo AwareFS já na sua primeira versão, espera-se um aprimoramento em trabalhos futuros com a implementação de detecção e recuperação de falhas dos componentes MS, CS, DS/LS e client, garantindo a resiliência importante em arquiteturas onde situações de falha não são uma exceção.

Também deseja-se incluir o armazenamento direto em dispositivos de blocos ("*block devices*") para aumentar o desempenho de escrita, incluindo capacidades de compressão de dados e de gerenciamento de volumes (espaço de armazenamento distribuído).

Quanto às capacidades de interface com o usuário, espera-se evoluir o cliente FUSE do AwareFS, tornando-o ainda mais completo ao incorporar um controle de cache mais eficiente, além de atender a outras exigências do padrão POSIX, como controles de acesso e permissões de leitura e escrita. Deseja-se também incluir outros padrões de cliente, especialmente um para o protocolo HDFS.

7.2 Publicações

Ao término da primeira fase de desenvolvimento deste trabalho, as diversas contribuições resultaram em material relevante para a elaboração de um artigo intitulado 'Distributed File System for Rewriting Big Data Files Using a Local-Write Protocol',

destacando a contribuição crucial do protocolo de escrita local para o armazenamento eficiente de dados. O artigo explora a localidade dos dados e o sistema distribuído de locks com leitura e escrita sequencial de arquivos grandes. Este trabalho encontra-se disponível na biblioteca digital IEEE Xplore e foi apresentado na ’2021 IEEE International Conference on Big Data’, realizada de 15 a 18 de dezembro de 2021.

Como desdobramento desse projeto, foi elaborado um novo artigo com o título ’Distributed File System to Leverage Data Locality for Large-File Processing’. Este segundo artigo foi desenvolvido após aprimoramentos no uso de múltiplas threads, levando em consideração os resultados de um novo processo de avaliação, que incluiu leitura e escrita randômica, variando o tamanho das requisições. Importante destacar que este artigo foi publicado no periódico ’Electronics’, da editora suíça MDPI.

REFERÊNCIAS

ADVE, S. V.; GHARACHORLOO, K. Shared memory consistency models: A tutorial. **computer**, v. 29, n. 12, p. 66–76, 1996. Publisher: IEEE.

AXBOE, J. Fio-flexible io tester. **URL http://freecode. com/projects/fio**, 2014.

BIJLANI, A.; RAMACHANDRAN, U. Extension framework for file systems in user space. In: **2019 USENIX Annual Technical Conference (USENIX ATC 19)**. [S.l.: s.n.], 2019. p. 121–134.

BLOMER, J. A Survey on Distributed File System Technology. **Journal of Physics: Conference Series**, v. 608, n. 1, p. 012039, abr. 2015. ISSN 1742-6596. Publisher: IOP Publishing. Disponível em: <https://dx.doi.org/10.1088/1742-6596/608/1/012039>.

CARNS, P. H.; III, W. B. L.; ROSS, R. B.; THAKUR, R. {PVFS}: A parallel file system for linux clusters. In: **4th Annual Linux Showcase & Conference (ALS 2000)**. [S.l.: s.n.], 2000.

CHANDRA, R. **Parallel programming in OpenMP**. [S.l.]: Morgan kaufmann, 2001.

CHANG, F.; DEAN, J.; GHEMAWAT, S.; HSIEH, W. C.; WALLACH, D. A.; BURROWS, M.; CHANDRA, T.; FIKES, A.; GRUBER, R. E. Bigtable: A distributed storage system for structured data. **ACM Transactions on Computer Systems (TOCS)**, v. 26, n. 2, p. 1–26, 2008. Publisher: ACM New York, NY, USA.

CHEN, M.; MAO, S.; LIU, Y. Big data: A survey. **Mobile Networks and Applications**, v. 19, abr. 2014.

CHOWDHURY, F.; ZHU, Y.; HEER, T.; PAREDES, S.; MOODY, A.; GOLDSTONE, R.; MOHROR, K.; YU, W. **I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning**. [S.l.: s.n.], 2019. Journal Abbreviation: ICPP 2019: Proceedings of the 48th International Conference on Parallel Processing Pages: 10 Publication Title: ICPP 2019: Proceedings of the 48th International Conference on Parallel Processing. ISBN 978-1-4503-6295-5.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. **Communications of the ACM**, v. 51, n. 1, p. 107–113, 2008. Publisher: ACM New York, NY, USA.

DEPARDON, B.; MAHEC, G. L.; SéGUIN, C. Analysis of six distributed file systems. 2013.

DOBRE, C.; XHAFA, F. Parallel Programming Paradigms and Frameworks in Big Data Era. **International Journal of Parallel Programming**, v. 42, n. 5, p. 710–738, out. 2014. ISSN 1573-7640. Disponível em: <https://doi.org/10.1007/s10766-013-0272-7>.

DUNNING, T.; FRIEDMAN, E. **AI and Analytics in Production: How to make it work**. [S.l.]: O'Reilly Media, 2018.

E. C. Da Silva; L. Matsumoto Sato; E. T. Midorikawa. Distributed file system for rewriting Big Data files using a local-write protocol. In: **2021 IEEE International Conference on Big Data (Big Data)**. [S.l.: s.n.], 2021. p. 3646–3655. Journal Abbreviation: 2021 IEEE International Conference on Big Data (Big Data).

ELIA, D.; FIORE, S.; ALOISIO, G. Towards HPC and big data analytics convergence: Design and experimental evaluation of a HPDA framework for escience at scale. **IEEE Access**, v. 9, p. 73307–73326, 2021. Publisher: IEEE.

FUSE, F. **Filesystem in userspace, 2013**. 2013. Disponível em: <http://fuse.sourceforge.net/>.

GANTZ, J.; REINSEL, D.; others. Extracting value from chaos. **IDC iview**, v. 1142, n. 2011, p. 1–12, 2011.

GEORGE, L. HBase - The Definitive Guide: Random Access to Your Planet-Size Data. 2011. ISSN 978-1-449-39610-7. Publisher: O'Reilly.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The Google file system. In: **Proceedings of the nineteenth ACM symposium on Operating systems principles**. [S.l.: s.n.], 2003. p. 29–43.

GLUSTER Administrator Guide. Disponível em: <https://docs.gluster.org/en/latest/Administrator-Guide/Performance-Testing/>.

HASHEM, I.; YAQOOB, I.; ANUAR, N.; MOKHTAR, S.; GANI, A.; KHAN, S. The rise of "Big Data" on cloud computing: Review and open research issues. **Information Systems**, v. 47, p. 98–115, jul. 2014.

HUA, X.; WU, H.; LI, Z.; REN, S. Enhancing throughput of the Hadoop Distributed File System for interaction-intensive tasks. **Journal of Parallel and Distributed Computing**, v. 74, n. 8, p. 2770–2779, 2014. Publisher: Elsevier.

J. Wang; D. Han; J. Yin; X. Zhou; C. Jiang. ODDS: Optimizing Data-Locality Access for Scientific Data Analysis. **IEEE Transactions on Cloud Computing**, v. 8, n. 1, p. 220–231, mar. 2020. ISSN 2168-7161.

KUBIATOWICZ, J. **CS194-24: Advanced Operating Systems Structures and Implementation – lecture 22**. 2014. Disponível em: <http://inst.eecs.berkeley.edu/~cs194-24/sp13/index_lectures.html.23/04/2017>.

LEE, J.-Y.; KIM, M.-H.; SHAH, S. A. R.; AHN, S.-U.; YOON, H.; NOH, S.-Y. Performance Evaluations of Distributed File Systems for Scientific Big Data in FUSE Environment. **Electronics**, v. 10, n. 12, 2021. ISSN 2079-9292.

LIAO, W.-k. Design and evaluation of MPI file domain partitioning methods under extent-based file locking protocol. **IEEE Transactions on Parallel and Distributed Systems**, v. 22, n. 2, p. 260–272, 2010. Publisher: IEEE.

MALHOTRA, R. An Efficient Locking Scheme for Path-based File Systems. 2014.

MD-Workbench. 2023. Disponível em: <https://www.vi4io.org/tools/benchmarks/md-workbench>.

MOORE, M.; BONNIE, D.; LIGON, B.; MARSHALL, M.; LIGON, W.; MILLS, N.; QUARLES, E.; SAMPSON, S.; YANG, S.; WILSON, B. OrangeFS: Advancing PVFS. **FAST poster session**, v. 124, p. 127, 2011.

MUTHITACHAROEN, A.; CHEN, B.; MAZIERES, D. A low-bandwidth network file system. In: **Proceedings of the eighteenth ACM symposium on Operating systems principles**. [S.l.: s.n.], 2001. p. 174–187.

NETTO, M. A.; CALHEIROS, R. N.; RODRIGUES, E. R.; CUNHA, R. L.; BUYYA, R. HPC cloud for scientific and business applications: taxonomy, vision, and research challenges. **ACM Computing Surveys (CSUR)**, v. 51, n. 1, p. 1–29, 2018. Publisher: ACM New York, NY, USA.

P, B. D.; MARCO, C.; others. **Understanding the linux kernel**. [S.l.]: O'Reilly, 2007.

PARALLEL Virtual File System, Version 2. Disponível em: <http://dev.orangefs.org/old/documentation/releases/current/doc/pvfs2-guide/pvfs2-guide.php>.

PATE, S.; BOSCH, F. V. D. **UNIX Filesystems: Evolution, Design and Impemenation**. [S.l.]: John Wiley & Sons, Inc., 2003. ISBN 978-0-471-16483-8.

PATGIRI, R.; AHMED, A. Big Data: The V's of the Game Changer Paradigm. dez. 2016.

RAKOWSKI, K. **Learning Apache Thrift**. [S.l.]: Packt Publishing Ltd, 2015.

RYDNING, D. R.-J. G.-J.; REINSEL, J.; GANTZ, J. The digitization of the world from edge to core. **Framingham: International Data Corporation**, v. 16, p. 1–28, 2018.

SAPUNENKO, V. **GPFS for advanced users**. 2014. Disponível em: <https://agenda.infn.it/event/8092/sessions/11347/attachments/51759/61118/GPFS_tutorial._Intro_.pdf>.

SATO, M.; KANZAKI, H.; KAWATA, S.; others. Performance Evaluation of Scale-out NAS for HDFS. In: **The Third International Conference on Advances in Information Mining and Management (IMMM)**. [S.l.: s.n.], 2013.

SCHMUCK, F.; HASKIN, R. GPFS: A Shared-Disk File System for Large Computing Clusters. **Proceedings of the FAST'02 Conference on File and Storage Technologies**, maio 2002.

SCHäLING, B. **The boost C++ libraries**. [S.l.]: Boris Schäling, 2011.

SHAFER, J.; RIXNER, S.; COX, A. L. The hadoop distributed filesystem: Balancing portability and performance. In: **2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)**. [S.l.]: IEEE, 2010. p. 122–133.

SHAN, H.; SHALF, J. Using IOR to analyze the I/O Performance for HPC Platforms. **Lawrence Berkeley National Laboratory**, 2007.

SHVACHKO, K.; KUANG, H.; RADIA, S.; CHANSLER, R. The hadoop distributed file system. In: **2010 IEEE 26th symposium on mass storage systems and technologies (MSST)**. [S.l.]: Ieee, 2010. p. 1–10.

SINGH, S. **Develop your own filesystem with FUSE**. 2014. Disponível em: <https://developer.ibm.com/articles/l-fuse/?mhsrc=ibmsearch_a&mhq=fuse>.

SPECIFICATIONS, B. Draft Standard for Information Technology—Portable Operating System Interface (POSIX®) Draft Technical Standard: Base Specifications, Issue 7. 2008.

SRIVAS, M.; RAVINDRA, P.; SARADHI, U.; PANDE, A.; SANAPALA, C.; RENU, L.; KAVACHERI, S.; HADKE, A.; VELLANKI, V. Map-Reduce Ready Distributed File System, 2011. **US Patent App**, v. 13, n. 162,439, 2017.

STERLING, T. L. **Beowulf cluster computing with Linux**. [S.l.]: MIT press, 2002.

SUMARAY, A.; MAKKI, S. K. A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform. In: **Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication**. New York, NY, USA: Association for Computing Machinery, 2012. (ICUIMC '12). ISBN 978-1-4503-1172-4. Event-place: Kuala Lumpur, Malaysia. Disponível em: <https://doi.org/10.1145/2184751.2184810>.

SUN, Z.; SHEN, J.; YONG, J. A novel approach to data deduplication over the engineering-oriented cloud systems. **Integrated Computer-Aided Engineering**, v. 20, n. 1, p. 45–57, 2013. Publisher: IOS Press.

T. D. Thanh; S. Mohan; E. Choi; S. Kim; P. Kim. A Taxonomy and Survey on Distributed File Systems. **2008 Fourth International Conference on Networked Computing and Advanced Information Management**, v. 1, p. 144–149, set. 2008.

TANENBAUM, A.; STEEN, M. van. **Distributed Systems: Principles and Paradigms**. Pearson Prentice Hall, 2007. ISBN 978-0-13-613553-1. Disponível em: <https://books.google.com.br/books?id=UKDjLQAACAAJ>.

TANENBAUM, A. S.; FILHO, N. M. **Sistemas operacionais modernos**. [S.l.]: Prentice-Hall, 1995. v. 37.

TANTISIRIROJ, W.; PATIL, S.; GIBSON, G. **Data-intensive file systems for internet services: A rose by any other name**. [S.l.], 2008.

UTA, A.; SANDU, A.; KIELMANN, T. Overcoming data locality: An in-memory runtime file system with symmetrical data distribution. **Future Generation Computer Systems**, v. 54, p. 144–158, 2016. Publisher: Elsevier.

VANGOOR, B. K. R.; AGARWAL, P.; MATHEW, M.; RAMACHANDRAN, A.; SIVARAMAN, S.; TARASOV, V.; ZADOK, E. Performance and Resource Utilization of FUSE User-Space File Systems. **ACM Trans. Storage**, v. 15, n. 2, p. Article 15, 2019. ISSN 1553-3077. Publisher: Association for Computing Machinery. Disponível em: <https://doi.org/10.1145/3310148>.

WANG, F.; ORAL, H. S.; SHIPMAN, G. M.; DROKIN, O.; WANG, D.; HUANG, H. **Understanding Lustre Internals**. United States, 2009. Disponível em: <https://www.osti.gov/biblio/951297>.

WEIL, S. A. Ceph: reliable, scalable, and high-performance distributed storage. 2007. Publisher: University of California, Santa Cruz.

WEIL, S. A.; BRANDT, S. A.; MILLER, E. L.; LONG, D. D.; MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In: **Proceedings of the 7th symposium on Operating systems design and implementation**. [S.l.: s.n.], 2006. p. 307–320.

WHAT is log4cxx. 2008. Disponível em: <http://logging.apache.org/log4cxx/index.html>.

WHITE, T. **Hadoop: The Definitive Guide**. O'Reilly, 2015. ISBN 978-1-4919-0163-2 1-4919-0163-2. Disponível em: <https://www.amazon.com/Hadoop-Definitive-Guide-Tom-White/dp/1491901632>.

WILLIAMS, A. **C++ concurrency in action**. [S.l.]: London, 2012.

YADAV, V. Working with HBase. In: . [S.l.: s.n.], 2017. p. 123–142.

YANG, X.; YIN, Y.; JIN, H.; SUN, X.-H. SCALER: Scalable parallel file write in HDFS. In: **2014 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.]: IEEE, 2014. p. 203–211.

ZAHARIA, M.; CHOWDHURY, M.; FRANKLIN, M. J.; SHENKER, S.; STOICA, I. Spark: Cluster computing with working sets. In: **2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)**. [S.l.: s.n.], 2010.

ZHANG, X.; XU, F. Survey of Research on Big Data Storage. In: . [S.l.: s.n.], 2013. p. 76–80.

ZHAO, D. Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems. **Illinois Institute of Technology**, 2014.

ZOU, J.; IYENGAR, A.; JERMAINE, C. Architecture of a distributed storage that combines file system, memory and computation in a single layer. **The VLDB Journal**, v. 29, n. 5, p. 1049–1073, set. 2020. ISSN 0949-877X. Disponível em: <https://doi.org/10.1007/s00778-020-00605-w>.

APÊNDICE A – CLASSES E MÉTODOS

A.1 Classe TContainer

Quando um objeto do tipo TContainer é criado, é feita uma verificação para identificar se existe o diretório local que armazena os dados do *container* e este é criado se não existir. Se o diretório existir, o mapa do *container* é criado a partir dos dados serializados e a lista CInodes é montada com base nos arquivos no diretório.

A.1.1 Metadados

No protótipo do AwareFS tanto dados como metadados são armazenados em *containers* idênticos. Contudo, a atualização de metadados como a lista de *chunks* que compõe um arquivo e o tamanho do arquivo é composta por uma manipulação de dados que precisa ser feita de forma eficiente evitando o acesso direto aos discos locais dos nós. Um inode é representado por um objeto Thrift do tipo inodeHdr sendo que a classe TContainer possui um vetor desses objetos (o headerCache) para mantê-los em memória. A classe TContainer usa os seguintes métodos para manipular metadados:

- AddFile: cria ou atualiza o inode (arquivo header com o nome iniciado por “h.”). Se for criado o inode, um primeiro *chunk* de dados já é adicionado ao *container* nesse momento, garantindo que dados e metadados coexistam em um mesmo *container* aumentando a eficiência no trato de arquivos pequenos. O arquivo header é a serialização de um objeto thrift inodeHdr. Com o objeto inodeHdr criado e persistido é adicionada uma entrada para o inode na lista CInodes. A função termina com uma chamada ao método SetHdr.

- SetHdr: inclui/altera uma entrada do tipo inodeHdr no vetor headerCache para que acessos posteriores a esse inode não necessitem de acesso a disco.
- GetHdr: recupera a estrutura inodeHdr de um inode a partir da memória ou desde o disco se está ainda não estiver no vetor headerCache.
- FlushHdr: persiste uma estrutura inodeHdr em disco devidamente serializada.
- FlushHeaderCache: persiste todas as entradas no vetor headerCache em disco.
- addChunkToFile: altera o inode incluindo ou alterando um *chunk* em um determinado arquivo. É o método chamado para acertar os metadados depois de alterados os dados de um *chunk* em um arquivo.

A.1.2 Dados

A classe TContainer possui uma pilha chamada WAL (do inglês " *write ahead log*") para manter em memória os dados de uma solicitação de escrita permitindo que ela retorne antes que os dados sejam persistidos em disco. Para controlar essa pilha e o trabalho com os dados existem métodos específicos para o tratamento do armazenados dos *chunks*, são eles:

- readChunk: lê um *chunk*, ou parte dele, a partir do disco.
- writeChunk: grava um *chunk*, ou parte dele, em disco.
- writeChunkToWAL: armazena em memória, na lista WAL.
- FlushWAL: para persistir em disco toda a pilha de solicitações de escrita.

A.1.3 Dados e Metadados

Alguns métodos para tratamento de inodes/chunks dos contêineres existem para ações necessárias tanto no tratamento de headers como no tratamento de *chunks*, são eles:

- CheckPoint: que persiste o mapa do contêiner e dados em memória e depois cria um diretório para as mudanças de dados subsequentes, com *hard links* do sistema de arquivos do Linux para os arquivos no diretório anterior (veja ítem 4.8.3.1).
- setOwnerVec: acerta o mapa do contêiner copiando as entradas de um ContainerMap de origem.
- InvalidateInode: marca um nodo como inválido no mapa do contêiner, criando as entradas desse nodo caso ele ainda não exista (em casos de replicação).
- IsInvalid: retorna se um inode/chunk foi invalidado.
- GetInodeOwner/SetInodeOwner: retorna ou altera o identificador de qual DS possui a cópia primária do inode/chunk.

A.1.4 Processamento concorrente

A classe TContainer possui ainda *locks* (variáveis de tipo omp_lock_t) para controlar o acesso concorrente à pilha WAL e ao headerCache. Um terceiro lock omp é utilizado para bloquear a thread quando o contêiner estiver em freeze (sem aceitar alterações) e o bloqueio for necessário.

A.2 Container Service

O serviço de controle de *containers* do AwareFS é baseado em uma classe TContainerService e em um serviço Thrift chamado ContainerService.

A.2.1 Classe TContainerService

A classe TContainerService possui como principais dados:

- ContainerList: um objeto thrift de tipo CL com a lista de *containers* e as cadeias de replicação (veja item 4.3).

- MPIclientVector: um vetor com uma entrada para cada DS contendo o comunicador MPI_comm iniciado pelo DS e o número de *containers* no DS.

- MPIcommVector: um vetor com uma entrada para cada *container* contendo o comunicador MPI_comm da cadeia de replicação do referido *container*.

No protótipo do AwareFS a classe TContainerService implementa os seguintes métodos:

- ConnectAll: publica o nome do serviço MPI e faz o merge do comunicador MPI_COMM_WORLD com os comunicadores iniciados separadamente por cada DS, gerando um comunicador entre todos os serviços que são iniciados um a um, de forma independente.

- createContainer: define um identificador para um novo *container*, atualizando a CL com a cadeia de replicação especificamente montada para ele.

- Poll: Para o tratamento das mensagens de criação de *container* e solicitação de uma versão atualizada da CL.

- ContinuousCheckpoints: envia solicitação para criação de checkpoint para o DS que é master da cadeia de replicação de cada *container*. A cada 5 segundos uma solicitação de checkpoint é gerada para um *container* e enviada. Depois de gerada solicitações para todos os *containers*, o ciclo se repete depois de um período definido (default 60 segundos).

- getCL: retorna a CL armazenada em memória.

- Init: esse procedimento é feito apenas uma vez na inicialização do CS e executa troca de mensagens com todos os DSs para criar a CL (veja item 4.3). Cada DS envia uma mensagem de CRCL (create CL) para o CS passando uma CL parcial com uma entrada para cada *container* identificado pelo DS contendo apenas o identificador do DS. O CS adiciona uma entrada para o CID (container ID) na CL global com o identificador do DS que originou a mensagem ou, caso já exista

uma entrada para o *container* na CL global, apenas adiciona o DS na cadeia de replicação do *container*. Exemplo:

- o CL global vazia e mensagem CRCL recebida com CID=1 e DS=3, resulta em uma nova entrada 1, 3 na CL global.
- o Mensagem CRCL recebida com CID=1 e DS=2, gera adição do DS2 na CL global, resultando na entrada 1, 3, 2.
- o Mensagem CRCL recebida com CID=1 e DS=12, gera adição do DS12 na CL global, resultando na entrada 1, 3, 2, 12.
- o Depois de criada uma cadeia de replicação de três componentes, o processo ignora novas entradas na CL parcial enviada na mensagem CRCL para o referente CID.
- o O CS repete o processo para todos os DSs em MPI_COMM_WORLD e ao final a primeira versão da CL global está gerada.
- o O CS faz um *broadcast* da CL para todos os DSs e o AwareFS está pronto para aceitar requisições de serviço.

A.2.2 Serviço ContainerService

O serviço Thrift ContainerService possui como dado um único objeto chamado CS da classe TContainerService e os seguintes métodos:

- getCL: retorna o resultado do método getCL do objeto CS.
- createContainer: retorna o identificador para um novo *container* gerado com o método createContainer do objeto CS.
- shutdown: manda mensagem para que todos os componentes do AwareFS sejam encerrados.

Esses métodos são chamadas RPC disponíveis para que os componentes do AwareFS possam solicitar ações do CS.

Na inicialização (função main), o serviço ContainerService começa identificando os parâmetros via linha de comando, arquivo de configuração ou valor default. Os parâmetros necessários para a execução do CS são:

- base_port: porta TCP a ser usada pelo CS, com valor default 9091.
- total_members: número total de DSs, com valor default 4.
- config: arquivo de parametrização, com valor default “AwareCS.properties”.

O serviço ContainerService é do tipo Thrift TThreadedServer que cria uma nova thread para cada conexão que fica ativa até que a conexão seja encerrada.

Mais duas outras threads de OpenMP são criadas na função main, uma para execução do método Poll() do objeto CS e outra para execução do método ContinuousCheckpoints().

A.3 Locking Service

O serviço de gerenciamento de locks do AwareFS tem como função controlar os bloqueios necessários para garantir a consistência dos dados.

Para garantir o uso adequado dos seus dados e atender solicitações e liberações de locks a classe TLockingService conta com os seguintes métodos:

- Init: cria o vetor dsVec com um ponteiro para um objeto Thrift do tipo DataServiceClient devidamente criado para cada DS do cluster.
- getDS: retorna o ponteiro DataServiceClient para comunicação com um DS especificado depois de obter os locks omp necessários.
- releaseDS: libera o ponteiro DataServiceClient especificado.
- SetLK: concede ou nega um bloqueio especificado por meio de uma estrutura aw_lock_request. Retorna uma estrutura aw_lock_request onde, para casos de

um bloqueio concedido, o RequestorID é o identificador do DS solicitante ou, para casos de um bloqueio negado, o RequestorID é o identificador do DS portador do bloqueio conflitante. Um bloqueio negado é enfileirado na Waiting_Llist a não ser que seja especificado para que isso não seja feito (via um parâmetro chamado “doNotEnqueue”).

- ReleaseLK: libera um bloqueio obtido anteriormente e valida todos os bloqueios enfileirados na Waiting_Llist concedendo bloqueios que eventualmente não sejam mais conflitantes.
- ExpireAllLK: libera todos os bloqueios de um FID especificado.

A.4 Data Service

Semelhante o Container Service, o Data Service também tem seu funcionamento baseado em uma classe TDataService e um serviço Thrift chamado DataService.

A.4.1 Classe TDataService

A classe TDataService é uma classe C++ que não é derivada de nenhuma outra, contendo variáveis que definem o características básicas e outras estruturas com dados sobre os objetos armazenados. São dados da classe TDataService:

- Cmap: uma lista com todos os objetos to tipo TContainer que são armazenados pelo DS. Essa lista é um objeto do tipo C++ std::map, com o identificador do *container* como chave para cada objeto TConatiner armazenado.
- containerList: um objeto Thrift de tipo CL (container list, veja item 4.3) que é uma cópia local da CL gerenciada pelo CS.
- vetores MPIcommVector: são listas com os comunicadores MPI_Comm usados por cada thread de processamento de mensagens para manipulação de cada cadeia de replicação de *container*. Essas listas são objetos C++ do tipo std::map e

a chave de acesso é o identificador do *container* master da cadeia de replicação (veja 4.3). O uso de um vetor por thread se fez necessário pois o MPI funciona melhor com um comunicador diferente por thread dedicada a um conjunto de mensagens.

- dsHostIDs: lista com um identificador único para cada host executando um DS. Esses identificadores são usados para detectar casos em que um DS coexiste com um cliente AwareFS em um mesmo servidor.

- newOwnerTxnList: lista de transações de mudança de owner sendo tratadas pelo DS. Cada mudança de owner gera uma transação que é registrada nessa lista por meio de um objeto Thrift de tipo msgNORQ contendo o identificador do *container*, o FID cujo owner deve mudar, o identificador do DS que será o novo owner e o timestamp indicando o momento da solicidação de mudança de owner. Essa lista é um objeto C++ do tipo std::deque.

- newCIDs: uma pilha com os identificadores dos novos *containers* criados pelo CS. A newCIDs é um objeto C++ do tipo std::stack.

- lockList: uma lista de locks com uma variável tipo omp_lock_t para cada registro aw_lock_request solicitado pelo DS. São esses locks que evitam a utilização de "*busy wait*" causando o bloqueio da thread enquanto ela espera que seja atendida a solicitação de bloqueio indicada pela estrutura aw_lock_request. Essa lista é da classe C++ std::unordered_map para um melhor desempenho.

- cv e mtx: Essas variáveis são respectivamente do tipo C++ std::condition_variable e std::mutex e são usadas em conjunto para fazer com que a thread que promove as replicações dos *containers* adormeça enquanto não houver solicitações a serem atendidas.

- LList_lock: uma variável do tipo omp_lock_t para viabilizar a inserção e remoção de locks na lockList pelas várias threads atendendo solicitações concorrentes de escrita e leitura de dados.

Além dos métodos para atenderem as solicitações de leitura e escrita, a classe TDataService possui métodos para tratamento de eventos de replicação de dados, para o tratamento de eventos da cadeia de replicação e de criação de *containers* além de métodos para mudança do papel de owner de um nodo e para interação com o LS. Os métodos do DS para atender solicitações RPC de leitura e escrita são:

- CheckFID: retorna um conjunto de bytes de um *chunk* indicado por seu FID. Se o FID indicado for um inode será retornado o inode completo ou um item da lista de FIDs do inode.
- CheckHDR: retorna o inode completo. Quando o Inode está invalidado no DS, essa solicitação sempre é redirecionada para o DS que é owner do Inode.
- getNewFID: armazena no AwareFS o buffer passado como parâmetro e retorna o FID que o identificará. Esse método é usado para append de dados em um arquivo e dispensa o uso de locks do AwareFS por ser um trecho novo do arquivo que ainda não são visíveis por outros processos. Ela recebe como parâmetro o tipo do FID a ser criado que pode ser 'c' para *chunks* de arquivos ou 'h' para inodes (cabeçalhos de arquivos).
- addChunkToFile: aidiciona ou atualiza um FID na lista de FIDs de um inode. Esse método é o reponsável por atualizar os metadados tornando o dado visível para uma leitura ou escrita subsequente.
- write: método de escrita de dados em um *chunk* com controle de consistência por meio de solicitações de bloqueio controladas pelo LS.
- read: método de leitura de dados em um *chunk* com controle de consistência por meio de solicitações de bloqueio controladas pelo LS.

Os métodos de chamadas RPC de controle do DS são:

- getDSOwner: retorna o identificador do DS que é o responsável pela cópia primária de um nodo

- InvalidateFID: marca um FID como invalido.

Para controlar a interação com o LS, o DS conta com os seguintes métodos:

- SetLK e ReleaseLK: Procede todo o controle de locks para viabilizar solicitações e liberações concorrentes de bloqueios para leitura e escrita no AwareFS e repassando a solicitação para o LS.
- expireLocks: libera todos os locks omp que existirem por mais tempo que o definido na configuração do AwareFS (parâmetro AWFS_LOCKDURATION) ou que sejam de um FID especificado.
- AcquireLK: método chamado por RPC quando um bloqueio pode ser concedido pelo LS depois que ele tenha liberado o bloqueio conflitante.

O tratamento de mensagens MPI é feito por threads concorrentes, processando mensagens para criação de *container* e sua cadeia de replicação, replicação de dados dentro da cadeia de replicação e criação de *checkpoints* pelo processo descrito em 4.8.3.1.

A.4.2 Serviço DataService

O serviço Thrift DataService (Figura 37) é do tipo TNonblockingServer, onde o servidor é criado passando como parâmetro um objeto de uma classe DataServiceFactory que tem como função principal a criação de um objeto DataServiceHandler para cada thread iniciada. Ambas as classes DataServiceFactory e DataServiceHandler possuem apenas dois dados, um objeto da classe TDataService e um objeto da classe TLockingService, os quais são singletons criados no construtor da classe DataServiceFactory e são passados como parâmetros na criação dos objetos DataServiceHandler. Cada conexão de cliente gera uma tarefa que é tratada por threads diferentes criadas e gerenciadas pelos objetos Thrift PosixThreadFactory e ThreadManager. O serviço Thrift do tipo TNonblockingServer é capaz de gerenciar muitas

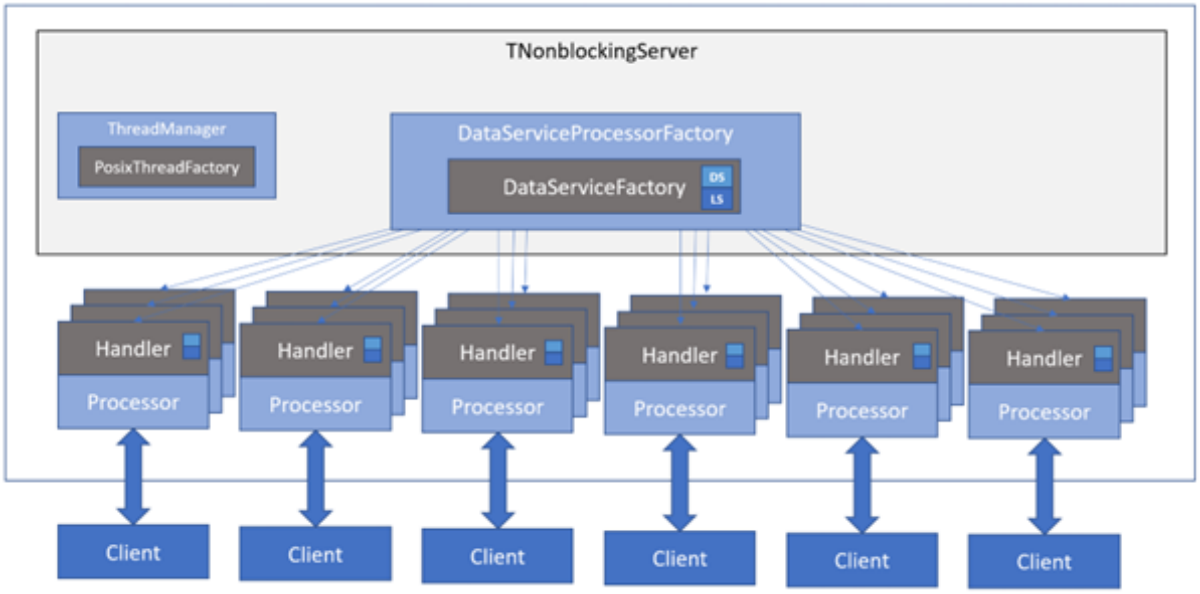

Figura 37 - Serviço Thrift DataService

conexões concorrentes com um número reduzido de threads. Para o caso do protótipo do AwareFS, cada DS usa um objeto TNonblockingServer com as conexões sendo distribuídas em seis threads para I/O.

Cada DataService handler responde às requisições equivalentes a cada uma das chamadas RPC a serem tratadas pelo objeto TDataService criado, ou seja, existe uma função CheckFID que chama o método CheckFID do DS (objeto TDataService do handler), uma função CheckHDR que chama o método CheckHDR do DS e assim por diante. De forma análoga existem funções SetLK, ReleaseLK, expireLocks e AcquireLK para chamar os métodos de RPC correspondentes do objeto TLockingService criado.

A.5 FuseClientAware – o cliente POSIX

O AwareFS, assim como tantos outros sistemas de arquivos, usou a biblioteca FUSE (FUSE, 2013) (file system in user space) na criação do seu cliente, buscando compatibilidade com o padrão POSIX (SPECIFICATIONS, 2008). É a compatibilidade com o padrão POSIX que faz do AwareFS um sistema de arquivos facilmente utilizável

com aplicações convencionais.

Mesmo com a utilização do FUSE, o desenvolvimento do FuseClientAware demandou mecanismos eficientes para a execução de múltiplas threads em paralelo e para garantir mecanismos para entrada e saída mais eficiente utilizando buffers.

Com o FUSE, o cliente de um sistema de arquivos pode abrir sessões concorrentes, aumentando o poder de resposta a múltiplas requisições. O AwareFS usa uma estratégia de execução de múltiplas threads para a capacidade sessões concorrentes do FUSE. Uma classe session_data foi criada para armazenar as estruturas Thrift para comunicação RPC com o MS, CS e DSs, possuindo também uma cópia local da CL, uma lista com um buffer para cada *chunk* sendo trabalhado e métodos para comunicação com os DSs de forma concorrente. Sempre que um novo PID (identificador de processo unix) é identificado, uma nova thread do FuseClientAware é criada e inicia uma instância da classe session_data.

Além das funções de read e write, o FuseClientAware contém as seguintes funções pra implementar sua interface POSIX:

- access: Definida mas não tem efeito pois não é algo aplicável para o AwareFS.
- chmod: Sem efeito. Essa função apenas registra no log que foi chamada mas não tem efeito pois o protótipo do AwareFS não implementa controle de autorização.
- chown: Sem efeito. Essa função apenas registra no log que foi chamada mas não tem efeito pois o protótipo do AwareFS não implementa controle de autorização.
- create: A função create recebe como parâmetro o nome do arquivo e o modo a ser usado. Para o AwareFS basta redirecionar uma chamada da função create para a função open, pois ambas têm o mesmo efeito.
- flush: Semelhante a função release, a função flush apenas persiste os dados de um arquivo aberto para escrita sem destruir estruturas de controle.

- fsync: Executa a função flush.
- fsyncdir: Definida mas não tem efeito pois não é algo aplicável para o AwareFS.
- getattr: Recebe como parâmetro um nome de arquivo e uma estrutura stat usada para armazenar uma série de metadados que descrevem o status do arquivo. Essa função obtem os metadados de um arquivo usando o método getFileID do objeto session_data e preenche os campos da estrutura stat recebida.
- getxattr: Definida mas não tem efeito pois não é algo aplicável para o AwareFS.
- link: Sem efeito. Essa função apenas registra no log que foi chamada mas não tem efeito pois o protótipo do AwareFS não implementa links.
- mkdir: Recebe como parâmetro o nome do diretório a ser criado e o modo de criação. Essa função cria uma entrada, apenas na base do MS, para um diretório.
- mknod: Recebe como parâmetro o nome do arquivo a ser criado e o modo de criação. Essa função cria uma entrada, apenas na base do MS, para um arquivo sem conteúdo ou para um diretório, dependendo do modo especificado.
- open: O objetivo dessa função é iniciar as estruturas de controle para o acesso a um arquivo. Ela recebe como parâmetro o nome do arquivo a ser aberto. O processo de abertura de um arquivo inicia usando os métodos getFileID e/ou setFileID do objeto session_data para criar, modificar ou ler qual o DS que pode ser usado para trabalhar com o inode de um orquivo. Uma vez definido qual o DS que pode ser usado, o inode é recuperado fazendo uma chamada CheckHDR para o DS. O DS pode ser um dos três DSs que possuem cópia e será preferencialmente o próprio host executando o FuseClientAware se este coexistir no mesmo host que um DS. A função open cria uma estrutura com o nome do arquivo, o identificador do DS a ser usado e o objeto Thrift inodeHdr com os metadados do arquivo. Essa estrutura armazenada em um vetor armazenado no objeto session_data para ser usada nas funções de leitura e escrita.

- readdir: Essa função recebe como parâmetro um nome de diretório e um buffer. O método readdir do MS é chamado para obter uma lista dos arquivos e subdiretórios do diretório especificado. O buffer recebido é então preenchido com os metadados do conteúdo do diretório.

- release: Chamada sempre que um arquivo é fechado, a função release recebe como parâmetro o nome de um arquivo, persiste os dados armazenados em *cache* e destrói estruturas usadas para sessões de leitura e escrita do arquivo no AwareFS.

- rename: Recebe como parâmetros o nome do arquivo a ser renomeado e o novo nome. Essa função invalida a entrada do MS com o nome original e cria uma nova entrada com o novo nome e o FID original.

- rmdir: A função rmdir recebe como parâmetro o nome de um diretório e procede sua remoção chamando o método unlink do MS.

- symlink: Sem efeito. Essa função apenas registra no log que foi chamada mas não tem efeito pois o protótipo do AwareFS não implementa links.

- truncate: Recebe como parâmetro o nome do arquivo a ser truncado e cria nova entrada, apenas na base do MS, para um arquivo sem conteúdo.

- unlink: A função unlink recebe como parâmetro o nome de um arquivo e procede sua deleção chamando o método unlink do MS.

A.5.1 Função de leitura no AwareFS

A função read recebe como parâmetros o nome do arquivo, a posição inicial da leitura, o número de bytes a serem lidos e o buffer para receber os bytes lidos. Inicialmente o arquivo é aberto com uma chamada a função open e é feita uma verificação para saber se já existe o dado a ser lido em um *cache* pré-armazenado em memória com o trecho desejado (indicado pela posição inicial e número de bytes desejados). Não havendo os dados em *cache*, o processo de leitura passa pelos seguintes passos:

- Se a leitura exceder o tamanho de um *chunk*, é feita uma divisão do trecho a ser lido em pedaços menores ou iguais ao tamanho de um *chunk*. Usando uma estratégia de “read-ahead”, toda leitura recupera no mínimo da posição inicial até o final do *chunk*, mantendo os bytes extras em *cache* para leituras subsequentes.

- Para cada *chunk* é identificado o DS a ser contactado para a leitura, que será preferencialmente o próprio host executando o FuseClientAware se este coexistir no mesmo host que um DS, ou do contrário, será o último DS que sabidamente foi dono desse *chunk*, essa informação é armazenada no campo lko (last known owner) dos metadados do arquivo.

- Os bytes lidos são armazenados no *cache* para leituras subsequentes.

A.5.2 Função de escrita no AwareFS

Dentro do FuseClientAware, a função write recebe como parâmetro o nome do arquivo a ser escrito, um buffer com os bytes a serem gravados, o número de bytes a serem gravados e a posição inicial da escrita. Assim como na função read, na função write inicialmente o arquivo é aberto com uma chamada a função open, depois caso o arquivo tenha tamanho igual a zero e só exista no MS, o arquivo é criado gravando-se o buffer recebido. Se o número de bytes a serem escritos for menor que o tamanho de um *chunk* dois caminhos podem ocorrer:

- Se for uma escrita sequencial, os bytes são armazenados em um buffer e só são persistidos no AwareFS quando o buffer alcançar o tamanho de um *chunk*.

- Se for uma escrita fora de sequência, os dados que já estão no buffer do arquivo são persistidos e os novos bytes são armazenados no buffer.

Para persistir os bytes recebidos no AwareFS, a função write divide a sequência de bytes em subdivisões de no máximo o tamanho de um *chunk*, depois identifica se é um append (i.e. os bytes serão adicionados em um *chunk* novo ao final do arquivo ou

são os primeiros bytes a serem armazenados no arquivo) ou se é uma reescrita (i.e. escrita no meio do arquivo) e para cada subdivisão toma um de dois caminhos:

- Para appends é determinado o DS a ser contactado para escritas e será preferencialmente o próprio host executando o FuseClientAware se este coexistir no mesmo host que um DS, ou do contrário, será arbitrariamente o último DS do cluster. Depois, se não existir o inode do arquivo, ele é criado chamando o método getNewFID do DS determinado passando o tipo 'h' como parâmetro. Com o inode criado, o método getNewFID do DS é chamada passando o tipo 'c' para que os bytes sejam persistidos e o método addChunkToFile do DS owner do inode é chamada para adicionar o FID do novo *chunk* à lista de FIDs do arquivo.
- Para reescrita o DS a ser contactado será o último DS que foi dono desse *chunk*, o lko, e o método write do DS é chamado passando o hostID do host executando o FuseClientAware, os bytes a serem escritos, o FID do inode do arquivo, o FID do *chunk* a ser sobrescrito, o número de bytes a serem persistidos e a posição para gravar os bytes no arquivo. Se a função write retornar um identificador diferente do DS utilizado para chamar o método write, significa que o dono do *chunk* foi alterado e o método write é chamado novamente a partir do DS dono do *chunk*.

A função write em casos de escrita de trechos intermediários de um arquivo vai sempre usar o DS dono do *chunk* em questão e, como explicado no item 5.7.4.4, é o DS que decide internamente se a propriedade sobre um *chunk* deve passar para outro DS dentro da cadeia de replicação.