PATRICK ANDJASUBU BUNGAMA

UM REPOSITÓRIO CHAVE-VALOR COM GARANTIA DE LOCALIDADE DE DADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Prof. Dra. Carmem Satie Hara

CURITIBA

2016

PATRICK ANDJASUBU BUNGAMA

UM REPOSITÓRIO CHAVE-VALOR COM GARANTIA DE LOCALIDADE DE DADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Prof. Dra. Carmem Satie Hara

CURITIBA

2016

B942r Bungama, Patrick Andjasubu
Um repositório chave-valor com garantia de localidade de dados / Patrick Andjasubu Bungama. – Curitiba, 2016..
76f. : il. [algumas color.] ; 30 cm.

Dissertação (mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2016.

Orientadora: Carmem Satie Hara.

1. Informática. 2. Banco de dados. I. Universidade Federal do Paraná. II. Hara, Carmem Satie. III. Título.

CDD: 004.74

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós Graduação em INFORMÁTICA
Código CAPES: 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **PATRICK ANDJASUBU BUNGAMA**, intitulada: "**Um Repositório Chave-Valor com Garantia de Localidade de Dados**", após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua aprovação.

Curitiba, 09 de Agosto de 2016.

Carmem Satie Hara
Prof CARMEM SATIE HARA
Presidente da Banca Examinadora (UFPR)

Prof LUIZ CARLOS PESSOA ALBINI
Avaliador Interno (UFPR)

Prof NADIA PUCHALSKI KOZIEVITCH
Avaliador Externo (UTFPR)

Flávio Rubens de Carvalho Sousa
Prof FLÁVIO RUBENS DE CARVALHO SOUSA
Avaliador Externo (UFC)

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

PATRICK ANDJASUBU BUNGAMA

UM REPOSITÓRIO CHAVE-VALOR COM GARANTIA DE LOCALIDADE DE DADOS

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comissão formada pelos professores:

Orientadora: Prof. Dra. Carmem Satie Hara
Departamento de Informática, UFPR

Prof. Dr. Luiz Carlos Albini
Departamento de Informática, UFPR

Profa. Dra. Nádia Kozievitch
Departamento de Informática, UTFPR

Prof. Dr. Flávio Sousa
Departamento de Informática, UFC

Curitiba, 09 de Agosto de 2016

Ao meu amigo, irmão, tio e pai, Anwar Kalombo.

(In memoriam)

AGRADECIMENTOS

Agradeço primeiramente a Deus Altíssimo, Criador dos ceus e da terra, por sua infinita fidelidade e bondade. Sem Ele, nada disso teria sido possível.

À minha família. Em especial à minha mãe Josine Claeys por sempre me incentivar nos momentos de dúvidas e tribulações. Não existem palavras que descrevam a incomensurável gratidão que sinto por você.

A honra vai a minha orientadora professora doutora Carmem Hara, a qual acreditou em mim para a realização deste trabalho e tem sido tão generosa e paciente em me transmitir a parte de seu grande conhecimento. Estou-lhe grato deveras.

Agradeço a minha companheira Katherine, pelo amor e cumplicidade. Obrigadao por estar ao meu lado, sempre.

Aos colegas de laboratório de banco de dados, e em especial ao meu amigo Wendel, que sempre que possível, me ajudaram e tornaram as coisas menos complicadas.

Aos meus Grandes Amigos, Steve Ataky e Mike Muya, por estarem sempre presentes com palvras de encorajamento.

Ao meu amigo Diego Souza(dgvncsz0f), pelas ajudas, dicas e principalmente pela pessoa que é. Você tem sido uma grande referência.

Agradeço aos professores Luiz Carlos Albini, Nádia Kozievitch e Flávio Sousa, por aceitarem de participar da banca de defesa deste mestrado.

Por fim, a todos aqueles que de uma maneira ou de outra contribuíram para que este percurso pudesse ser concluído.

"La persévérance est un talisman pour la vie."

Proverbe africain

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Grandes volumes de dados produzidos diariamente trouxeram desafios envolvendo a definição de formas eficientes de como extraí-los, armazená-los e acessá-los. Entretanto, soluções tradicionais de bancos de dados não se mostraram eficientes diante de tais desafios, principalmente no requisito de escalabilidade.

Uma possível abordagem para prover escalabilidade horizontal aos sistemas gerenciadores de banco de dados é a adoção de uma arquitetura estratificada, tendo como base um sistema de armazenamento distribuído com uma interface simples para o acesso a dados remotamente armazenados.

Esta dissertação apresenta o **ALOCS**, um repositório distribuído de dados que adota o modelo chave-valor, mas que permite a alocação de um conjunto de pares agrupados em uma única estrutura, cuja localidade é controlada pela aplicação usuária do sistema.

O controle de localidade permite que dados usualmente utilizados em conjunto possam ser alocados em um mesmo servidor, reduzindo a quantidade de comunicações entre servidores no processamento de suas consultas. Isto é essencial para prover escalabilidade e melhorar o desempenho de processamento das consultas em ambientes distribuídos.

Os estudos experimentais monstram a melhoria no tempo de resposta das consultas utilizando a solução proposta.

ABSTRACT

Large volumes of data produced every day brought new challenges involving the definition of efficient ways to extract, store and access them. However, traditional database solutions are not efficient to solve these challenges, especially with respect to the scalability requirement.

One approach to provide horizontal scalability to database management systems is the adoption of a layered architecture, based on a distributed storage system with a simple interface to access data remotely stored.

This dissertation presents **ALOCS**, a distributed storage repository of data which adopts the key-value model, and which allows the allocation of a set of pairs grouped into a single structure whose location is controlled by the user application of the system.

This control allows data commonly used together to be allocated on the same server, reducing the amount of communications between servers for query processing. This is essential to provide scalability and improve the processing of query execution in distributed environments.

Experimental studies shows that **ALOCS** improves query response times by reducing the amount of remote data accesses.

CAPÍTULO 1

INTRODUÇÃO

1.1 Contextualização

Nos últimos anos, a Internet passou a ser usada como uma das principais vias para a troca de dados entre instituições e entre pessoas. Assistiu-se ao seu aperfeiçoamento acelerado e incessante junto com a explosão de sistemas computacionais, tais softwares corporativos, aplicações web, dispositivos móveis, entre outros. Uma consequência é a geração de volumes gigantescos de dados [26].

O mundo tem, hoje em dia, cerca de cinco bilhões de telefones celulares, milhões de sensores espalhados para capturar dados de quase todos os aspectos da vida e também inúmeras redes sociais [76]. O exemplo mais ilustrativo das redes sociais é o do Facebook ([1]), que por si só concentra mais de 900 milhões de usuários ativos que compartilham cerca de 30 bilhões de arquivos por mês com conteúdos diversos. No mesmo intervalo de tempo, mais de 20 bilhões de buscas são executadas na Internet. Todos estes dados se juntam à já significativa quantidade de informação gerada pelos meios tradicionais. A tendência é que este universo digital cresça ainda mais (de 130 exabytes para 40.000 exabytes, ou 40 trilhões de gigabytes) [26]. De 2010 até 2020, o universo digital irá dobrar a cada dois anos [26].

Esta avalanche de dados trouxe novos desafios, como a definição de formas eficientes para coletar, armazenar, acessar e extrair estes dados [85]. Contudo, soluções tradicionais de banco de dados não têm se mostrado eficientes diante destes desafios, principalmente por não apresentarem escalabilidade aceitável [2]. A escalabilidade vertical, obtida por meio da adição de recursos a um único servidor, é limitada e apresenta um alto custo. Já a escalabilidade horizontal, que adiciona mais servidores ao sistema, é mais atrativa, mas apresenta desafios tecnológicos para dar suporte a um modelo de armazenamento de dados

[1] http://www.facebook.com

distribuídos [87, 76]. As novas soluções precisam ser não apenas escaláveis mas também elásticas para garantir uma resposta rápida ao crescimento da demanda de capacidade de armazenamento [22].

Existem diversos sistemas gerenciadores de bancos de dados (SGBDs) baseados em uma arquitetura estratificada, na qual o módulo de armazenamento possui uma interface bem definida com o módulo de processamento de consultas. Isso permite que diferentes estruturas de armazenamento possam ser utilizadas, mantendo a mesma linguagem de consulta para o usuário e para as aplicações. Tal arquitetura é adotada por SGBDs relacionais (MySQL[2], MariaDB[3]), XML (Zorba[4]) e mais recentemente, NoSQL (MongoDB[5]).

Desta forma, uma possível abordagem para obter escalabilidade horizontal é a integração de um módulo de armazenamento escalável para estes sistemas. Segundo Cattell [13], existem quatro características comuns em repositórios de dados escaláveis: (i) são baseados em uma interface simples;(ii) conferem a habilidade de escalar horizontalmente um sistema sobre muitos servidores; (iii) dispõem de índices distribuídos e; (iv) permitem um ajuste dinâmico da distribuição da carga de trabalho. Todas estas características estão associadas à natureza *shared-nothing* de redes par-a-par (P2P). Tendo em vista que as tabelas de dispersão distribuídas (*DHT - Distributed Hash Table*) foram propostas neste contexto para oferecer uma interface de uso geral para a indexação e armazenamento distribuído de dados, alguns sistemas adotaram DHTs para prover escalabilidade aos SGBDs [19, 8, 62].

Uma das dificuldades relatadas pelos sistemas que adotaram uma DHT como módulo de armazenamento é o alto custo de comunicação para o processamento de consultas. Isso se deve ao fato das DHTs adotarem o modelo chave-valor e oferecerem uma interface para acesso de valores individuais a partir da chave (operações *get-put-rem*) [80]. Em SGBDs, as consultas envolvem um conjunto de valores relacionados. Como as DHTs em geral não oferecem um controle sobre a localidade, a recuperação deste conjunto de dados pode incorrer em uma comunicação entre servidores para a recuperação de cada elemento

[2]http://www.mysql.com
[3]http://mariadb.com
[4]www.zorba.io
[5]http://www.mongodb.com

individualmente.

Assim, uma possível alternativa é a utilização de um sistema de arquivos distribuídos (SAD) no módulo de armazenamento do SGBD. Nos SADs, o modelo usado para armazenar dados é o arquivo. Os SADs permitem que aplicações armazenem e acessem arquivos remotos a partir de qualquer computador em uma rede como se fossem locais, sendo responsáveis por organizá-los, armazená-los, recuperá-los, compartilhá-los e protegê-los [61]. Arquivos de metadados são utilizados para armazenar a localização dos arquivos. O acesso aos metadados, quando centralizado, gera de 50% a 80% de todo tráfego de rede [54]. Apesar do tamanho dos metadados ser geralmente pequeno em comparação com a capacidade de armazenamento do sistema, ele pode tornar-se um gargalo potencial. Evitar este tipo de gargalo é fundamental para que um sistema de armazenamento atinja alto desempenho e escalabilidade. Alguns SADs, o Ceph [81] e o PVFS [64] entre outros, permitem que aplicações especifiquem em quais servidores armazenar arquivos e assim garantir a o controle de localidade dos dados, isto é, decidir como alocar dados em nodos em um ambiente distribuído para otimizar a performance do sistema [56].

1.2 Motivação e Objetivos

Em sistemas que usam a DHT, a alocação de dados é feita de maneira aleatória pela função de espalhamento. Isso faz com que a aproximação de dados das aplicações não seja considerada. A função de espalhamento se encarrega de distribuir e localizar os dados [87]. Portanto, o controle de localidade de dados pelas aplicações é nulo em soluções baseadas em DHT, embora estas soluções garantem a escalabilidade.

Diferentemente da DHT, alguns SADs permitem que aplicações especifiquem em quais nós armazenar arquivos (dados). Esta vantagem pode ser usada para implementar sistemas de armazenamento distribuído com controle de localidade de dados. O controle de localidade permite que dados sejam colocados perto de aplicações clientes, o que pode reduzir o número de acessos remotos, reduzir significativamente a latência de operações, e evitar o congestionamento da rede. Implementar tais sistemas deve levar em consideração o papel importante desempenhado pelos arquivos de metadados na localização eficiente

de dados em nós de armazenamento.

Como mencionado anteriormente, embora o tamanho de metadados seja geralmente pequeno em comparação a capacidade de armazenamento total de tal sistema, um gargalo potencial é o acesso a eles [54]. Evitar gargalos é fundamental para que um sistema de armazenamento distribuído atinja alto desempenho e escalabilidade. O desenvolvimento de uma abordagem de armazenamento distribuído baseada em um SAD com garantia de controle de localidade de dados usando um sistema de metadados que não seja um gargalo é o tema desta dissertação.

Esta dissertação apresenta o ALOCS, um repositório distribuído de dados que adota o modelo chave-valor, mas que permite a alocação de um conjunto de pares agrupados em uma única estrutura, denominada *bucket*, cuja localidade é tratada de maneira controlada e orientada pela aplicação usuária do sistema. Assim, embora a interface para a aplicação seja similar aos repositórios NoSQL (baseada nas operações get-put-rem), a unidade de comunicação de dados entre servidores é o *bucket*. Desta forma, somente o acesso ao primeiro par chave-valor armazenado em um *bucket* remoto requer uma comunicação entre servidores. Como o *bucket* é mantido localmente em Cache, acessos subsequentes a chaves armazenadas no mesmo *bucket* não requerem nova comunicação, caso o item ainda esteja no Cache. Esta funcionalidade de controle de localidade, aliada a um modelo simples chave-valor, permite que o ALOCS possa ser usado como *backend* de armazenamento para SGBDs, dando suporte a um modelo de *clusterização* distribuído.

No ALOCS, o armazenamento físico de dados é feito em um SAD. Como os SADs apóiam-se em uma estrutura de metadados para associar o dado ao servidor no qual ele está armazenado, o ALOCS permite a replicação destas informações que são os metadados, sendo que o número de réplicas pode ser definido de acordo com o volume de leituras e escritas da aplicação. Ou seja, aplicações com muitas leituras podem possuir diversas réplicas dos metadados para evitar que ele torne-se um gargalo. No entanto, para aplicações com muitas escritas a existência de muitas réplicas torna sua atualização um gargalo e portanto uma arquitetura com poucos servidores de metadados é mais apropriado. Das características comuns de repositórios de dados escaláveis, o ALOCS possui a interface

simples, a escalabilidade horizontal e a indexação distribuída. A distribuição da carga de armazenamento pode ser atinjida por meio da funcionalidade de controle de localidade de dados.

Além de apresentar uma abordagem nova de armazenamento de dados, a idéia principal desta dissertação é avaliar os custos acarretados pelo controle de localidade de dados e destacar os benefícios do agrupamento de informações correlatas em um mesmo servidor, e da proximidade dos dados com a aplicação consumidora. Espera-se também que os conceitos discutidos sirvam de embasamento para os trabalhos futuros relacionados ao tema tratado.

1.3 Organização do trabalho

O restante deste trabalho está organizado da seguinte maneira: O capítulo 2 apresenta conceitos sobre os sistemas de arquivos distribuídos (SADs), soluções baseadas em tabelas de espalhamento distribuído (DHT) (bancos de dados NoSQL) e a localidade de dados. No capítulo 3, são apresentados os trabalhados relacionados ao tema explorado nesta dissertação. São descritos alguns SADs e bancos de dados NoSQL correlatos. O capítulo 4 apresentada o SAD Ceph e o *Zookeeper*, duas ferramentas usadas na implementação da solução proposta nesta dissertação. A descrição do projeto (arquitetura, componentes do sistema, interfaces e operações de manipulação de dados) é apresentada no capítulo 5. Os resultados experimentais para a validação do sistema proposto são descritos no capítulo 6 e, finalmente, o capítulo 7 delineia as conclusões e as considerações finais.

CAPÍTULO 2

ARMAZENAMENTO DISTRIBUÍDO DE DADOS

O objetivo deste capítulo é servir de embasamento teórico para o restante do trabalho. Ele está dividido em 4 seções principais. Inicialmente são apresentados os conceitos básicos de sistemas de armazenamento de dados. Em seguida, são descritos os sistemas de arquivos distribuídos e os sistemas NoSQL. E por fim, são discutidos conceitos relacionados ao controle de localidade de dados.

2.1 Armazenamento de dados

Um sistema de armazenamento pode ser vista como uma pilha, onde a camada superior mascara o funcionamento da inferior. A camada mais baixo descreve a estrutura de armazenamento físico. Ela utiliza um modelo de dado físico. No nível acima da camada física, encontra-se o suporte lógico, que abstrai o funcionamento da camada física. O nível externo (visão externa) é o nível mais abstrato dos dados.

O mapeamento é o mecanismo que promove a integração de dados entre os modelos lógico e físico, permitindo o uso do equivalente de um modelo por uma camada subjacente a camada responsável do modelo. O mapeamento expressa a correspondência entre as ações do domínio de um modelo para as ações equivalentes no outro modelo. Na transição entre os modelos de dados, é necessário preservar as informações, operadores e relações. Isto requer que os modelos sejam repesentados de forma equivalente pelo mecanismo de mapeamento. Isto ocorre quando cada esquema em um modelo pode ser colocado em correspondência um para um com o esquema equivalente em outro modelo, concomitante ao fornecimento de conformidade entre os operadores de manipulação de dados definidos em cada modelo.

No caso de um disco rígido, o mapeamento é executado pela controladora de interface, que oferece às camadas superiores uma visão linear do disco. Sobre as camadas física

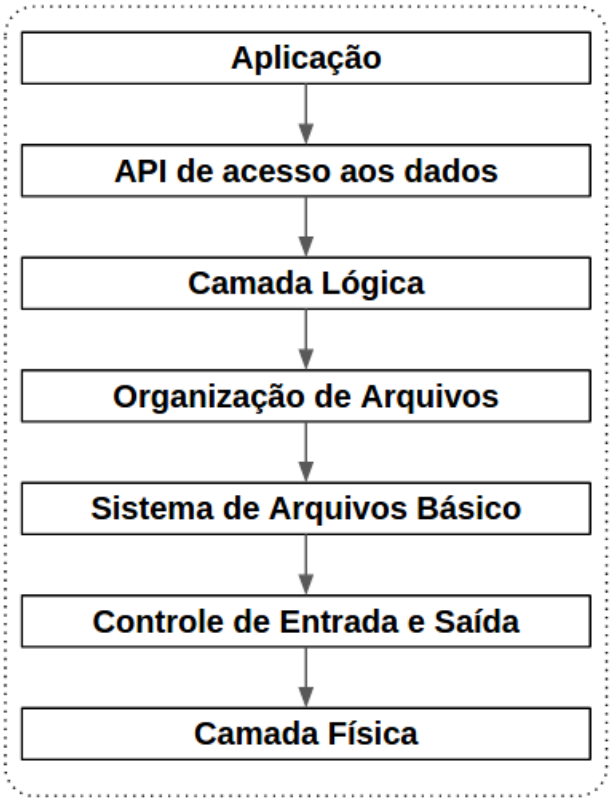

Figura 2.1: Camadas de *Software* e *Hardware* envolvidas no armazenamento de dados

e lógica podem existir outras camadas como, por exemplo, um sistema de arquivos ou um sistema de redundância RAID, adicionando mais funcionalidades ao armazenamento (Figura 2.1)[70]. No nível mais próximo da aplicação, geralmente existe algum mecanismo responsável por controlar a distribuição de dados e oferecer à aplicação meios de acesso de alto nível. Este mecanismo pode ser o sistema de arquivos distribuídos(SAD) ou a tabela de espalhamento distribuída (DHT), usada por grande parte dos bancos de dados NoSQL.

Nas seções seguintes serão abordados os principais conceitos relacionados ao sistemas de arquivos distribuídos e sistemas NoSQL.

2.2 Sistemas de Arquivos Distribuídos - SADs

Sistemas de arquivos distribuídos (SADs) são sistemas que provem o armazenamento permanente de arquivos de modo distribuído [46]. Isto é, o armazenamento de arquivos

é realizado por um ou mais dispositivos conectados entre si por meio de uma rede de comunicação.

Um SAD permite que programas armazenem e acessem arquivos remotos como se fossem locais. Isso possibilita que usuários acessem arquivos a partir de qualquer computador em uma rede. O SAD cria uma visão hierárquica e unificada destes arquivos, criando uma federação de recursos de armazenamento dispersos na rede. Os SADs tornaram-se a principal solução usada por conjunto de computadores *(Clusters)* para garantir o armazenamento permanente de dados em ambiente distribuído [20]. Os SADs dão suporte as operações de criação, escrita, leitura e exclusão do mesmo modo que os sistemas de arquivos tradicionais ou centralizados. Três componentes principais precisam ser definidos para entender melhor a estrutura de um SAD. São eles:

- **Serviço**: Aplicação *(software)* executada em um ou mais nós do ambiente distribuído;
- **Servidor**: Nó (nodo) do ambiente distribuído que executa algum serviço (aplicação);
- **Cliente**: Usuário que interage com o sistema a fim de obter recursos providos pelo serviço.

Por definição, um SAD deve ser capaz de oferecer, além de alguns recursos presentes nos sistemas de arquivos tradicionais (ou centralizados), outros recursos tais como: transparência, tolerância a falhas e escalabilidade. Estes conceitos são apresentados na subseção a seguir.

2.2.1 Caracteríscas de SADs

As principais características dos SADs são a transparência, a tolerância a falhas, a escalabilidade, o desempenho, o controle de concorrência, e a segurança. Tais atributos são descritos a seguir.

2.2.1.1 Transparência

A transparência permite que a complexidade do sistema seja escondida dos usuários. A solução distribuída apresenta-se como um único sistema para o usuário ou aplicação [61]. Considerando os SADs, ela pode ser dividida em:

- **Transparência de acesso**: esconde como um recurso é acessado. Não é necessário fornecer a localização dos recursos, ou seja, as aplicações devem executar os processos de leitura e escrita de arquivos remotos da mesma maneira que operam sobre arquivos locais. O usuário não deve perceber se o recurso acessado é local ou remoto.

- **Transparência de localização**: esconde a localização física de um recurso. A aplicação cliente deve ver um espaço de nomes de arquivos uniforme, sem a necessidade de fornecer a localização física dos arquivos para encontrá-los.

- **Transparência de replicação**: esconde a existência de cópias de um recurso. Existem várias cópias dos mesmos arquivos armazenados em locais diferentes para garantir a disponibilidade e a tolerância a falhas. A aplicação cliente deverá visualizar apenas uma cópia do mesmo, não necessitando saber a quantidade replicada e o local.

A transparência é altamente desejável em sistemas distribuídos, mas nem sempre é possível alcançá-la ou, em determinadas situações, não convém ocultá-la. Pode-se destacar uma situação em que seja mais conveniente o usuário tomar uma decisão sobre alguma falha do que o sistema tentar resolver por si só. Um exemplo ilustrativo disso é quando um serviço, por repetidas vezes, tenta estabelecer uma comunicação com um servidor remoto, neste caso o melhor é informar ao usuário sobre a falha e que ele tente mais tarde [75].

2.2.1.2 Tolerância a falha

Um sistema tolerante a falhas não deve ser interrompido em caso de falhas transitórias ou parciais. As falhas consideradas são as de rede e nós que podem afetar a disponibi-

lidade de dados ou serviços, integridade e consistência de dados. Considerando que as falhas são inevitáveis, a maior preocupação concentra-se em detectá-las e como tratá-las. Detectar falhas não é um processo exato. Nesse sentido, atitudes como mascaramento de falhas, recuperação após falhas e redundância podem ser aplicadas com o intuito de apoiar o tratamento das falhas. A tolerância a falhas também compreende um conjunto de medidas que procuram antecipar e contornar possíveis problemas. Geralmente, esses problemas acontecem quando há uma sobrecarga do uso dos recursos de software e hardware do sistema, resultando em mensagens perdidas, mensagens corrompidas, travamento de sistemas e dados corrompidos [75]. Das falhas, podem se destacar os seguintes problemas:

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

Em alguns casos, a saída mais adequada para tratar as falhas consiste com interação o usuário e adiar as ações de um processo enquanto o serviço responsável por ele não estiver respondendo.

2.2.1.3 Escalabilidade

A escalabilidade é a capacidade de lidar com quantidade crescente de trabalho, tendo a possibilidade de ser estendido para acomodar este crescimento [34]. Normalmente, trata-se da adição de dezenas de milhares de nós ao sistema existente. Existem dois tipos de escalabilidade [76]:

- **Escalabilidade horizontal** (*scale-out*): consiste em adicionar mais nós em um sistema para aumentar o seu poder computacional ou até mesmo sua capacidade de armazenamento. Neste tipo de ambiente o modelo seguido é o "dividir para conquistar". A cada nó é atribuído apenas um subconjunto do problema global.

- **Escalabilidade vertical** (*scale-up*): nesta abordagem, os recursos como CPU, memória e espaço de armazenamento são adicionados a alguns nós. Isto permite que serviços em execução tenham recursos para consumir.

Fornecer condições para agregar mais componentes conforme a demanda aumenta, maximizar a eficiência das interações entre os recursos a fim de evitar perda no desempenho e evitar projetos de sistemas com um número limitado de usuários ou recursos agregados são os grandes desafios ligados à escalabilidade. Diante disso, algumas situações que podem acontecer são:

- **Sobrecarga na rede**: se o SAD foi planejado para alta escalabilidade através de simples replicação dos arquivos pelos nós responsáveis em armazená-los, a latência aumentaria devido às buscas pelo arquivo em cada servidor da rede, pois não se sabe o local exato deste arquivo. Nesse caso, um servidor específico poderá ser utilizado para a resolução dos caminhos, agindo como centralizador, mas falhas ou sobrecarga poderão ocorrer sobre ele, comprometendo o sistema.

- **Incompatibilidade de *hardware* e sistema operacional**: se não definidas as interfaces do serviço de maneira que o software cliente e servidor possam ser implantadas por diferentes plataformas e computadores, o SAD poderá não ser compatível com as variações de hardware e as mudanças do sistemas operacional.

- **Consistência dos arquivos**: o acesso concorrente aos arquivos distribuídos nos quais há requisições de leitura e escrita pode ocasionar alguns problemas de consistência nos dados. Os sistemas de arquivos tradicionais oferecem a semântica de atualização de cópia única *(one-copy)* a qual, independente do acesso concorrente de diversas cópias replicadas, faz com que o retorno para o usuário seja da existência de apenas um arquivo.

2.2.1.4 Replicação de arquivos

Técnicas simples ou recursos mais avançados, que envolvem réplicas mantidas por gerenciadores de réplicas podem garantir a replicação. Neste último caso, a replicação pode ser passiva, quando alterações realizadas no *backup* primário são atualizadas em *backups* escravos, ou ativa, quando o grupo de dispositivos responsável das réplicas desempenham a mesma função [17].

A consistência de arquivos é um problema associado em manter várias cópias distribuídas. Operações de leitura não oferecem muitos problemas, mas a escrita precisa de maior atenção, pois após a atualização de algum arquivo, ele necessariamente deve ser replicado antes que um próximo evento exija a sua utilização. Com isso, a sincronização das réplicas precisam ser gerenciadas através de métodos que garantam consistência dos arquivos e um consumo reduzido da banda [75].

A eficiência do método de sincronização varia entre tornar as cópias consistentes ou economizar na comunicação e transferência dos arquivos. Isso porque ao elevar a prioridade para consistência dos dados, conferências de validação são executadas em pouco espaço de tempo, aumentando o consumo da rede e tornando o processo pouco eficiente e muito custoso para o sistema. Por outro lado, um tempo maior na conferência das versões dos arquivos em benefício ao consumo da rede pode ocasionar a defasagem dos dados.

2.2.1.5 Controle de concorrência

Diferentemente da leitura de arquivos, a operação de escrita necessita ser controlada. O controle de concorrência gerencia todas as ações de leitura e escrita de um arquivo ou

grupo deles. Este controle permite que todas as alterações feitas por um único cliente não influenciem as operações de outros clientes que acessam o mesmo arquivo naquele exato momento. Existem técnicas e algoritmos responsáveis em administrar várias requisições de escritas sobre o mesmo arquivo. A implantação delas varia de acordo com a aplicação, mas a maior parte dos serviços de arquivos atuais seguem através de travas *(locks)* em nível de arquivo ou em nível de registro [17].

Com esse bloqueio é possível serializar as operações sobre o arquivo através de listas de ações a serem executadas. O resultado obtido da serialização é o mesmo que se fossem executadas simultaneamente. Porém, o uso desta técnica pode criar o *deadlock*. Este ocorre quando em um conjunto de processos um ou mais aguardam o desbloqueio de um determinado recurso que, por sua vez, aguarda o desbloqueio de outro recurso em uso pelo primeiro processo [43].

2.2.2 Arquitetura de SADs

Existem diferentes arquiteturas de SADs [61]. As principais são:

- **Arquitetura Cliente-Servidor (centralizada)**, na qual tarefas e cargas de trabalho são distribuídas entre servidores e os clientes. Clientes solicitam serviços oferecidos por servidores. Exemplo: *Network File System* (NFS).

- **Arquitetura baseada em *Cluster***, na qual se encontra um servidor mestre, juntamente com vários servidores que, geralmente, compartilham recursos como memória. Exemplo: *Google File System* (GFS)[27].

- **Arquitetura simétrica (descentralizada)**: mais conhecida como peer-to-peer (P2P), onde cada um dos nós da rede funciona tanto como cliente quanto como servidor. Isso permite compartilhamentos de serviços e dados sem a necessidade de um servidor central [5].

- **Arquitetura assimétrica**, que considera a existência de um ou mais gerenciadores de metadados dedicados que mantêm o sistema de arquivos e suas estruturas de disco associados. Exemplo: Lustre [21].

- **Arquitetura Paralela**, onde blocos de dados são armazenados, através de vários servidores de armazenamento que dão suporte a escrita e leitura concorrentes. Exemplo: *Parallel Virtual File System* (PVFS) [64].

Grande parte dos SADs utilizam um sistema de metadados que coleta e armazena informações de proveniência de dados usando formatos de armazenamento otimizados, registros de índice de proveniência desses dados, e dá suporte às linguagens de consulta para obter a proveniência de dados [78]

2.2.3 Metadados

Como mencionado na subseção anterior, a maioria dos SADs guardam informações que servem para identificar os dados e manter outros atributos do arquivo. Este tipo de informação é chamado de metadado. Os metadados são "dados que descrevem os dados"[45]. Eles podem conter qualquer tipo de informação considerada importante pelo sistema de arquivos, tais como:

- **Método para recuperar os dados**: pode ser desde um bloco inicial e um tamanho, até estruturas complexas descrevendo os blocos de dados utilizados;
- **Tamanho do arquivo**: é preciso armazenar o tamanho do arquivo para saber a posição em que os dados serão anexados quando for necessário inserir dados no final do arquivo;
- **O tipo do arquivo**: é necessário para os sistemas de arquivos que suportam diferentes tipos de arquivos, com ligações simbólicas;
- **Propriedades adicionais**: são utilizadas para gerenciar permissões, propriedades dos dados, versões, comentários, entre outras;
- **Dados estatísticos**: alguns exemplos são a data de criação, de último acesso e frequência de acesso. Isto pode ser utilizado para otimizar o acesso aos dados, colocando os que são requisitados com mais frequência em regiões de acesso mais rápido. Também podem ser utilizados para controlar cópias de segurança (backup).

Embora tenha uma grande variedade de atributos de metadados, nem todos têm o mesmo impacto no acesso aos dados. Alguns servem de meio de obtenção de maiores informações sobre os arquivos, como data de acesso e criação. Por outro lado, outros atributos são cruciais para o acesso ao conteúdo, como a localização e o tamanho. Devido a esta forte associação entre metadados e dados, é importante otimizar o acesso e a localização destas informações para reduzir o tempo necessário para obter os dados de um arquivo. Diferentes técnicas podem ser usadas dependendo do tipo de sistema de proposto.

Tradicionalmente, metadados e dados são gerenciados pelo mesmo sistema de arquivos, nos mesmos nós, e armazenados nos mesmos dispositivos [52]. Por motivo de eficiência, metadados são geralmente armazenados fisicamente próximos aos dados que eles descrevem [49].

Uma abordagem adotada por alguns SADs é a distribuição de gerenciamento de metadados. Esta abordagem elimina a ideia de centralização. As funções de gerenciamento de dados e metadados são decompostas, e os metadados são armazenados separadamente em diferentes nós distantes de dados do usuário [73]. Os nós de metadados formam um *Cluster* de servidores de metadados (MDSs). As estratégias de distribuição de metadados usadas são a DHT, DFS ou sistemas específicos. No capítulo 4 é apresentado o *Zookeeper*, que é um sistema de coordenação de processos, usado nesta dissertação.

Clusters de metadados podem também enfrentar problemas ligados à escalabilidade, tolerância a falhas e disponibilidade de metadados. Mecanismos de controle de metadados estudados na atualidade buscam soluções que resolvam estes problemas para garantir um bom desempenho de sistema de metadados.

2.3 Sistemas NoSQL

Not Only SQL(NoSQL) é um conceito que engloba as bases de dados que não seguem os princípios das bases de dados relacionais. São diferentes sistemas de armazenamento que vieram para suprir necessidades nas quais os bancos de dados relacionais eram ineficazes.

Dentre as principais características apresentadas pelos bancos de dados NoSQL podem

ser destacadas as seguintes [33]:

- **Não existência de um esquema** pré definido (*schema free*): Os dados podem possuir qualquer número de atributos e de qualquer tipo. A falta de um esquema global facilita o desenvolvimento de um banco de alta disponibilidade, pois não é necessário interromper o funcionamento do banco para efetuar alterações no esquema.

- **Interface de programação de aplicações** (API) de buscas simples: diferente dos bancos relacionais, a API encoraja buscas simples, sem necessidade de grandes uniões de tabelas.

- **Escalabilidade** proporcionada pelos nós da rede, possivél através da elasticidade, que caracteriza como um *cluster* reage quando novos nós são adicionados ou removidos sob carga e do *sharding*, que é um método para distribuir dados em várias máquinas [42].

- **Consistência eventual**: diferente dos bancos de dados relacionais, que utilizam as propriedades ACID, os bancos NoSQL promovem uma "consistência eventual". Esta característica tem embasamento no teorema CAP *(Consistency, Availability e Partition tolerance)* [1].

- Alta disponibilidade: Isto é alcançado através da identificação de falhas e de recuperação utilizando cópias de nós espelhos.

Os bancos de dados NoSQL tem sido amplamente adotados no desenvolvimento de aplicações Web colaborativas, as quais, na maioria das vezes, necessitam de uma tecnologia que ofereça suporte ao gerenciamento e escalabilidade de grandes volumes de dados, de maneira simples e eficiente [47]. Os conceitos de BigData [48] e também o *Cloud Computing* [24] os tornaram ainda mais populares. As plataformas de *clouds*, tais o Amazon's EC2 [1], OpenStack [2] e Eucalyptus [53] que fornecem a infraetrutura como serviço são exemplos mais ilustrativos de plataformas que usam os bancos de dados NoSQL.

[1] https://aws.amazon.com/pt/ec2/
[2] http://openstack.org

2.3.1 Taxonomias de banco de dados NoSQL

Existem diversos tipos de banco de dados NoSQL, e a taxonomia é feita usando os modelos de dados que eles suportam. Os sistemas NoSQL são baseados em DHT, por isso não oferecem mecanismos de controle de localidade de dados. Os principais tipos de bancos de dados NoSQL são detalhados a seguir.

2.3.1.1 Banco de dados Chave-valor

O modelo Chave-valor é o mais simples de todos os modelos. Ele manipula os pares chave-valor, ou tabelas *hash*, cujo acesso é feito exclusivamente pela chave. Trata-se de mecanismo na memória como *Redis* [11] ou sistemas distribuídos inspirado como o *Dynamo* [25] e o *Riak* [33]. Estes sistemas oferecem recursos simplificados, muitas vezes menos riqueza funcional, em termos de consulta por exemplo, todavia excelente desempenho graças ao seu modelo de acesso simples. Um sistema de pares de chave-valor é relativamente fácil de se implementar: o padrão de acesso pela chave não requer nenhum mecanismo de consulta complexa, e este ponto de acesso único permite melhorar o desempenho.

2.3.1.2 Banco de dados Orientado a Documentos

Este modelo armazena coleções e documentos. Ele permite representar um documento estruturado que contém tipos de dados simples, mas também lista, em forma hierárquica. Trata-se de um modelo ideal para a representação de dados estruturados ou semi-estruturados. O seu formato de armazenamento nativo é o *JavaScript Object Notation* (JSON). Este é o caso do MongoDB [15] e do CouchDB [4].

Os sistemas NoSQL orientados a documentos possuem vantagens por oferecer flexibilidade, disponibilidade, linguagem de consulta simples e performance. Porém, a principal desvantagem é a questão de consistência de dados. A Figura 2.2 mostra como é a estrutura de documentos usados neste tipo de banco de dados.

```
{
    "Subject": "I like Plankton",
    "Author": "Rusty",
    "PostedDate": "5/23/2006",
    "Tags": ["plankton", "baseball", "decisions"],
    "Body": "I decided today that I don't like baseball. I like
       plankton."
}
```

Figura 2.2: Exemplo de documento de CouchDB

2.3.1.3 Banco de dados Orientado a grafos

Diferentemente de outros tipos de bancos de dados NoSQL, esse está diretamente relacionado a um modelo de dados estabelecido, o modelo de grafos (Figura 2.3). Este modelo surgiu como uma alternativa para a normalização dos SGBDs relacionais. Os dados e/ou o esquema de dados são representados como grafos dirigidos, ou como estruturas que generalizem a noção de grafos [6]. Operações sobre os dados são transformações sobre o grafo, e fazem uso de conceitos de grafos, como caminhos, vizinhos e sub-grafos. Esse modelo também dá suporte ao uso de restrições sobre os dados, como restrições de identidade e de integridade referencial, por exemplo.

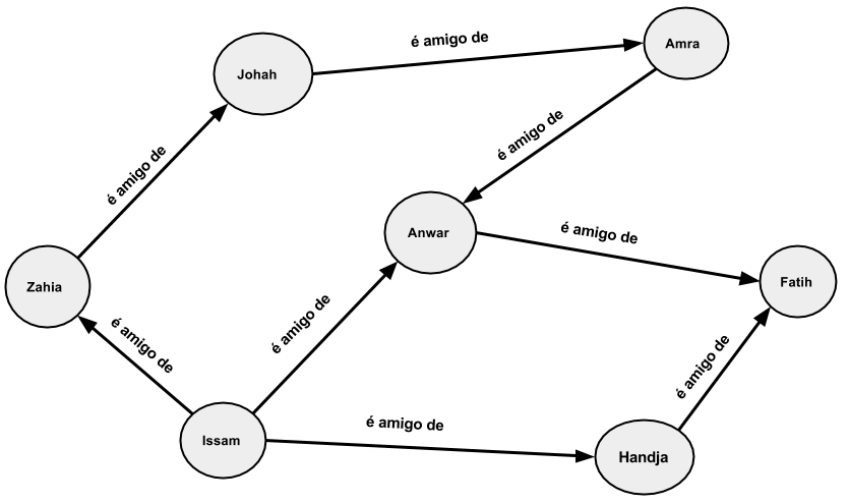

Figura 2.3: Exemplo de grafo [3]

O modelo de grafos é mais interessante que outros quando "informações sobre a interconectividade ou a topologia dos dados é mais importante, ou tão importante quanto, os dados propriamente ditos"[6]. Exemplos de bancos de dados nessa categoria são o Neo4j [51] e InfoGrid [30].

2.3.1.4 Banco de dados Orientado a colunas

Os SGBDs relacionais normalmente guardam os registros das tabelas contiguamente no disco. Caso se queira guardar, por exemplo, atributos como id, nome e endereço de usuários em um sistema de cadastro, os registros seriam: *Id1, Nome1, Endereço1; Id2, Nome2, Endereço2* . Essa estrutura torna a escrita muito rápida, pois todos os dados de um registro são colocados no disco com uma única escrita no banco. Essa estrutura também é eficiente caso se queira ler registros inteiros. Mas para situações onde se quer ler algumas poucas colunas de muitos registros, essa estrutura é pouco eficiente, pois muitos blocos do disco terão que ser lidos. Para esses casos onde se quer otimizar a leitura de dados estruturados, estes bancos de dados são mais interessantes, pois eles guardam os dados contiguamente por coluna [74].

O exemplo anterior em um banco de dados dessa categoria ficaria: *Id1, Id2; Nome1, Nome2; Endereço1, Endereço2.* Por esse exemplo é possível perceber a desvantagem de um banco de dados de famílias de colunas: a escrita de um novo registro é bem mais custosa do que em um banco de dados tradicional. Assim, em um primeiro momento, os bancos tradicionais são mais adequados a processamento de transações online (OLTP) enquanto os bancos de dados de famílias de colunas são mais interessantes para processamento analítico online (OLAP). Vale também ressaltar que os sistemas NoSQL orientados a colunas são conhecidos por possuírem uma capacidade notável de escalar horizontalmente, de modo a conseguirem armazenar grandes quantidades de dados.

Os principais bancos de dados nessa categoria são Cassandra [44] , HBase e Hypertable [41], implementações baseadas no Bigtable [14]. A Tabela 2.1 apresenta um resumo das características de cada tipo de bancos de dados NoSQL.

2.3.2 Distribuição de dados em soluções NoSQL

A maior parte das soluções NoSQL optou pelo armazenamento baseado em tabelas de espalhamento distribuídas (DHT) como mecanismo de distribuição e localização de dados pelo fato de tratar problemas recorrentes em sistemas distribuídos, como escalabilidade, tolerância a falhas e alta disponibilidade. DHTs armazenam blocos de dados em centenas

Bancos de dados NoSQL	Características
Chave/Valor	Modelo Simples Armazenamento de objetos indexados por chaves usadas na busca dos mesmos Exemplos: DynamoDB, Redis e Riak
Orientado a Documentos	Documentos (XML, JSON) são coleções de atributos e valores Bancos de dados sem esquema, adequado para o armazenamento de dados semi-estruturados Exemplos: MongoDB e CouchDB
Orientado a Grafos	Dados representados grafosBusca de itens feita pela navegação dos objetos Exemplos: Neo4J, InfoGrid e HyperGraphDB
Orientado a Colunas	A coluna é o modelo de armazenamento de dados Exemplos: Cassandra, HBase e HiperTable

Tabela 2.1: Resumo de características de bancos de dados NoSQL

ou milhares de máquinas conectadas a uma rede (local ou geograficamente distribuída), replicam os dados para fins de confiabilidade, e localizam rapidamente os dados. Todos os dados armazenados em uma solução DHT são tratados como um par chave-valor (*key-value*).

A DHT provê uma interface genérica, que facilita sua adoção como substrato de armazenamento: a operação *put* armazena os dados associados a uma chave no sistema; a operação *get* recupera os dados; a operação *delete* remove os dados. A Figura 2.4 ilustra o uso desta interface. A conveniência desta interface de armazenamento incentivou a adoção de DHTs mesmo em sistemas pequenos, nos quais a capacidade do sistema para encaminhar consultas sem manter informações de localização é desnecessária. DHTs nestes cenários são atraentes porque além de reduzirem o ônus do desenvolvimento da infra-estrutura, também proporcionam um potencial de expansão.

A distribuição e localização de dados é feita através de uma função de espalhamento. Estas operações são feitas de maneira descentralizada, escalável e balanceada usando chave de correspondência exata [87]. A responsabilidade de manter o mapeamento de chaves para valores é distribuída entre os nós de tal forma que mudanças no conjunto de participantes não causem grande impacto. Isto oferece alta eficiência, descentralização

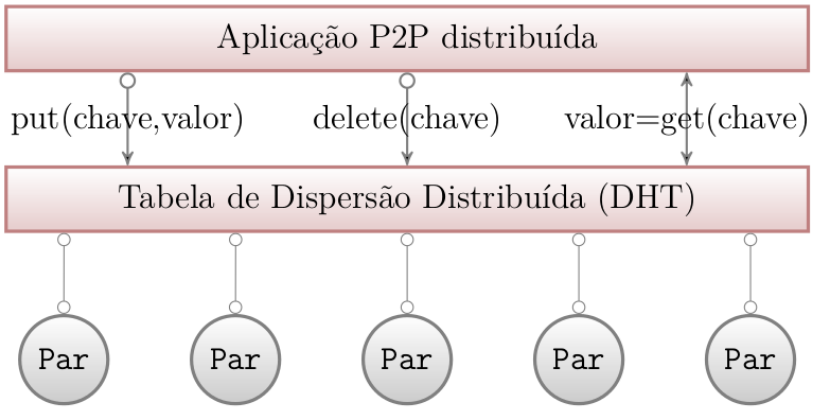

Figura 2.4: Interface de programação para sistemas baseados em DHT[7]

e escalabilidade, permitindo que as DHTs escalem para um número grande de nós e gerenciem chegadas, saídas e falhas contínuas dos mesmos. Os nós membros de um sistema DHT são tratados de forma igual e não há necessidade de um servidor central no sistema [37, 28, 29, 12].

Por usar a função de espalhamento, a DHT não permite que seja possível controlar a localidade e a distribuição de dados. Este o motivo de não usar a DHT na abordagem proposta nesta dissertação, cujo objetivo principal é o controle de localidade de dados em *Cluster* de armazenamento distribuído.

2.4 Localidade de dados

A alocação de dados é uma parte essencial de aplicações distribuídas visto que tem muitos benefícios qunado controlada. Geralmente o objectivo de definição de política de alocação de dados é alcançar o balanceamento de carga, distribuindo os dados uniformemente entre os servidores de armazenamento, independentemente da utilização prevista para os dados [23]. Isso pode ser usado para obter um aumento significativo no desempenho do sistema como um tudo. Os requisitos de dados cada vez maior de aplicações científicas e comerciais,por exemplo, tornam ainda mais importante a alocação controlada de dados.

Em soluções que usam sistemas de arquivos distribuídos, a localidade de um arquivo armazenado tem um impacto significativo em critérios de desempenho, como o tempo de consultas. Ao colocar dados perto dos clientes, pode-se reduzir o número de acessos

remotos, reduzir significativamente a latência de operações, e evitar o congestionamento da rede [59]. Ao levar em consideração a capacidade de nós e a carga de trabalho, a localidade de dados pode ser usada para evitar a sobrecarga de alguns nós, impedindo que estes se transformam em gargalos para todo o sistema [86, 55].

Desta forma, controlar a localidade de dados é muito importante, pois pode garantir que os dados usualmente utilizados em conjunto possam ser alocados em um mesmo servidor, reduzindo a quantidade de comunicações no processamento de suas consultas. Isto evita acesso a múltiplos servidores, e pode ser usado para aproximar informações de suas aplicações usuárias. Todas as consultas relacionadas aos dados coalocados no mesmo servidor são processadas localmente, e somente seus resultados são enviados pela rede. Isto torna o controle de localidade de dados um fator essencial para evitar a alta latência em ambiente distribuído como em uma rede WAN [16]. Garantir que aplicações controlem a localidade de seus dados é, portanto, capital para a escalabilidade e desempenho de sistemas de armazenamento.

Os benefícios alcançados pelo controle de localidade de dados devem ser ponderado com os custos de consulta de dados. De fato, o controle de localidade de dados pode exigir o uso de algum mecanismo, que armazena o mapeamento entre os itens de dados e os nodos, isto é, informações de localidades dos dados. Infelizmente, pode existir custos adicionais ligados às pesquisas e consultas destas informações. Devido a este problema, muitos sistemas usam estratégias de alocação de dados simples, como hashing consistente [40].

Nesta dissertação, foi explorado a possibilidade de controle de localidade que oferecem alguns SADs, combinado com o uso dos metadados para armazenar informações sobre as localidades de arquivos para implementar um repositório com controle de localidade de dados. Não foi utilizada técnica baseada em DHT,pois estes não oferecem a possibilidade de controle de localidade. Porém, vale ressaltar que existe um *tradeoff* entre a escalabilidade e a replicação na abordagem usada nesta dissertação. Detalhes sobre este ponto são mencionados no capítulo 5.

Este capítulo apresentou os conceitos relacionados ao armazenamento de dados, SADs,

bancos de dados NoSQL e localidade de dados. Segue o capítulo 3 que apresenta alguns trabalhos correlatos usados para desenvolver esta dissertação.

CAPÍTULO 3

TRABALHOS CORRELATOS

Neste capítulo são apresentados alguns trabalhos relacionados ao armazenamento distribuído. Algumas características importantes na comparação entre sistemas de armazenamento distribuído são o método utilizado para dispersar e recuperar os dados, a escalabilidade e disponibilidade. A dispersão e recuperação influenciam a escalabilidade do sistema, uma vez que elas são determinantes no tempo de recuperação dos dados [57]. Duas técnicas são comumente utilizadas como *backend* de sistemas sistemas de armazenamento distribuídos: as tabelas de espalhamento distribuídas (DHTs) e sistemas de arquivos distribuídos (SADs).

3.1 Sistemas de Arquivos Distribuídos

Muitos SADs com diferentes arquiteturas e características têm sido implementados nos últimos anos com objetivo de suprir as principais características de sistema em ambiente distribuído [68, 21]. Eles apresentam uma estrutura não centralizada e os dados são dispersos em um conjunto de nós, que tem como objetivo aumentar a capacidade de armazenamento da solução. Os clientes e aplicações usuárias não possuem acesso direto à estrutura de disco subjacente e a interação é realizada através de um protocolo pré-estabelecido. A dispersão dos arquivos é transparente e é responsabilidade da solução distribuída localizar o arquivo e transportá-lo até a aplicação que solicitou a informação.

Dentre os SADs, o *Hadoop Distributed File System*(HDFS) [69] e o *Google File System*(GFS) [27] não implementam meios de alocação controlada dos arquivos. Eles dividem os arquivos em fragmentos que são alocados em servidores com maior disponibilidade de recursos (menor carga para o HDFS ou menor uso de armazenamento para o GFS).

- O GFS possui uma arquitetura baseada em um nó mestre com informações sobre os arquivos (metadados) e multiplos servidores (*chunkservers*) que garantem o armaze-

namento persistente de arquivos. Os servidores armazenam os fragmentos de dados em discos locais. Para confiabilidade, cada fragmento é replicado em vários servidores. Por padrão, o fator de replicação é três. Porém, os usuários podem especificar diferentes níveis de replicação [27]. Os arquivos são organizados hierarquicamente em diretórios e são localizados através de seus *full path names*. O GFS implementa a tolerância a falhas e a replicação para garantir disponibilidade e integridade de dados [65]. É feito um monitoramento constante dos nós para detectar possíveis falhas.

- Por sua parte, a arquitetura do HDFS é centralizada e metadados são armazenados em nós dedicados chamados de *NameNode*. Os mesmos são também responsáveis por gerenciar o espaço de nomes do sistema de arquivos e controlar os acessos aos arquivos. Dados são replicados em servidores específicos nomeados de *DataNodes*. Requisições de escrita e/ou leitura de arquivos são realizadas em *DataNodes*. O espaço de nomes de arquivos e diretórios é hierárquico. HDFS oferece uma biblioteca que permite a usuários escrever, ler e deletar arquivos, e criar e deletar diretórios. Estas operações são realizadas de forma transparente usando API em C, Java ou REST. Os nós em HDFS são totalmente conectados e se comunicam para detectar falhas e manter o sistema disponível.

Outros SADs difundidos são o Ceph [81], PVFS [64], Lustre [21], GlusterFS [18] e FusionFS [88].

- O Ceph, baseado em objetos, mapeia arquivos para um ou mais objetos. O Ceph cria uma abstração entre o armazenamento físico e o lógico. Essa abstração é o equivalente a uma unidade de armazenamento e é chamada de *pool*. Os *pools* são mapeados pelo algoritmo CRUSH [82] e o resultado é conhecido como CRUSH Map. Neste mapa, constam as informações sobre a associação que existe entre os *pools* e seu armazenamento nos servidores do ambiente. Desta forma, a replicação e localização das cópias ocorre automaticamente. O Ceph baseia-se no CRUSH Map para determinar quando, quantas e onde as cópias devem ser mantidas. Por

meio da integração das API com o CRUSH Map, o Ceph fornece a possibilidade do usuário criar um conjunto de regras que podem ser usadas para organizar o armazenamento dos arquivos de acordo com sua utilização. Este recurso pode ser usado para controlar a localidade de dados. Isto motivou o uso deste SAD nesta dissertação.

- Como o Ceph, o PVFS também permite que a alocação de arquivos seja definida através de uma interface, na qual é possível especificar o nome do nó no momento da criação do arquivo. Contudo, ele não dá suporte à replicação. O PVFS fragmenta dados em arquivos e os distribui de maneira uniforme entre os nós para alcançar alto desempenho [64]. Isto possibilita um acesso paralelo às partes disjuntas de dados.

- O Lustre [21] foi implementado para remover os gargalos tradicionalmente encontrados em SADs e prover alto desempenho e escalabilidade em *clusters* de dezenas de milhares de servidores. Ele apresenta uma arquitetura centralizada que não fornece cópias de dados nem de metadados, e que possui uma estrutura de distribuição de dados semelhante ao PVFS, com a diferença que oferece a possibilidade de ter nós de metadados operando em pares para tolerância a falha. O seu modelo de armazenamento é baseado em objetos como o Ceph. A forma de distribuir arquivos nos *Object Storage Target* (OST) é configurável manualmente pelo usuário. O balanceamento de carga é realizado pelo servidor de metadados (MDS) e a detecção de falha é feita durante as operações do cliente e não por meio de troca de mensagens realizada periodicamente pelos nós como é o caso dos outros SADs. Quando um cliente abre um arquivo, ele recebe um *layout* do mesmo, transferido a partir do MDS. O cliente usa então esta informação para realizar leituras e escritas no arquivo, interagindo diretamente com os nós, onde os objetos são armazenados.

- O GlusterFS, por sua parte, possui uma arquitetura cliente-servidor na qual não existe o servidor de metadados [84]. Dados e metadados são armazenados em diversos dispositivos ligados a diferentes nós, cujo conjunto é chamado de *volume*. A localização destes dados é feita pelo algoritmo *Elastic Hashing Algorithm* (EHA)

que utiliza uma função de espalhamento (*hash function*) [67]. A mesma função garante que todos os dispositivos de armazenamento tenham a mesma quantidade de arquivos (balanceamento de carga). O cliente pode interagir, enviando e recuperando arquivos de um *volume* mediante a API REST fornecida pelo GlusterFS. O GlusterFS garante a replicação usando o RAID 1. Vale ressaltar também que quando um nó não está disponível, o mesmo é removido do sistema.

- Diferentemente dos outros SADs, o FusionFS é uma solução implementada para dar suporte às aplicações científicas com grande escalabilidade e alto desempenho [88]. Na sua arquitetura, todos os nós são conectados e os clientes têm a visão global de todos os arquivos, sejam eles locais ou remotos. O espaço de nome é mantido por uma tabela de espalhamento (DHT), que tem como função a distribuição de metadados nos nós. A tolerância a falha é garantida pela replicação dos metadados.

Uma das dificuldades de utilização de SADs como *backend* para SGBDs distribuídos é que a unidade de armazenamento é o arquivo, uma granularidade maior que aquela de manipulação de SGBDs, baseada em itens de dados.

Apesar de existirem diferenças, a maioria dos SADs compartilham características como escalabilidade, desempenho, balanceamento de cargo, transparência, fragmentação de dados (*striping*) entre outras. Em relação aos metadados, SADs como Lustre e PVFS optam pela separação dos mesmos com os dados. Isto permite modelar nós de armazenamento relativamente simples e rápidos, enquanto operações complexas, envolvendo atributos de arquivos e permissões, são tratados pelos servidores de metadados. A tabela 3.1 apresenta o resumo das principais características dos SADs mencionados nesta dissertação.

3.2 Sistemas baseados em DHT

Grande parte de sistemas NoSQL, como *Scalaris* [66] e *MemcacheDB* [77], utilizam protocolos baseados em DHT como mecanismo de dispersão de dados. Como mencionado anteriormente, a DHT é o ponto principal que permite que a escalabilidade seja alcançada. Nestes sistemas, a alocação das informações é feita pela função de espalhamento (*hash*).

	PVFS	Lustre	GFS	HDFS	GlusterFS	FusionFS	Ceph
Arquitetura	Cliente-Servidor	Centralizada	Distribuída	Centralizada	Cliente-Servidor	Distribuída	Distribuída
Localização de dados	Índice	Índice	Índice	Índice	Elastic Hashing Algorithm	DHT	Hash (CRUSH)
Detecção de falha	Troca de Mensagens	Manual	Troca de Mensagens	Troca de Mensagens	Troca de Mensagens	Troca de Mensagens	Troca de Mensagens
Replicação	Não	Não	Sim	Sim	Sim	Sim	Sim
Balanceamento de carga	Manual	Não	Automático	Automático	Manual	Fraco	Manual
Consistência	X	Forte	Forte	Forte	Forte e Fraco	Forte e Fraco	Forte

Tabela 3.1: Tabela comparativa dos SADs

Isto torna o controle de localidade de dados pela aplicação muito baixo ou praticamente nulo.

- O modelo de armazenamento usado pelo Scalaris é o chave-valor. Este banco de dados NoSQL utiliza uma rede de sobreposição para alcançar disponibilidade através da replicação de dados e transições distribuídas para manutenção da consistência dos dados [66]. A sua arquitetura é composta por três camadas (Comunicação, Replicação e Transação) que implementam a escalabilidade e consistência eventual (comunicação), a disponibilidade (Replicação) e as propriedades ACID transicionais (Atomicidade, Consistência, Isolamento, Durabilidade). Operações de distribuição e localização de dados são baseadas no protocolo DHT Chord [72]. Este armazena as chaves em ordem lexicográfica, possibilitando consultas por abrangência, sem controle de localidade.

- O MemcacheDB utiliza o armazenamento distribuído chave-valor, e é conhecido por ter alto desempenho de escrita e leitura, alta disponibilidade, replicação de dados e por ter compatibilidade com o protocolo Memcache [10]. O Memcache é uma solução de *caching*, trata-se de uma solução de persistência e recuperação de dados.

Outros sistemas NoSQL baseados em DHT, com noções de localidade para distribuição dos dados, são explorados nos trabalhos [63, 8] e estão presentes em soluções comerciais envolvendo nós geograficamente distribuídos, como Cassandra [44] e o Amazon SimpleDB [31]. A localidade dos dados nestes repositórios distribuídos tem sido proposta com o intuito de permitir que dados fiquem próximos das suas aplicações. Tanto o Cassandra e como o Amazon SimpleDB são soluções orientadas a colunas.

- Cassandra foi projetado para gerir grandes volumes de dados, alcançando simultaneamente uma alta disponibilidade, escalabilidade e tolerância a falha [44]. Outra propriedade do Cassandra é a sua velocidade de escrita, sem que isso afete a eficiência das leituras de dados. A replicação de dados é garantida, porém isso acarreta um *tradeoff*. Réplicas distintas de base de dados podem possuir dados diferentes e criar inconsistência.

- Diferentemente do Cassandra que é um banco de dados *open source*, o Amazon SimpleDB é proprietário. A sua arquitetura é baseada no S3 da Amazon (*Simple Storage Service*) [31], na qual dados são distribuídos através dos dispositivos de armazenamento na rede distribuída e escalável da Amazon. Dados são armazenamento nos domínios definidos por nomes. Estes domínios contêm itens, cada um possuindo nomes únicos e atributos. O SimpleDB permite que usuário organize seus dados em domínios, porém sem ter noções das suas localidades, isto, em que servidores estes dados estão armazenados. Uma característica fundamental deste sistema é a utilização de consistência eventual, ou seja, alterações podem não ser vistas por operações de leitura subsequentes. Embora na maioria das vezes essa situação não ocorra, a possibilidade deve ser levada em conta se houver requisitos de coerência de dados.

Estas soluções, que adotaram noções de localidade na distribuição dos dados, são baseadas em um modelo, no qual a aplicação utiliza a ordem lexicográfica das chaves para organizar as informações. Contudo, esta abordagem, no entanto, obriga a aplicação a modificar ou adequar as chaves, adicionando prefixos para possibilitar seus agrupamentos

e, apesar de as aproximarem, não há garantias na localidade delas no mesmo servidor.

Outras soluções que procuram atender o requisito de localidade baseado em DHT incluem o AutoPlacer [58], Infinispan[1] e o Spanner [16].

- O AutoPlacer apresenta uma técnica de otimização da colocação de dados que são considerados críticos para o desempenho do sistema, isto é, dados frequentemente acessados. O AutoPlacer atinge isso coletando dados estatísticos relacionados às consultas remotas destes dados. Apenas suas réplicas são realocadas, e consequentemente terão a localidade controlada. O controle de localidade não é possível na primeira escrita de um determinado item no repositório, visto que este não possui nenhum dado estatístico sobre suas consultas.

- O Infinispan não oferece controle de localidade. A distribuição de dados é feita por uma função *hash*. Vale salientar que o Infinispan foi usado como repositório para validar o AutoPlacer.

- O Spanner foi implementado pelo Google para ser um banco de dados globalmente distribuido e com alta disponibilidade de dados. Ele oferece alta escalabilidade permitindo que bilhões de colunas de dados sejam armazenados em milhões de nodos. Quando uma aplicação consulta um dado, a requisição é encaminhada ao *datacenter* geograficamente próximo que possui o dado ou sua réplica. Embora o Spanner permita que aplicações especifiquem os *datacenters* a serem usados para armazenar dados, ele não oferece o controle de localidade exata de dados. Outra caracterísa importante desta solução é a possibilidade de consultar versões antigas de arquivos.

A Tabela 3.2 ilustra uma síntese das principais propriedades de dos sistemas NoSQL apresentados nesta seção.

Ao contrário destas soluções, esta dissertação apresenta o ALOCS, uma solução de armazenamento distribuído que oferece o controle de localidade exata de dados. Ele adota a interface simples baseada no modelo chave-valor das DHTs, porém com um módulo de

[1] infinispan.org

	Cassandra	Scalaris	MemcacheDB	SimpleDB	Spanner
Modelo de dados	Coluna	Chave-Valor	Chave-Valor	Coluna	Tabelas relacionais semi - esquematizadas
Replicação	Sim	Sim	Sim	Sim	Sim
Localização de dados	Consistent Hash	Consistent Hash	Consistent Hash	Consistent Hash	Consistent Hash
Consistência	Eventual	Forte	Eventual	Eventual	Forte
Escalabilidade	Alta	Alta	Alta	Alta	Alta
Disponibilidade	Alta	Alta	Alta	Alta	Alta

Tabela 3.2: Tabela comparativa dos sistemas NoSQL

gerenciamento de metadados acoplado para garantir o controle de localidade de dados. O armazenamento é feito no SAD Ceph, e módulo de metadados baseia-se no *Zookeeper* [39]. Estes dois sistemas são apresentados no capítulo 4.

CAPÍTULO 4

CEPH E ZOOKEEPER

O capítulo anterior apresentou alguns Sistemas de Arquivos Distribuídos (SADs) e também algumas soluções NoSQL estudados para a elaboração da presente dissertação. Os trabalhos mencionados são usados para prover sistemas de armazenamentos distribuídos. Para cada solução, foram discutidos aspectos como a arquitetura, o mecanismo de distribuição e localização de dados, a replicação, a escalabilidade entre outros. Em relação ao controle de localidade de dados, observou-se que as soluções NoSQL, baseadas em tabelas de espalhamento distribuídas (DHTs), usam uma função de espalhamento *(Hash Function)*. Esta função é responsável pela distribuição e localização de dados. Isto torna o controle de localidade de dados pela aplicação nulo. Por outro lado, os SADs usam os metadados para localizar arquivos. Alguns desses SADs oferecem a possibilidade de especificar locais onde arquivos podem ser armazenados, como é o caso do PVFS. O Ceph é um outro SAD que permite um controle parcial de localidade de dados. A presente dissertação explorou essa possibilidade para implementar um sistema de armazenamento distribuído com controle parcial de localidade.

A solução apresentada nesta dissertação combina dois sistemas, o Ceph e o *Zookeeper*. Este capítulo traz mais detalhes sobre esses dois sistemas, ressaltando dados sobre suas características, arquiteturas entre outros aspectos.

4.1 Ceph

4.1.1 Conceitos e arquitetura

O Ceph [81] é um SAD baseado no conceito de objeto, onde um arquivo é mapeado para um ou mais objetos e os discos convencionais são substituídos por dispositivos denominados *Object Storage Devices* (OSDs). Estes substituem as interfaces tradicionais baseadas

em blocos, permitindo escrever uma quantidade maior de bytes por operação de escrita [9]. Sistemas baseados neste conceito, distribuem as tarefas relacionadas ao gerenciamento de dados, como alocação, replicação e recuperação entre os dispositivos disponíveis, permitindo paralelizar as operações com maior eficiência. O repositório de armazenamento implementado nesta dissertação é baseado no Ceph. Esta escolha se justifica pelo fato do Ceph ser distribuído, por possuir um algorítimo de distribuição de dados que é responsável por alocar e replicar os dados no *cluster* [81]. Este algoritmo faz a distribuição a partir de um conjunto de regras, que indicam onde um determinado objeto e suas réplicas devem ser alocados. Portanto, as regras definem as estratégias de alocação e replicação, isto é, políticas de distribuição que permitem que seja especificado como o CRUSH irá alocar objetos e réplicas nos *pools*. Este recurso permite que sejam exploradas as características do ambiente físico e que o controle da localidade dos dados seja garantido. Ao implementar estas regras no *cluster*, as políticas de colocação do CRUSH podem separar réplicas de objetos em diferentes domínios de falhas, mantendo a distribuição balanceada [82]. Esta abordagem concede ao Ceph flexibilidade.

O elemento principal da arquitetura é o RADOS [83], responsável pela distribuição balanceada de dados através de um *cluster* dinâmico e heterogêneo. O Ceph monitora o espaço de disco usado por cada OSD, e transfere dados para outros OSDs para permitir o balanceamento de carga, que é garantido em dois níveis: dados e metadados. Metadados muito acessados são replicados ou transferidos em nós menos sobrecarregados. A transparência é garantida por meio de uma API REST. O RADOS garante uma visão consistente da distribuição dos dados e consistência no acesso aos objetos através do uso de um mapa do *cluster* versionado. O Ceph foi projetado com o intuito de aproveitar a inteligência dos OSDs, portanto o RADOS transmite à eles as tarefas de: gerenciamento da replicação de objetos, expansão do *cluster* e detecção de falhas (feita pela troca de mensagens periodicamente) e recuperação.

O Ceph *Storage Cluster* é a base para todas as aplicações do Ceph, sendo executado sobre o RADOS, é composto por dois *daemons*: Ceph OSD que armazena dados como objetos em um nó de armazenamento; e Ceph *Monitor* que gerencia o mapa do *cluster*. A

Figura 4.1 ilustra um *cluster* de armazenamento de objetos composto por muitos OSDs. O conjunto de *Monitors* gerencia o mapa do *cluster* de armazenamento. Cada cliente possui uma interface simples usada para interagir com o sistema.

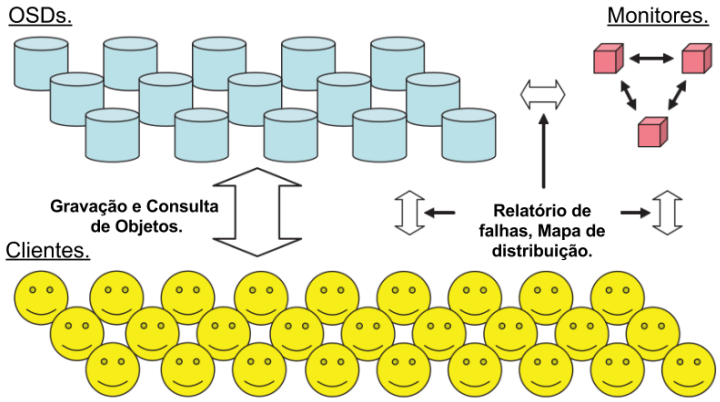

Figura 4.1: Arquitetura do Ceph[83]

O mapa do *cluster* é o meio pelo qual as informações sobre a distribuição dos dados, e alterações no estado dos OSDs são mantidas, o gerenciamento do sistema é feito exclusivamente através dele. Para que não ocorra uma sobrecarga no sistema com acessos ao mapa, é mantido uma cópia principal no monitor e réplicas entre os nós de armazenamento e os clientes.

Um dos mapas que compõem o mapa do *cluster* é o CRUSH *Map* que especifica os OSDs e os *hosts* [82]. Também faz parte do mapa um conjunto de regras que especificam como os objetos e suas réplicas devem ser alocados, denominadas regras de alocação ou CRUSH *Ruleset*, elas são utilizadas pela função CRUSH para alocar e replicar os objetos.

4.1.2 Função CRUSH

A função CRUSH é uma função de *hash* determinística que permite ao administrador definir políticas de alocação flexíveis de dados sobre a estrutura de forma hierárquica. O que diferencia o CRUSH das funções de *hash* comuns é o fato de ser mais estável, no sentido de que, quando um ou mais dispositivos são adicionados ou removidos, apenas uma pequena parte dos dados é movida para manter o balanceamento [81].

A função CRUSH possibilita que o RADOS distribua os objetos e suas réplicas sem a

necessidade de armazenar uma grande quantidade de metadados, pois a localização dos objetos é obtida através de um cálculo que utiliza o identificador do objeto, atribuído pelo OSD primário e o CRUSH *Map* como entrada.

4.1.3 Alocação e Replicação de dados

O Ceph armazena os objetos e suas réplicas em uma partição lógica denominada *Pool*. A aplicação cliente ao gravar ou recuperar dados deve indicar o *Pool* que deve ser utilizado na operação.

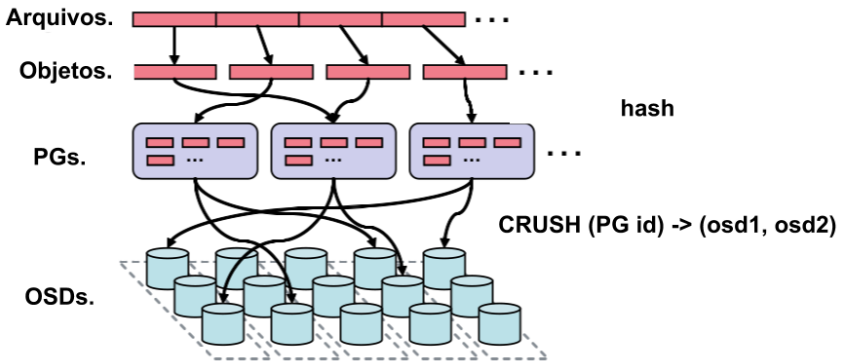

Figura 4.2: Distribuição de dados no Ceph [81]

O *Pool* é organizado por grupos de alocação (*Placement Group* - PG), cada objeto pertence a um único PG, e cada PG é associado a uma lista de OSDs, sendo um primário, e os demais réplicas. A Figura 4.2 mostra como arquivos são distribuídos entre vários objetos, agrupados em PGs, distribuídos por sua vez nos OSDs via a função CRUSH. O servidor de metadados gerencia o espaço de nome e a consistência do sistema assim como a desempenho das requisições de metadados enquanto os OSDs garante a desempenho das operações de escrita e leitura de dados.

A quantidade de grupos em um *Pool* é especificada pelo administrador do sistema, sendo determinada pelo grau de replicação, relacionado ao *Pool*, e a quantidade de OSDs disponível para o mesmo. Esta estrutura foi criada para otimizar a movimentação dos objetos no *cluster*, pois o custo para alocação e manutenção de metadados por objeto é computacionalmente alto [81].

A Figura 4.3 mostra a função dos *Pools* na arquitetura. O cliente recupera uma réplica

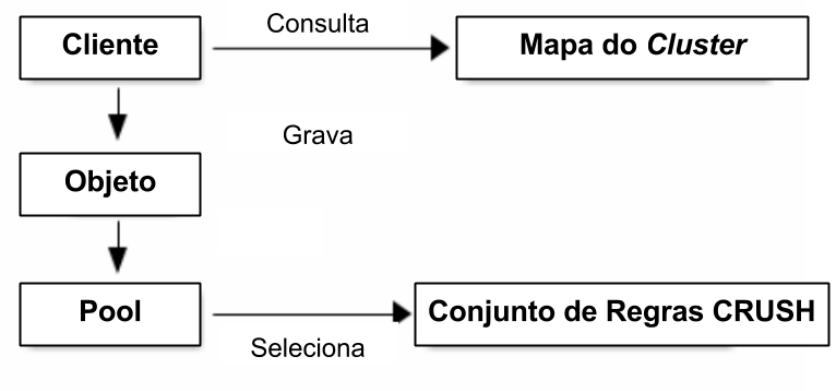

Figura 4.3: Função dos *Pools* [36]

do mapa do *cluster* através de um Ceph Monitor e escreve um objeto no *Pool*, o grupo de alocação é determinado por uma função de hash baseada nos identificadores do *Pool* e do objeto. O CRUSH *Map* é consultado para que o CRUSH distribua os objetos a partir do que foi especificado pelas regras de alocação associadas ao *Pool*.

As regras de alocação são o meio pelo qual o CRUSH identifica quais OSDs devem ser utilizados para alocar determinado objeto e suas réplicas, expressas por meio de uma sequência de comandos, para que o CRUSH percorra a hierarquia de *hosts*, especificada no CRUSH *Map*. Estas regras proporcionam um meio de obter escalabilidade, pois podem ser alteradas de acordo com o crescimento do ambiente, e permitem distribuir dados conforme quantidade de acessos, estabelecendo que dados mais acessados sejam alocados em OSDs que suportem a carga de operações.

O protocolo de replicação utilizado por padrão no Ceph é uma variação do *Primary-Copy* [3], em que muitas vezes é desejável que após uma falha, a próxima réplica se torne o novo dado primário. Isso garante o controle de concorrência em dados replicados e também não replicados. Porém o RADOS implementa mais dois protocolos, *Chain* e *Splay* [83]. De forma geral os clientes enviam operações de escrita e leitura para um único OSD e o *cluster* garante que todas as réplicas serão atualizadas corretamente, após a atualização uma única confirmação é enviada para o cliente. Os estágios dos protocolos estão ilustrados na Figura 4.4.

No *Primary-Copy* as réplicas são atualizadas em paralelo e operações de escrita e leitura são encaminhadas para o OSD primário, o que o diferencia dos outros, são a forma

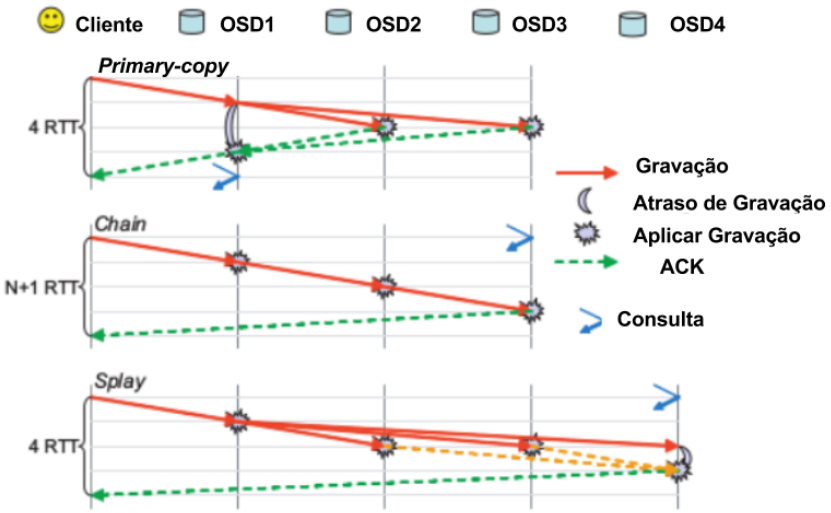

Figura 4.4: Protocolos de replicação [81]

que as atualizações são distribuídas e o OSD que recebe as operações de leitura.

O Ceph foi usado para implementar o repositório de armazenamento da solução apresentada nesta dissertação.

4.2 Zookeeper

4.2.1 Conceitos e arquitetura

O *Zookeeper* é um serviço de coordenação de processos com alta disponibilidade e confiabilidade que pode ser usado para implementar soluções de sincronização, registro de nomes, configuração, eleição de *leader*, notificação de eventos, bloqueio, mecanismo de prioridade em fila em grandes sistemas distribuídos [39]. Este sistema funciona usando processos distribuídos para coordenar os demais através de um espaço de nomes hierárquico compartilhado. Soluções baseadas no *Zookeeper* podem ser implementadas por meio de um conjunto simples de primitivas que ele fornece.

O *Zookeeper* pode ser usado de duas forma: no modo *stand alone* ou no modo de replicação. No modo *stand alone*, o serviço é centralizado em um único nó, ao passo que no modo de replicação, os dados são replicados em um *cluster* de nós chamado de *ensemble* (Figura 4.5) [38].

O *Zookeeper* usa o mecanismo de fiscalização (*watch*) para permitir que aplicações

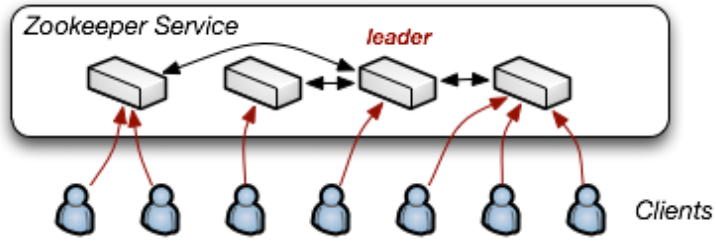

Figura 4.5: *Zookeeper ensemble* [38]

clientes consigam fazer o Cache de dados sem gerenciar diretamente o Cache dos próprios clientes. Com este mecanismo, o cliente pode fiscalizar atualizações de uma dado objeto e receber a notificação assim que estas atualizações ocorrerem.

4.2.2 Modelo de armazenamento e API

A unidade de armazenamento usada no *Zookeeper* é denominada de *znode*. São estruturas organizadas de acordo com um espaço de nomes hierárquico, semelhante às estruturas de dados de diretórios (Figura 4.6) [71]. Nessa hierarquia, os *znodes* são objetos que aplicações clientes manipulam. Eles podem ser de tipo *persistent*, isto é, criado e removido explicitamente pelo cliente, ou *ephemeral*, quando existe apenas enquanto a sessão que o criou permanece ativa.

Cada *znode* possui um metadado com *time stamps* e contador de versões. Isso permite que clientes sejam capazes de controlar mudanças que podem ocorrer em *znodes* com intuito de executar atualizações baseadas em versões destes *znodes* [35].

Um dos objetivos do *Zookeeper* é prover uma interface de programação simples para implementar soluções simples e mais complexas [35]. por isso, ele suporta apenas seguintes operações:

- ***create (path, data, flags):*** cria um *znode* usando o nome do caminho passando*(path)*, armazena o seu metadado *(data)* e retorna o nome do novo *znode*. O parâmetro *flags* permite que o cliente especifique o tipo do *znode (persistent ou ephemeral)*.

- ***delete (path, version):*** remove o *znode* indicado pelo *path*, o *znode* tem que ser

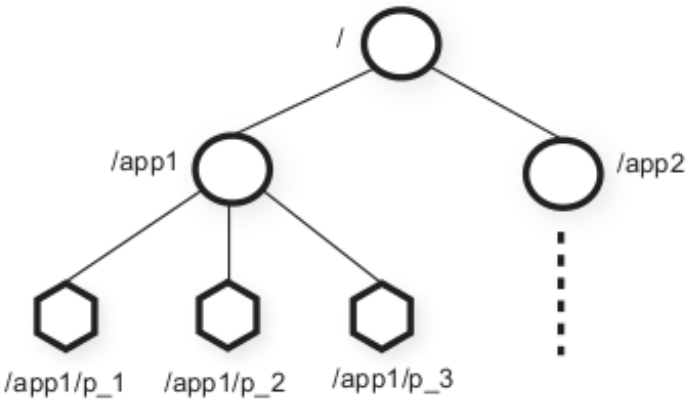

Figura 4.6: Espaço de nome hierárquico do *Zookeeper* [35]

da versão indicada.

- ***exists (path, watch):*** retorna *true* se o *znode* especificado pelo *path* existe, e *false* no caso contrário. O parâmetro *watch* permite que seja atribuído um mecanismo de fiscalização.

- ***getData (path, watch):*** retorna dados e metadados, como informações de versão, associados ao *znode*. Nesta requisição, o parâmetro *watch* funciona da mesma forma que a requisição *exists()*.

- ***setData (path, data, version):*** escreve dados no *znode (path)* se a versão especificada é a mesma do *znode*.

- ***getChildren(path, watch):*** retorna o conjunto de nomes de todos os filhos de um *znode*.

- ***sync (path):*** espera que todas as alterações pendentes no início da operação se propaguem no nó que o cliente está conectado.

Essas operações podem ser agrupadas em operações de atualização/escrita (create, setData, delete) e leitura (*exists, getData, getChildren*).

4.2.3 Características do Zookeeper

O *Zookeeper* oferece um alto desempenho na recuperação de dados e permite escalar até centenas de milhares de transações por segundo através de um *cluster* de nós devidamente configurados [38]. A recuperação rápida de dados é decorrente do fato que estes são mantidos em memória. Isso permite também que o sistema lide com altas cargas, porém, limita o armazenamento somente a pequenas informações e não grandes *blobs*.

A replicação é outra característica importante deste sistema. Ela permite que seja alcançada a disponibilidade de serviços. A arquitetura do *Zookeeper* suporta alta disponibilidade através de redundância de serviços no modo replicação [35]. Aplicações clientes podem ler e escrever de / para os nós com serviço de configuração compartilhada e atualizações de dados são feitas de forma ordenada.

Replicar serviços permite também obter a tolerância a falhas, que é uma característica muito requisitada em serviços de coordenação em ambiente distribuído [71]. Na inicialização de um *ensemble*, um nó líder é eleito para coordenar o *ensemble*. O cliente se conecta a apenas um nó membro do *ensemble* por meio de uma conexão TCP. Todos os nós armazenam cópias de dados que os clientes podem consultar. A escrita de dados é feita apenas através do nó líder, que necessita do consenso da maioria dos nós (metade do número de nós que compõem o *ensemble* mais 1) para efetivar a operação. Em termo ilustrativo, um *ensemble* de 6 nós só pode suportar falha de até 2 nós, mantendo assim o serviço disponível com 4 nós ativos.

Em um ambiente de produção, deve implantar um *ensemble* com um número ímpar de servidores. Enquanto a maioria dos servidores do *ensemble* estiver disponíveis, o serviço *ZooKeeper* estará disponível. A quantidade mínima recomendado é de três servidores [1]. Além disso, o processo do servidor *ZooKeeper* deve ter o seu próprio armazenamento em disco dedicado, se possível.

Existe um *trade-off* entre a disponibilidade e a consistência de dados no *Zookeeper* [35]. A fim de atingir alta disponibilidade, o *Zookeeper* garante consistência fraca das réplicas. As leituras podem ser respondidas por qualquer nó membro do *cluster Zookeeper*, porém

[1] https://zookeeper.apache.org/

as respostas retornadas podem ser obsoletas. Em sistemas com atualizações frequentes de dados, isso pode ser um problema muito sério.Esse não é o caso da solução proposta nesta dissertação, pois atualizações acontecem apenas na criação e remoção de buckets (e não dos pares chave-valor).

No capítulo 5, é apresentado o ALOCS, um sistema de armazenamento distribuído com garantia de controle de localidade de dados. A sua arquitetura, os principais componentes, assim como o seu modelo de armazenamento de dados também são detalhados. O *Zookeeper* foi usado para implementar o componente que garante o controle de localidade de dados.

CAPÍTULO 5

ALOCS - REPOSITÓRIO DE DADOS CHAVE-VALOR COM CONTROLE DE LOCALIDADE DE DADOS

Neste capítulo é apresentado o ALOCS, um repositório chave-valor que permite à aplicação usuária controle sobre a localidade dos dados. Ele foi projetado para dar suporte a SGBDs nos quais a coalocação de dados envolvidos em uma consulta é fator determinante para o seu desempenho. As próximas seções descrevem o modelo proposto (Seção 5.1), a arquitetura do sistema (Seção 5.2) e sua implementação (Seção 5.3). Dois pontos devem ser considerados para apoiar sua arquitetura. O primeiro é o armazenamento de dados, que envolve não só o modelo de armazenamento adotado, mas também a distribuição de dados nos dispositivos de armazenamento. O segundo ponto é a definição das operações de interfaces entre os principais componentes do sistema.

Antes de apresentar a arquitetura do ALOCS, segue a descrição do modelo de armazenamento usado pelo sistema proposto nesta dissertação.

5.1 Modelo de Armazenamento

O modelo de armazenamento usado pelo ALOCS é baseado em quatro conceitos:

(1) par chave-valor, que corresponde à unidade de acesso. O conceito chave-valor é usado pela maioria das soluções NoSQL, conforme apresentado na subseção 2.3.2 do capítulo 2. A sua utilização permite a visualização do banco de dados como uma tabela *hash*. Todo o banco de dados é composto por conjunto de chaves, cada uma associada a um único valor [33]. Trata-se de um modelo flexível que possui operações simples de manipulação de dados *(get(), put() e delete())*, que pode empacotar dados sem esquema (qualquer tipo de dados - subgrafos, conjunto de linhas, enre outros) e que pode servir de base para implementar vários tipos de aplicação;

(2) *bucket*, que consiste de um conjunto de pares chave-valor e corresponde à unidade

de comunicação entre servidores. Um *Bucket* é formado por um cabeçalho onde vão informações como, a quantidade máxima de chaves permitidas, um conjunto de informações para localização dos pares chave-valor denominado *slot* e um mapa de bits para controle dos *slots* livres. Quando a aplicação requisita um par chave-valor, a localização do *Bucket* onde a chave está armazenada é recuperada. Este *Bucket* é lido por completo e posto em memória a fim de reduzir o tempo de busca das requisições feitas para o mesmo *Bucket*. É retornado para aplicação apenas o par correspondente á chave solicitada;

(3) diretório, formado por um conjunto de *buckets* e é a unidade de replicação do modelo; e

(4) servidor, que contém um conjunto de diretórios, e corresponde a um servidor físico do sistema de armazenamento distribuído.

Os componentes do modelo de armazenamento definem uma hierarquia, como mostra a Figura 5.1, que é usada para a localização dos dados. Ou seja, um caminho do tipo /Servidor/Diretório/Bucket/Chave, determina o local de persistência física de um par chave-valor.

O ALOCS oferece um conjunto de operações associado a cada um destes conceitos, tais como a adição e remoção de servidores, bem como adição, remoção de diretórios em determinados servidores. Cada *bucket* é criado com um conjunto sequencial de Chaves (intervalo de chaves) através da operação `create_bucket(dir, idBucket, chaveIni, chaveFim)`. Desta forma, para executar a operação `put(chave, valor)`, o par é criado dentro do *bucket* responsável pelo intervalo que contém a chave, que já está previamente associado a um diretório, que por sua vez, pode estar replicado em um conjunto de servidores.

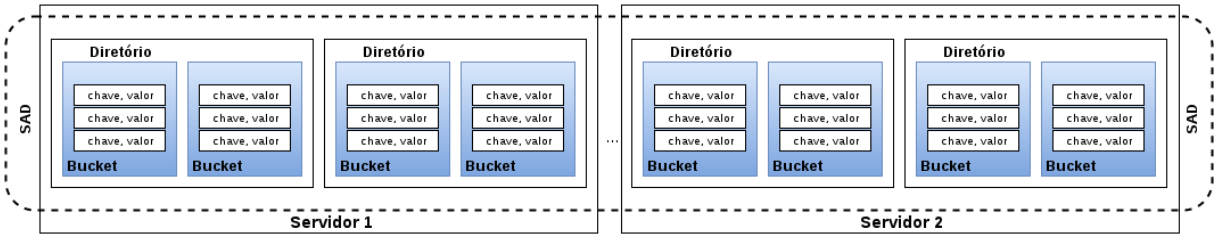

Figura 5.1: Modelo de armazenamento

A interface de aplicação oferecida pelo ALOCS para pares chave-valor é idêntica à interface dos repositórios chave-valor desenvolvidos sobre DHTs, ou seja, com as operações `get(chave)`, `put(chave, valor)` e `rem(chave)`. Contudo, a execução de uma operação `get` recupera não apenas o valor associado à chave, mas o *bucket* no qual ele está armazenado. Assim, acessos subsequentes a chaves que pertencem ao mesmo *bucket* não requerem novas comunicações entre servidores, se o *bucket* estiver no Cache. Isso depende da política de Cache. O objetivo é dar à aplicação o controle sobre a distribuição dos dados para que ela explore esta funcionalidade para minimizar o custo de comunicação no processamento de consultas. Mais detalhes sobre as interfaces são apresentados na seção 5.4.2.

Como mencionado na introdução, o ALOCS foi implementado para ser utilizado como *backend* de armazenamento para SGBDs, dando suporte a um modelo de *clusterização* distribuído. A Figura 5.2 ilustra o mapeamento do modelo relacional para o modelo do ALOCS. Nesta figura, a tabela cliente é mapeada para o diretório, uma deteminada quantidade de registros ou tuplas para o *bucket* e cada qual corresponde ao par chave-valor.

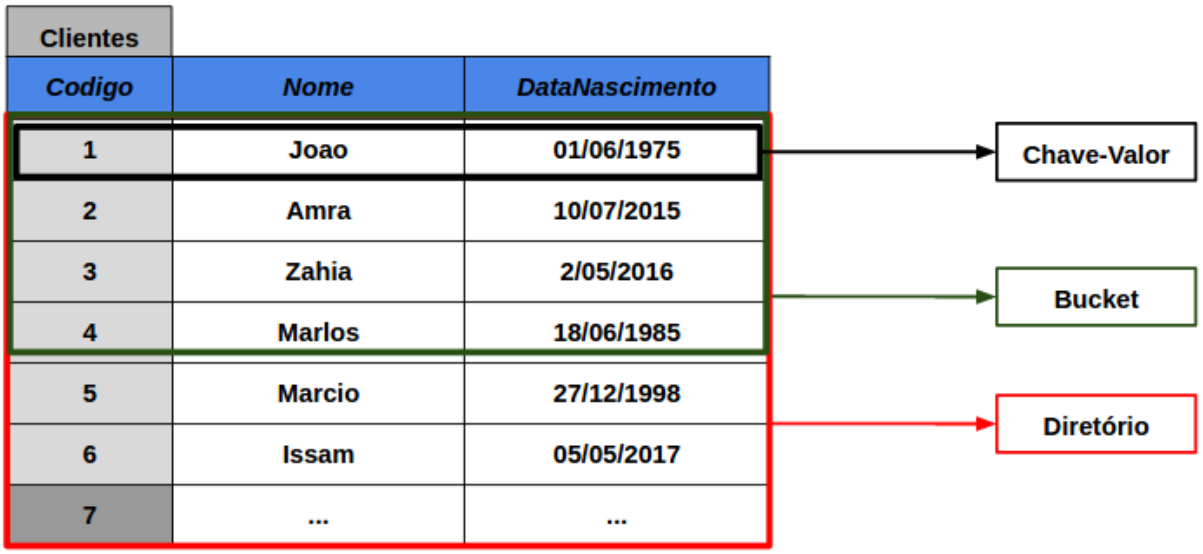

Figura 5.2: Mapeamento do modelo relacional para o modelo do ALOCS

5.2 Arquitetura

De acordo com a classificação dos SADs apresentada na subseção 2.2.3 do capítulo 2, o ALOCS possui uma arquitetura assimétrica. Esta arquitetura considera um conjunto de nodos (máquinas físicas), como ilustrado na Figura 5.3. Além de dar suporte ao armazenamento distribuído de dados, o sistema possui um gerenciador de metadados dedicado, responsável por manter *metadados* a respeito da associação dos intervalos de chaves aos *buckets* e informações sobre a hierarquia de servidores, diretórios e *buckets*. Assim, os nodos que compõem o repositório distribuído podem ser apenas servidores de dados (como os nodos na parte inferior da ilustração) ou servidores de dados e metadados (como os nodos na parte superior). Os metadados são replicados em todos os servidores de metadados. Os servidores de dados tem como objetivo dar suporte ao armazenamento de um grande volume de dados (escalabilidade de armazenamento). Com a possibilidade dos nodos assumirem papéis distintos, é possível criar um conjunto maior de servidores de dados sem incorrer no custo de replicação dos metadados em todos eles. O custo da replicação pode ser proibitivo para aplicações com um grande volume de escritas. Assim, é possível ajustar a quantidade de servidores de metadados de acordo com o volume de escritas e leituras de cada aplicação.

Os componentes do sistema responsáveis por prover tais funcionalidades são os módulos de controle, módulos de armazenamento e módulos de metadados, que são detalhados na sequência.

5.2.1 Módulo de controle

O módulo de controle é responsável por receber as requisições dos programas de aplicação e fazer a interface com os módulos de metadados e de armazenamento. Por exemplo, considere que a aplicação submeta a operação `put(k,v)`. O módulo de controle envia ao módulo de metadados a operação `getPath(k)`, que retorna o caminho servidor/diretório/bucket (`s/d/b`) no qual a chave `k` deve ser armazenada. Com posse destas informações, a operação `put(s/d/b/k, v)` é então enviada para o módulo de armazenamento. Caso nenhum *buc-*

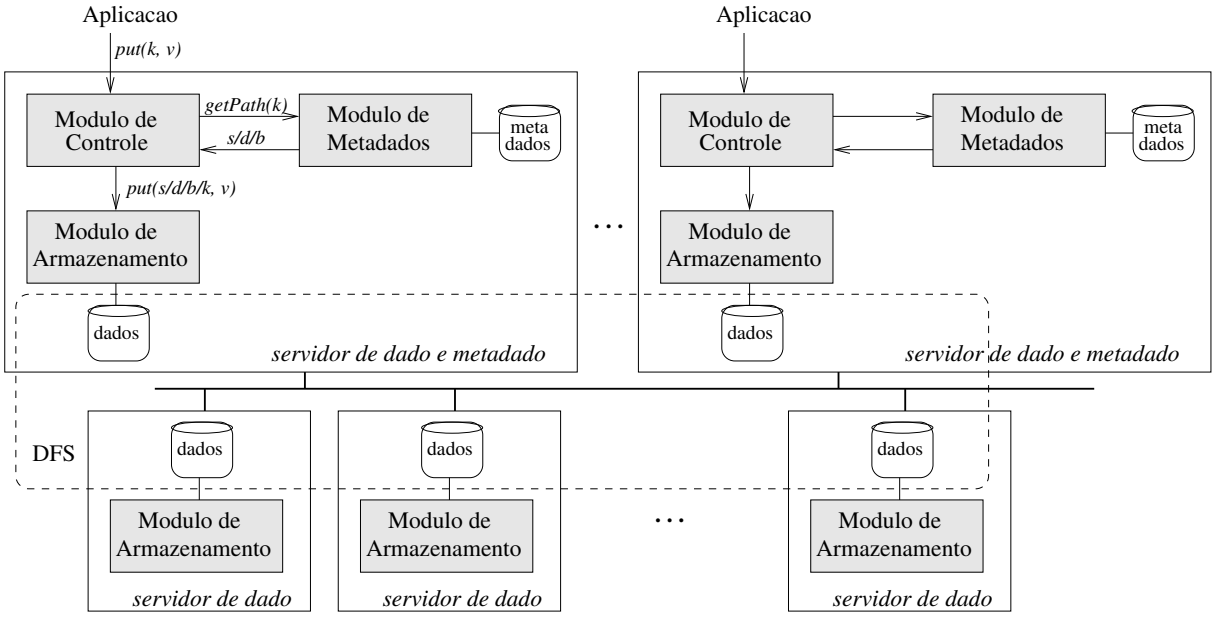

Figura 5.3: Arquitetura do Repositório de dados

ket possuir a chave passada, o módulo de metadados retorna uma mensagem informando que a chave não pode ser usada, pois não pertence a nenhum *bucket*.

5.2.2 Módulo de armazenamento

A principal função do módulo de armazenamento é a interface entre o módulo de controle e o sistema de armazenamento distribuído de dados, fazendo o mapeamento entre o modelo implementado pelo ALOCS e o modelo de armazenamento físico. Em sua implementação atual, é adotado um SAD no qual cada *bucket* é armazenado em um arquivo. Assim, a recuperação de um par chave-valor envolve a leitura do arquivo que armazena o *bucket* e a extração do par chave-valor de interesse. O módulo de armazenamento é também responsável pelo gerenciamento do Cache.

O ALOCS mantém em Cache os *buckets* acessados pela aplicação durante o processamento, para reduzir o tempo de resposta das consultas que requisitarem chaves de *buckets* que foram acessados anteriormente. O tamanho do Cache, pode ser configurado de acordo com a carga de trabalho da aplicação. Atualmente o ALOCS está configurado para manter 500 *buckets* em Cache. Quando um *bucket* é requisitado, a interface do sistema de armazenamento inicia a busca no Cache, se não for encontrado a interface copia o *bucket*

do sistema de armazenamento para uma linha de Cache disponível. Se não houver uma linha de Cache disponível, a interface utiliza uma política de substituição baseada no algoritmo *Least Recently Used* (LRU) [50]. Assim, sempre que uma chave é requisitada, cabe ao módulo de armazenamento verificar se o *bucket* correspondente já encontra-se no Cache para evitar uma nova requisição ao SAD. O Cache utilizado funciona apenas para operações de leitura. No caso de operações de escrita, o par chave-valor é atualizado primeiro no Cache e depois em disco.

5.2.3 Módulo de metadados

O módulo de metadados é responsável pelo armazenamento, distribuição e gerenciamento de informações sobre as localidades de *buckets* encontrados no módulo de armazenamento. Estas informações são coletadas durante a inserção de dados no sistema, visto que o usuário especifica o local onde deseja armazenar seus dados; isto é, passando o *bucket*, o diretório e o servidor. No módulo de metadados, cada *bucket* é armazenado com o intervalo de chaves correspondentes aos valores armazenados nele. Além disso, os metadados mantém também informações sobre o seu modelo hierárquico de armazenamento. Assim, o módulo de metadados possui 2 subcomponentes (Figura 5.4):

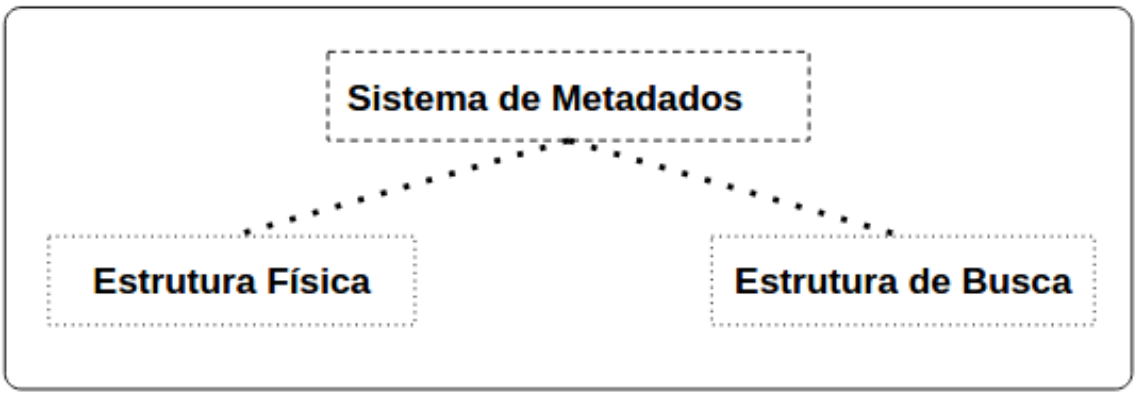

Figura 5.4: Sistema de Metadados

- **Estrutura Física**: ilustrada na Figura 5.5; ela é responsável pelo armazenamento do mapeamento físico de todos dispositivos de armazenamento usados no sistema,

isto é, todos os servidores com seus diretórios, e estes com todos os seus *buckets*.

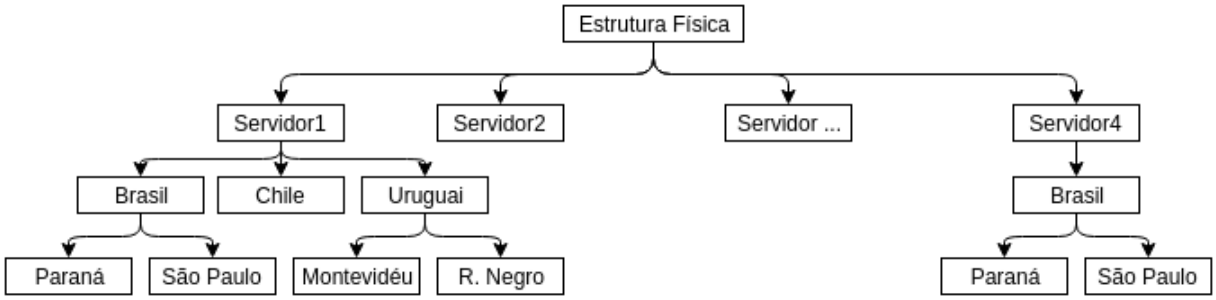

Figura 5.5: Estrutura Físca do Módulo de Metadados

- **Estrutura de Busca**: corresponde a uma estrutura de indexação para facilitar a busca de chaves. Ela é uma adaptação da árvore de intervalos, na qual os nodos correspondem aos intervalos de chaves [60]. Cada nodo da árvore mantém, além do intervalo, o valor máximo dentre todos os intervalos em sua subárvore e a localização do extitbucket responsável pelo seu intervalo de chaves (*/servidor/diretório/bucket*). Um exemplo desta árvore é apresentada na Figura 5.6. O sistema garante que não há sobreposição entre os intervalos no momento da criação dos *buckets*. Assim, a busca na árvore é idêntica à busca em uma árvore binária, considerando apenas o valor inicial de cada intervalo armazenado em cada nodo.

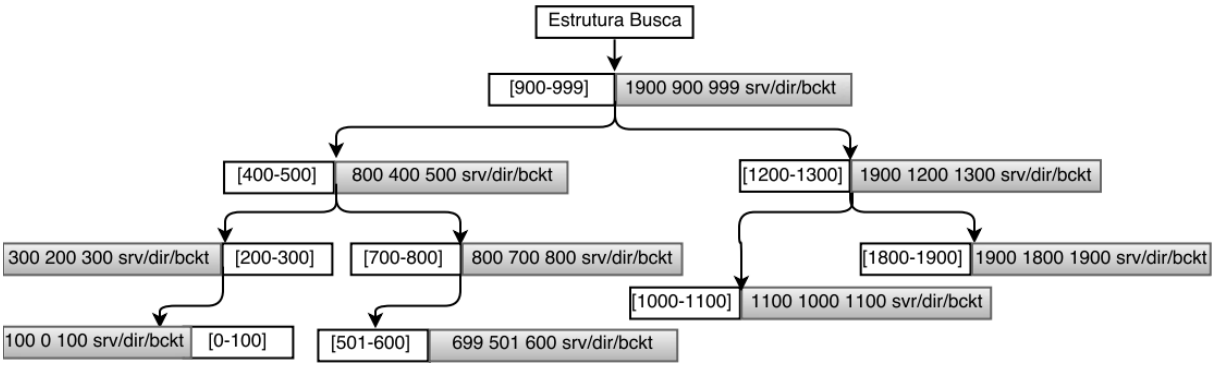

Figura 5.6: Estrutura de Busca do Módulo de Metadados

Na arquitetura, os dados a serem persistidos no sistema são alocados em formato pares chave-valor agrupados em *bucket*, cuja localidade é controlada pela aplicação mediante o Sistema de Metadados. Esta abordagem prove regras para o fluxo de dados entre modelos lógico e físico, e permite também que seja possível implementar vários tipos de aplicações

que podem utilizar a solução como repositório de armazenamento, visto que o modelo chave-valor possui uma grande flexibilidade. Seguem detalhes sobre a implementação do ALOCS.

5.3 Implementação

O ALOCS foi implementado utilizando o SAD Ceph[1] para a implementação do módulo de armazenamento e o *Zookeeper*[2] para o módulo de metadados. Ele foi projetado de forma modular, com uma interface bem definida entre os três módulos que o compõem. Dessa forma, outros sistemas de armazenamento e de controle de metadados podem ser considerados no desenvolvimento de versões futuras do ALOCS.

O módulo de armazenamento herda do Ceph propriedades como escalabilidade, desempenho e disponibilidade de dados. A unidade de armazenamento do Ceph é o objeto. Estes são agrupados em *Placement Group* (PG) como mencionado na seção 4.1.3. Cada PG é associado a uma lista de OSDs, sendo um primário, e os demais réplicas conforme ilustrado pela Figura 5.7. As tarefas relacionadas ao gerenciamento de dados, como alocação, replicação e recuperação, são distribuídas entre os dispositivos disponíveis, permitindo paralelização das operações com maior eficiência [9].

Para implementar a hierarquia do ALOCS no Ceph, cada *bucket* é mapeado para um objeto do Ceph e cada diretório é mapeado para um PG, associando-o aos OSDs que correspondem aos nodos do servidor ALOCS no qual o diretório deve ser armazenado. Assim, caso o diretório seja replicado, basta associar um novo OSD ao grupo de alocação correspondente.

O objeto que corresponde ao *bucket* contém um cabeçalho onde são armazenadas informações para a localização dos pares chave-valor, denominado *slot*, e um mapa de bits para o controle dos slots livres. Nesta implementação o tamanho do *bucket* foi limitado a 64 KBytes.

O SAD Ceph foi escolhido por ser distribuído e possuir um algoritmo para distribuição

[1] http://ceph.com
[2] http://zookeeper.apache.org

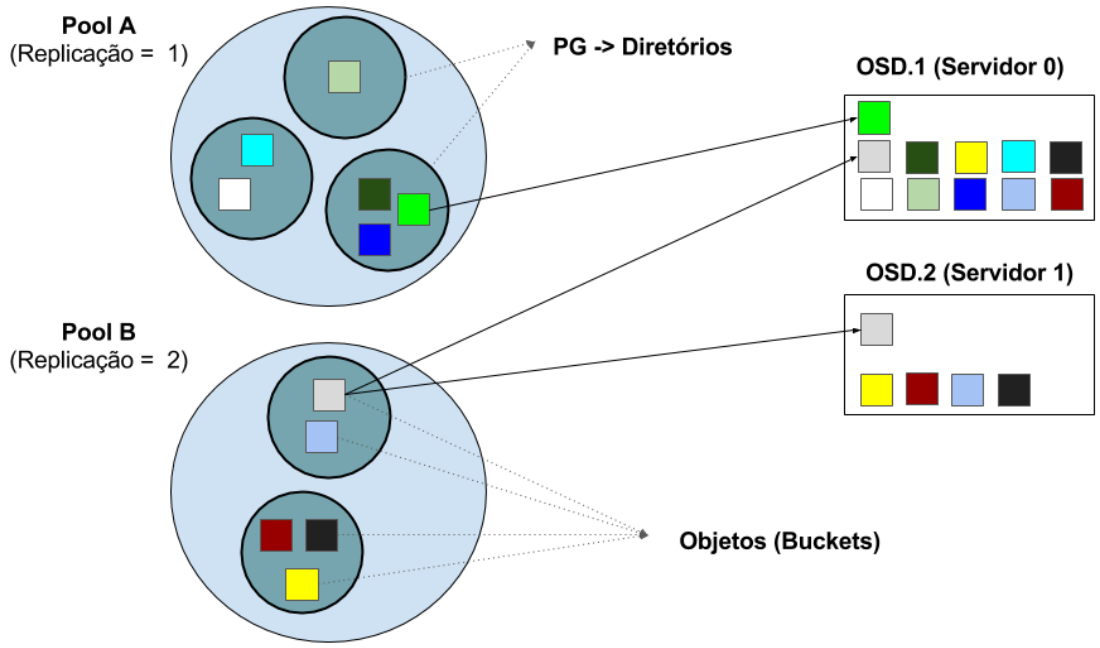

Figura 5.7: Mapeamento do modelo de armazenamento do Ceph para o modelo do ALOCS

de dados denominado CRUSH [82], que é responsável por alocar e replicar os dados em um *cluster* de servidores. O CRUSH faz a distribuição a partir de um conjunto de regras, que indicam onde um determinado objeto e suas réplicas devem ser alocados, e no mapa do *cluster*, que permite explorar as características do ambiente físico. Esta abordagem concede ao Ceph flexibilidade, o controle na alocação (controle de localidade) e replicação de dados. Este controle de alocação é implementado por meio da Librados [3] , que é uma interface de programação de aplicações (API) do Ceph na linguagem C. A cada inserção de um determidado de dado são passados como parâmetros o servidor, o diretório e o *bucket* onde o será armazenado.

O módulo de metadados é implementado usando o serviço de coordenação de processos *Apache Zookeeper* e consequentemente herda propriedades como a alta disponibilidade, escalabilidade e confiabilidade [39]. Os metadados são armazenados sob forma de *znodes*, que são organizadas de acordo com um espaço de nomes hierárquico, semelhante às estruturas de dados de diretórios [71].

O módulo de metadados foi implementado na linguagem Java. A interação deste módulo com o *Zookeeper* é feita usando a versão Java do *Zookeeper* API. O módulo

[3] http://docs.ceph.com/docs/master/rados/api/librados/

de controle, implementado em C, interage com o módulo de metadados através de uma interface desenvolvida utilizando o *Java Native Interface* (JNI). Esta primeira versão de ALOCS é para as aplicações 64 bits.

5.4 Princípios de funcionamento

Para que uma aplicação cliente e os módulos de armazenamento e de metadados operem de maneira coordenada, o módulo de controle gerencia a interação entre eles através de três interfaces que expõem operações específicas que são executadas em conjunto. De forma geral, o fluxo de execução é: a aplicação requisita operações ao módulo de controle, que por sua vez solicita informações sobre a localidades dos dados ao módulo de metadados, e encaminha as operações solicitadas ao módulo de armazenamento, baseado na localidade fornecida pelos metadados.

5.4.1 Inserção e recuperação de dados

A Figura 5.8 ilustra o fluxo de inserção de um par chave-valor ressaltando as principais interfaces. No passo 1, a aplicação executa a operação *put_pair(k, v)*, onde k representa a chave e v o valor a ser persistido. Não é necessário especificar o *Bucket*, pois o módulo de controle obtém a sua localização por meio da operação *get(k)*, que recupera através de intervalos de chaves o servidor (S), diretório (D) e *Bucket* (B) em que a chave especificada será alocada. Baseado na localização obtida, o módulo de controle solicita ao módulo de Armazenamento a adição do par chave-valor no passo 3 mediante a operação *put_pair(S,D,B,K,V)*.

Após a inclusão do par chave-valor, o módulo de armazenamento retorna uma mensagem para o módulo de controle confirmando a situação da operação; em seguida no passo 4, é retornado para a aplicação o resultado final. Se a operação *get(K)* não encontrar no módulo de metadados o intervalo de chaves correspondente a chave requisitada, é retornado um erro para a aplicação, que pede ao usuário se quer inicializar um novo *Bucket* com um intervalo de chave que contenha a chave usada.

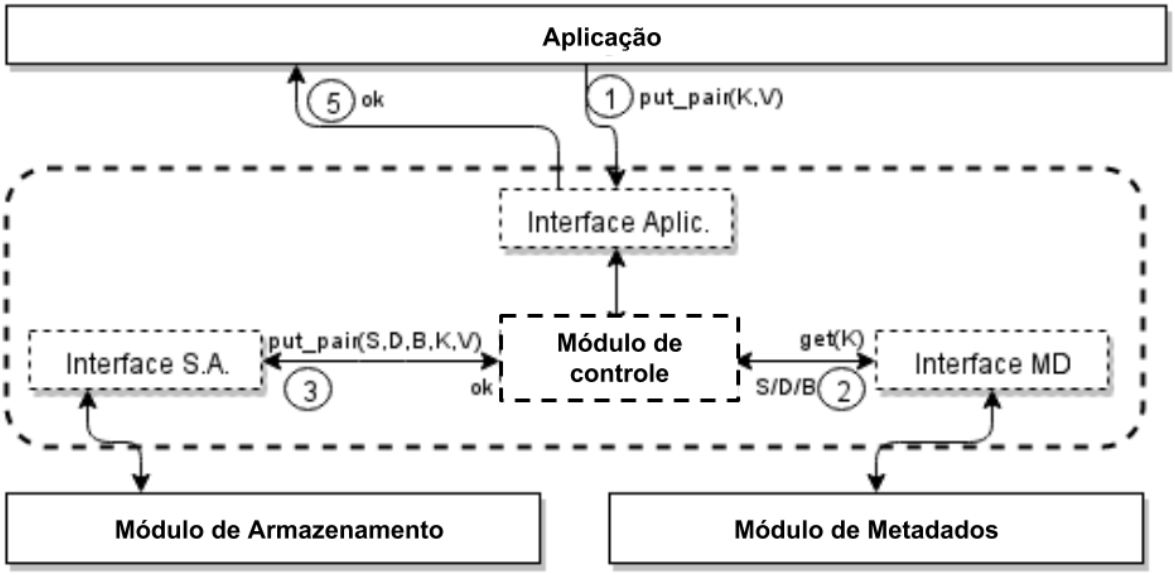

Figura 5.8: Interação entre as interfaces para inserir um par chave-valor

A Figura 5.9 ilustra a interação entre as interfaces durante a leitura de um par chave-valor. No passo 1, a aplicação executa a operação *get_pair(K,V)*, não é necessário especificar o *Bucket*, pois, o módulo de controle obtém a localização da chave através o Sistema de metadados no passo 2 por meio da operação *get(K)* que retorna o servidor (S), diretório (D) e *Bucket* (B), isto é, a localização do intervalo de chaves onde a chave K foi encontrada.

Após o módulo de controle receber a localização da chave, no passo 3 é solicitado ao Sistema de Armazenamento o *Bucket* em que o par chave-valor está localizado, por meio da operação *getBucket(S,D,B)*, que retorna o *Bucket* baseado na localização fornecida pelo módulo de Metadados. Com o *Bucket*, o módulo de controle extrai o par chave-valor solicitado no passo 4 e retorna para a interface da aplicação, e esta por sua vez retorna o par extraído para a aplicação no passo 5. Se a operação *get(K)* não encontrar o intervalo de chaves correspondente a chave requisitada, é retornada uma mensagem de erro para a aplicação. A seguir uma breve descrição das interfaces e suas operações.

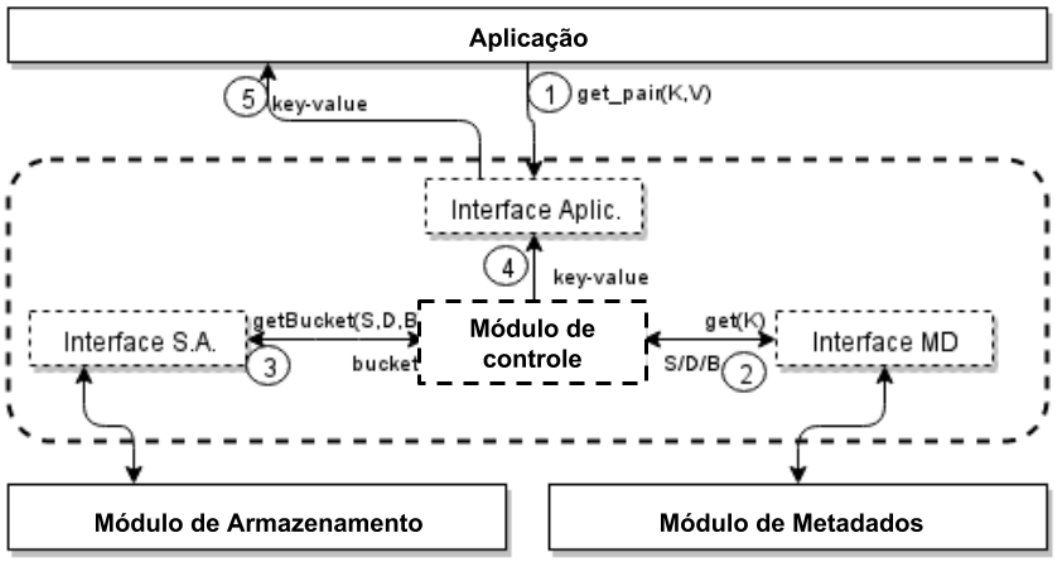

Figura 5.9: Interação entre as interfaces para recuperar um par chave-valor

5.4.2 Funções de manipulação de dados entre as interfaces

5.4.2.1 Interface para a aplicação

Qualquer aplicação que deseja utilizar o sistema como infraestrutura de armazenamento faz uso desta interface. Ela permite que sejam realizadas operações de criação, remoção e busca dos componentes citados na seção sobre o modelo de armazenamento. As operações que esta interface disponibiliza são:

- ***clean():*** Requisita a remoção de todos os *Buckets* que estiverem vazios.
- ***creat_bucket(dirName,idBucket,iniKey,finKey):*** Requisita a criação de um *Bucket* em Diretório especificado nos parâmetros de entrada, em conjunto com o identificador do *Bucket*, e o intervalo de chaves que será armazenado.
- ***create_dir(dirName,srvName):*** Requisita a criação de um Diretório, no Servidor que foi especificado nos parâmetros de entrada.
- ***replicate_dir(dirName,srvName):*** Requisita a replicação de um Diretório, especificado como parâmetro de entrada, em conjunto com o Servidor de destino. Vale

ressaltar que não foi implementado o suporte à replicação nesta primeira versão de ALOCS.

- ***drop_ dir(dirName,srvName):*** Requisita a remoção de um Diretório, especificado nos parâmetros de entrada, em conjunto com o Servidor em que ele está alocado.

- ***dropALL_ dir(dirName):*** Requisita a remoção de um Diretório, e suas réplicas, especificado como parâmetro de entrada.

- ***drop_ bucket(idBucket):*** Requisita a remoção de um *Bucket* especificado nos parâmetros de entrada.

- ***put_ pair(key,value):*** Requisita a adição de um par chave-valor especificado nos parâmetro de entrada. O *Bucket* é identificado pelo sistema de Metadados baseado no intervalo de chaves.

- ***get_ pair(key):*** Requisita um par chave-valor identificado por uma chave especificada como parâmetro de entrada. O *Bucket* é identificado pelo sistema de Metadados baseado no intervalo de chaves.

- ***rem_ pair(key):*** Requisita a remoção de um par chave-valor identificado por uma chave especificada como parâmetro de entrada. O *Bucket* é identificado pelo sistema de Metadados baseado no intervalo de chaves.

5.4.2.2 Interface para o módulo de armazenamento

Esta interface permite que qualquer sistema de Armazenamento seja integrado ao módulo de controle, sem que haja a necessidade de alterar a aplicação, desde que implemente esta interface. Através desta interface, as operações requisitadas pela aplicação cliente são executadas no sistema de armazenamento. Até o momento, três sistema foram testados para implementar o sistema de armazenamento: dois SADs, PVFS [32] e Ceph (no caso desta dissertação) [81], e um repositório chave-valor, Scalaris [66]. O ALOCS foi planejado para que seja possível, alterar o sistema de armazenamento sem que haja a necessidade

de alterar a aplicação. Seguem as operações de manipulação de dados disponíveis nesta interface.

- ***create_ bucket(srvName,dirName,idBucket):*** Cria um *bucket* em Diretório especificado nos parâmetros de entrada, em conjunto com, o identificador do *Bucket*, e Servidor. A versão atual do ALOCS não permite que seja possível reconfigurar os buckets criados.

- ***create_ dir(dirName,srvName):*** Cria um Diretório em Servidor especificado por parâmetro.

- ***copy_ dir(dirName,srvName1,srvName2):*** Copia um Diretório de um Servidor para outro Servidor, ambos especificados por parâmetros.

- ***drop_ dir(dirName,srvName):*** Remove um Diretório de um Servidor, ambos especificados por parâmetros.

- ***drop_ bucket(idBucket,dirName,srvName):*** Remove um *Bucket*, do Servidor e Diretório especificados por parâmetros em conjunto com o identificador do *bucket*.

- ***getBucket(srvName,dirName,idBucket):*** Retorna um *Bucket*, de um Diretório e Servidor especificados nos parâmetros de entrada.

- ***is_ Empty(idBucket,dirName,srvName):*** Verifica se um *Bucket*, especificado por parâmetro, está vazio, são indicados também o Diretório e Servidor.

- ***is_ Empty(dirName,srvName):*** Verifica se um Diretório, especificado por parâmetro, está vazio, deve ser indicado também o Servidor.

- ***put_ pair(srvName,dirName,idBucket,key,value):*** Adiciona um par chave-valor em *Bucket* especificado por parâmetro, em conjunto com a chave e o valor.

- ***rem_ pair(srvName,dirName,idBucket,key):*** Remove um par chave-valor do *Bucket* correspondente ao identificador passado como parâmetro de entrada, é necessário fornecer também o Servidor e Diretório.

5.4.2.3 Interface para módulo de metadados

A interface para o módulo de metadados permite que o módulo de controle solicite informações, sobre dados alocados, ao sistema que gerencia e controla os metadados. As requisições que podem ser feitas são:

- ***drop_ bucket(idBucket,dirName):*** Remove dos metadados um *Bucket*, que foi removido por uma operação *drop_ bucket*. O identificador do *Bucket* e o Diretório devem ser especificados.

- ***drop_ dir(dirName,srvName):*** Remove dos metadados um Diretório, que foi removido por uma operação *drop_ dir*. O Diretório e o Servidor devem ser especificados.

- ***get(dirName):*** Retorna o Servidor, em que o Diretório especificado por parâmetro, deve estar alocado. Caso o Diretório relacionado tenha sido replicado, a função retorna uma lista com todos os Servidores associados.

- ***get(idBucket):*** Retorna o Servidor, e o Diretório em que o *Bucket* especificado por parâmetro deve estar alocado. Caso o Diretório relacionado tenha sido replicado, a função retorna uma lista com todos os Servidores associados.

- ***get(key):*** Retorna o Servidor, Diretório e *Bucket* em que o par chave-valor, cuja chave foi especificada por parâmetro, deve estar alocado. Caso o Diretório relacionado tenha sido replicado, a função retorna uma lista com todos os Servidores associados.

- ***getInterval(idBucket):*** Retorna o intervalo de chaves armazenado no *Bucket* passado como parâmetro.

- ***put_ bucket(idBucket,dirName,iniKey,finKey):*** Adiciona aos metadados um *Bucket* criado pela operação *create_ bucket*. Além do identificador do *Bucket* devem ser especificados, o Diretório e o intervalo de chaves.

- ***put_ dir(dirName,srvName):*** Adiciona aos metadados um Diretório criado pelas operações *create_ dir* ou t*copy_ dir*. Além do Diretório, deve ser especificado o Servidor.

No próximo capítulo são apresentados experimentos realizados para validar a abordagem proposta pelo ALOCS.

CAPÍTULO 6

EXPERIMENTOS E RESULTADOS

O capítulo anterior apresentou o ALOCS, um repositório chave-valor que permite que a aplicação usuária controle a localidade de seus dados. A sua arquitetura estratificada tem como base o sistema de arquivos distribuídos Ceph e um módulo de gerenciamento de metadados que utiliza o *Zookeeper*. O presente capítulo apresenta os experimentos realizados em prol de validar a abordagem proposta pelo ALOCS.

6.1 Ambiente

A fim de validar o modelo proposto neste trabalho, foi elaborado um ambiente de testes com 5 máquinas virtuais Debian 7.0.0, cada uma com 30GB de disco e 2GB de memória RAM. Para os experimentos, foi configurado um *cluster* com 2 servidores de armazenamento e com 3 servidores de metadados. Os módulos de controle e metadados do ALOCS são sempre executados em servidores que armazenam os metadados, a fim de evitar comunicações remotas para a obtenção da localização das chaves. Os servidores de metadados formam um *cluster* (*Zookeeper*) e os de dados um *cluster* Ceph.

Em cada servidor foi criado um único diretório, contendo um conjunto de buckets, cada um contendo até 100 pares chave-valor. A escolha desta quantidade de chaves foi feita aleatoriamente. Os valores correspondem aos arquivos de textos de 50 bytes. Embora o Ceph permita a replicação de dados, esta funcionalidade não foi utilizada na realização dos experimentos. Os testes foram repetidos 3 vezes e foram calculadas as médias. Embora existam vários benchmarks (Rados Bench [79], Cosbench [89] e Bonnie++ [1] entre outros), nenhum se adequou para ser usado especificamente nestes experimentos.

[1] http://www.coker.com.au/bonnie++/

6.2 Experimentos e análise de resultados

Tendo em vista que o modelo de controle de localidade de dados proposto nesta dissertação é baseado na manutenção de metadados que relacionam pares chave-valor ao seu armazenamento físico, nesta seção são apresentados dois experimentos que tem como objetivo determinar o custo de acesso aos metadados. O controle de localidade permitiu a coalocação de dados em *buckets*. O terceiro experimento descreve o impacto desta coalocação sobre o tempo de acesso aos dados e o último experimento avalia a escalabilidade do *cluster* de metadados.

Experimento 1: Custo da gravação de metadados

Para determinar o custo da gravação dos metadados, neste experimento é executada uma sequência de comandos que geram *buckets* de intervalos distintos em um mesmo diretório (operação `createBucket(dir, idBucket, chaveIni, chaveFim)`). Cada comando requer duas escritas nos metadados: uma na estrutura física, para inserir o *bucket* `idBucket` como filho de `dir` e outra escrita na estrutura de busca. Além disso, um objeto é escrito no SAD Ceph, contendo apenas o cabeçalho para manter as informações sobre os objetos que serão posteriormente inseridos no *bucket*. A sequência de comandos foi executada em três configurações: com um único servidor de metadados e comandos ordenados pelo valor da chave inicial do intervalo; com um único servidor de metadados e comandos em ordem aleatória; e com três servidores de metadados e comandos em ordem aleatória. Nesta última configuração, cada comando requer gravação nas três réplicas dos metadados. Os tempos de execução (em milissegundos) de sequências de 5 a 50 operações são apresentados na Figura 6.1.

Pode ser observado que a inicialização sequencial teve custo mais alto do que as aleatórias e consiste no cenário de pior caso. Isso se deve ao fato da altura da árvore da estrutura de busca ter crescimento linear com esta ordem de inserção, transformando-a em uma lista. Nesta configuração, para 50 operações o tempo total de execução foi de 4650 ms (93 ms por operação). No final da execução, a árvore resultante possui altura 50, o que explica o seu alto tempo de execução. Já para a inicialização aleatória, o tempo total de execução para 50 operações foi de 349 ms (7 ms por operação), uma redução de

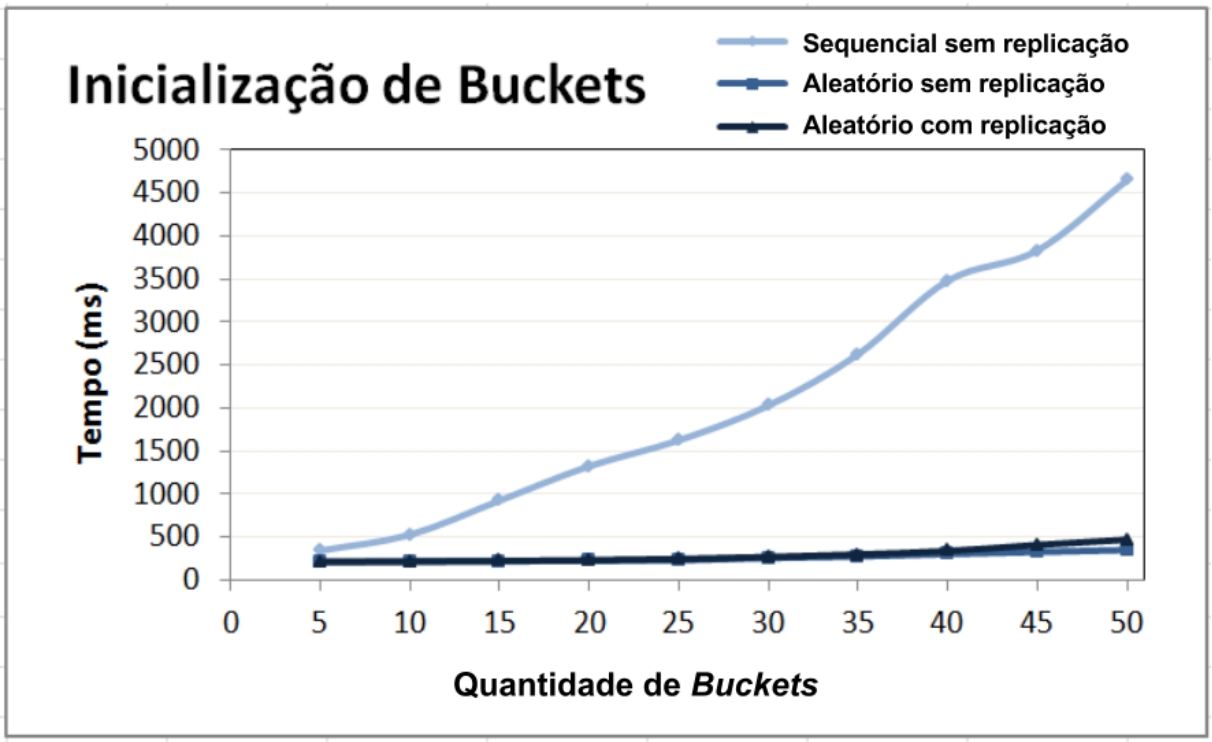

Figura 6.1: Inicialização de Buckets em Milisegundos

92.5% com relação ao tempo de inicialização sequencial. No final da sequência o altura da árvore era de 6. Isso mostra o grande impacto que o balanceamento da árvore tem sobre o tempo de geração dos *buckets*. Um ponto importante a ser observado é que o tempo total de execução para a configuração sem replicação possui apenas uma redução de 28% em relação ao tempo com replicação. Isso mostra que em uma aplicação com muitos operações de leitura é possível replicar os metadados em diversos nodos para obter escalabilidade de processamento, sem que isto tenha um grande impacto sobre o tempo de execução das operações de escrita.

Experimento 2: Custo da leitura dos metadados

Neste segundo experimento foi avaliado o tempo de leitura aos metadados. Para isso, foram executadas sequências de operações de inserção de pares chave-valor nos buckets previamente criados, como relatados no Experimento 1. Todas as operações foram submetidas para um único módulo de controle do ALOCS. A Figura 6.2 apresenta o tempo total de execução (em milissegundos) de sequências de 10 a 100 operações de gravação `put(k, v)`. Observe que cada operação requer uma consulta aos metadados, para obter a localização do bucket e a gravação do par chave-valor no SAD Ceph. No gráfico,

o tempo de gravação no Ceph é apresentado na barra “Sem metadados”, enquanto as demais barras apresentam o tempo total de execução da sequência com os metadados gerados randomicamente e sequencialmente.

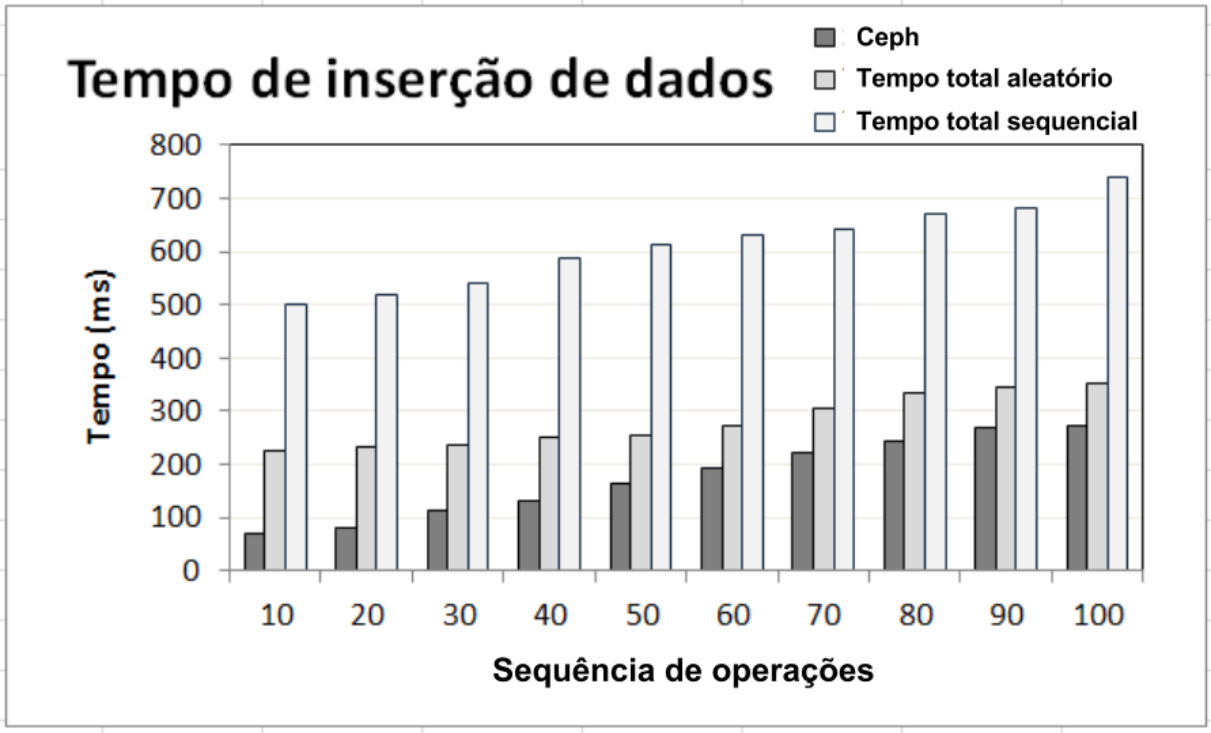

Figura 6.2: Tempo de inserção de dados

Para uma sequência de 50 operações, o tempo de gravação no SAD foi de 163 ms. O tempo total quando os metadados foram gerado randomicamente foi de 253 ms, o que resulta em uma média de 6 ms por operação de gravação. Quando os metadados foram gerados sequencialmente, o tempo de acesso aos metadados é bem maior, com 13 ms por operação de gravação. Isso era esperado, tendo em vista a altura da árvore de busca resultante com os metadados inseridos desta forma. Embora o custo de acesso aos metadados pareça alto para uma sequência pequena de operações, o *overhead* sobre o tempo de execução total para sequências maiores não é tão grande, como pode ser observado nos resultados com 100 operações.

Experimento 3: Impacto da coalocação de dados

O grande benefício de controlar a localidade de dados é o agrupamento dos mesmos em *buckets* específicos determinados pela aplicação, isto é a coalocação de dados. Para avaliar o impacto da coalocação no acesso a um conjunto de dados, neste experimento

foram executadas sequências de operações de leitura (`get(k)`) nas quais há um porcentual distinto de coalocação das chaves requisitadas pelas operações. Foram considerados os porcentuais de 0%, 50% e 100%. Ou seja, foram geradas sequências nas quais cada par chave-valor está armazenado em *buckets* distintos (0% de coalocação), com metade dos pares coalocados com um outro par da sequência (50% de coalocação) e sequências com todos os pares alocados em um único *bucket* (100% de coalocação). As sequências tem tamanho variando de 10 a 100 e a Figura 6.3 apresenta os resultados.

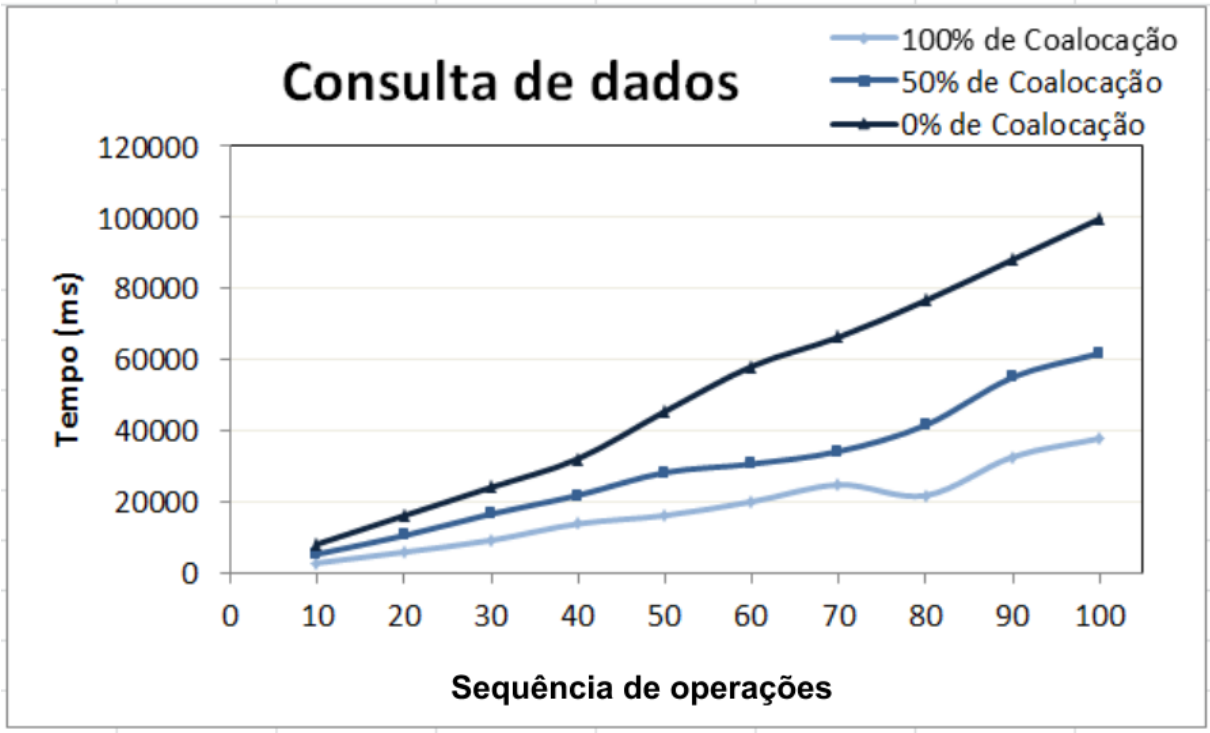

Figura 6.3: Consulta de sequência de dados

Como esperado, o tempo de consultas de dados com 100% de coalocação é menor do que os demais. Isto ocorre porque como o *bucket* é a unidade de transmissão do ALOCS, apenas o primeiro acesso a um par chave-valor de um *bucket* requer uma leitura em um nó remoto. As leituras subsequentes a pares no mesmo *bucket* são executadas localmente no Cache do sistema. Para 10 consultas, o tempo médio de execução de cada operação `get(k)` é de 269 ms para dados coalocados em um único *bucket* (100% de coalocação). Este tempo aumenta de 92% para dados com 50% de coalocação e 197,7% para dados sem coalocação. Para sequências de 100 operações, com 100% de coalocação o tempo médio de cada operação foi de 367 ms. Este tempo aumenta para 617 ms para 50% de coalocação

(68% maior que o tempo com 100% de coalocação) e 996 ms para 0% de coalocação (171% maior). Estes resultados mostram o enorme impacto que a coalocação tem sobre o acesso de um conjunto de dados alocados no mesmo *bucket*. O controle de localidade de dados propocionou esta coalocação de dados.

Experimento 4: Impacto da escalabilidade do módulo de metadados

O módulo de metadados é um componente importante do sistema ALOCS. A utilização de um único servidor de metadados o torna um ponto único de falha. Em contrapartida, o uso de muitos servidores pode criar um *overhead* no ALOCS. O experimento 4 é muito similar ao experimento 2. Os metadados foram inseridos randomicamente. Operações de inserção de dados foram executados em ambiente com 1, 3 e 5 servidores de metadados. Como mencionado na subção 4.2.3, o número ímpar de servidores de metadados é recomendado para garantir a maioria do quorum e assim a disponibilidade de serviço. Os dados inseridos foram os pares alocados em *buckets* previamente criados. Foram realizadas gravações de dados em sequências de 10 a 100 e os resultados são apresentados na Figura 6.4.

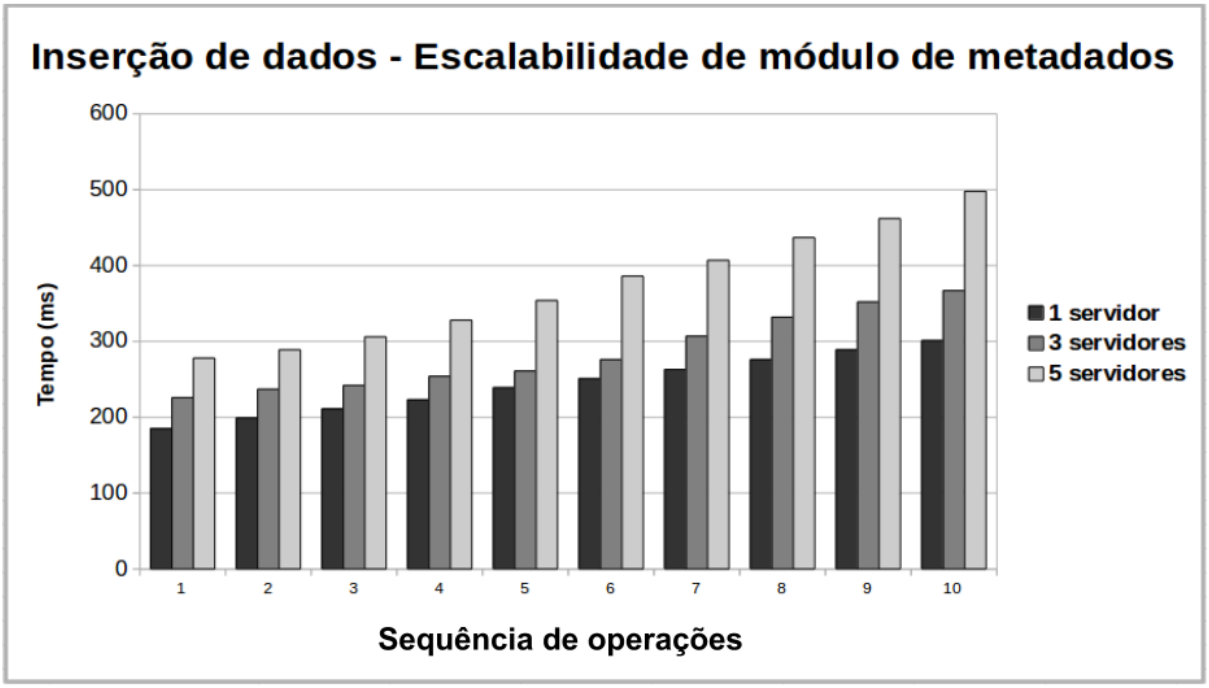

Figura 6.4: Inserção de dados usando a variação do número de servidores de metadados

Como pode se observar, o tempo de inserção de dados com o módulo de metadados usando apenas 1 servidor é menor que os outros 2 casos (3 e 5 servidores). O tempo médio

de inserção de 10 pares de chaves é de aproximadamente 18 ms quando existe apenas um servidor de metadados. A média é de 23 ms para 3 servidores de metadados e 28 ms para 5. Esse aumento é decorrente ao custo acarretado pela replicação dos metadados em todos os servidores membros do *cluster* de metadados.

CAPÍTULO 7

CONCLUSÃO

7.1 Conclusão e trabalhos futuros

Este capítulo descreve as conclusões obtidas considerando os resultados alcançados nos experimentos e relacionando os com a proposta almejada por esta dissertação. Além disso, discute-se algumas questões pendentes a serem abordadas nos trabalhos futuros. Vale ressaltar também que esta dissertação produziu o artigo intitulado "**Um repositório Chave-Valor com Controle de Localidade**", que foi aceito no 31 Simpósio Brasileiro de Banco de Dados (SBBD2016) [1].

7.1.1 Conclusão

Este trabalho propôs o ALOCS, um repositório de armazenamento distribuído chave-valor com controle de localidade de dados. O ALOCS baseia-se em modelo de armazenamento hierárquico e nos metadados para gerenciar a localidade de dados. Esta proposta possibilita à aplicação a alocação de dados em determinado servidor no repositório distribuído. A implementação da proposta usa o Ceph para o armazenamento distribuído e o *Zookeeper* para gerenciar os metadados.

As soluções de armazenamento distribuído atuais possuem pouco ou nenhum controle sobre a alocação das informações nos servidores. Algumas implementam mecanismos de distribuição uniforme dos dados ou baseados em ordenação lexicográfica. O modelo proposto neste trabalho proporciona o controle de localidade de dados. Com esta propriedade é possível otimizar aplicações distribuídas globalmente, coalocando os dados usados em conjunto e aproximando os mesmos de suas aplicações usuárias. Isto evita o acesso a múltiplos servidores e diminui a incidência de transições distribuídas. Os experimentos comprovaram que o custo de controle de localidade é baixo. Isto viabiliza a solução

[1] http://sbbd2016.fpc.ufba.br

proposta e seu modelo de armazenamento.

O primeiro objetivo dos experimentos de validação do ALOCS era avaliar o custo acarretado pelo controle de localidade. Por este motivo foi realizada a comparação com o próprio Ceph. O segundo objetivo foi demostrar o benefício de alocar um conjunto de dados em um mesmo *bucket*. Como comprovado pelos resultados positivos obtidos, o grande ganho do ALOCS é a redução das requisições feitas durante as consultas de dados alocados em um mesmo *bucket*, uma vez que é preciso apenas uma consulta para trazer todos os dados do *bucket*. As leituras dos demais dados são executadas localmente no Cache do sistema. Vale ressaltar que os resultados não consideram o tempo de transmissão dos dados, nem da rede, pois foram usadas máquinas virtuais dentro da mesma máquina.

7.1.2 Trabalhos futuros

Como trabalhos futuros, planeja-se estudar e implementar abordagens utilizando outros sistemas de arquivos distribuídos. É nesta perspectiva que iniciou-se um trabalho baseado no PVFS como módulo de armazenamento. Comparações entre as duas versões de ALOCS (Ceph e PVFS), e possivelmente com algumas soluções que não tratam de controle de localidade são previstas.

Pretende-se também realizar os experimentos em ambiente em escala muito maior para avaliar o impacto da escalabilidade dos servidores de armazenamentos e metadados no desempenho das inserções e consultas de dados. Além da replicação, a otimização da árvore de busca adotada pelo módulo de metadados também deverá ser tratada para mantê-la balanceada e assim reduzir seu impacto na inicialização e tempo de leitura dos *buckets*. Por fim, pretende-se redefinir e otimizar a política de atualização de Cache.

BIBLIOGRAFIA

[1] Daniel J Abadi. Consistency tradeoffs in modern distributed database system design. *Computer-IEEE Computer Magazine*, 45(2), 2012.

[2] Divyakant Agrawal, Amr El Abbadi, Shyam Antony, e Sudipto Das. Data management challenges in cloud computing infrastructures. *Databases in Networked Information Systems*. Springer, 2010.

[3] Peter A Alsberg e John D Day. A principle for resilient sharing of distributed resources. *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976.

[4] J Chris Anderson, Jan Lehnardt, e Noah Slater. *CouchDB: the definitive guide*. "O'Reilly Media, Inc.", 2010.

[5] Stephanos Androutsellis-Theotokis e Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM computing surveys (CSUR)*, 36(4), 2004.

[6] Renzo Angles e Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1), 2008.

[7] Davi Einstein Melges Arnaut. Phoenis. 2011.

[8] Davi EM Arnaut, Rebeca Schroeder, e Carmem S Hara. Phoenix: A relational storage component for the cloud. *IEEE Cloud*. IEEE, 2011.

[9] Alain Azagury, Vladimir Dreizin, Michael Factor, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, Julian Satran, Ami Tavory, e Lena Yerushalmi. Towards an object store. *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*. IEEE, 2003.

[10] Khairina Abu Bakar, Mohd Hafiz Md Shaharill, e Mohiuddin Ahmed. Performance evaluation of a clustered memcache. *Information and Communication Technology for the Muslim World (ICT4M), 2010 International Conference on*. IEEE, 2010.

[11] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.

[12] Josh Cates. *Robust and efficient data management for a distributed hash table*. Tese de Doutorado, Massachusetts Institute of Technology, 2003.

[13] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Record*, 39(4), 2011.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, e Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 2008.

[15] Kristina Chodorow. *MongoDB: the definitive guide*. "O'Reilly Media, Inc.", 2013.

[16] James C Corbett, Jeffrey Dean, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3), 2013.

[17] George F Coulouris, Jean Dollimore, e Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[18] Alex Davies e Alessandro Orsaria. Scale out with glusterfs. *Linux Journal*, 2013(235), 2013.

[19] Carla Amaral de S. Rodrigues, Júlia Ferreira de Almeida, Vanessa Braganholo, e Marta Mattoso. Consulta a bases xml distribuídas em p2p. *SBBD - Sessão de Demos*, 2009.

[20] Benjamin Depardon, Gaël Le Mahec, e Cyril Séguin. Analysis of six distributed file systems. 2013.

[21] S Donovan, G Huizenga, AJ Hutton, CC Ross, MK Petersen, e P Schwan. Lustre: Building a file system for 1000-node clusters. *Proceedings of the Linux Symposium*, 2003.

[22] Thibault Dory, Boris Mejías, PV Roy, e Nam-Luc Tran. Measuring elasticity for cloud databases. *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*. Citeseer, 2011.

[23] Mohamed Y Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, e John McPherson. Cohadoop: flexible data placement and its exploitation in hadoop. *Proceedings of the VLDB Endowment*, 4(9), 2011.

[24] Borivoje Furht e Armando Escalante. *Handbook of cloud computing*, volume 3. Springer, 2010.

[25] Santhosh Kumar Gajendran. A survey on nosql databases. *University of Illinois*, 2012.

[26] John Gantz e David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2007, 2012.

[27] Sanjay Ghemawat, Howard Gobioff, e Shun-Tak Leung. The google file system. *ACM SIGOPS operating systems review*, volume 37. ACM, 2003.

[28] Ali Ghodsi. *Distributed k-ary system: Algorithms for distributed hash tables*. Tese de Doutorado, The Royal Institute of Technology, 2006.

[29] Ali Ghodsi, Luc Onana Alima, e Seif Haridi. Symmetric replication for structured peer-to-peer systems. *Databases, Information Systems, and Peer-to-Peer Computing*. Springer, 2007.

[30] Nikolaos Giannadakis, Anthony Rowe, Moustafa Ghanem, e Yi-ke Guo. Infogrid: providing information integration for knowledge discovery. *Information Sciences*, 155(3), 2003.

[31] Mocky Habeeb. *A Developer's Guide to Amazon SimpleDB*. Addison-Wesley Professional, 2010.

[32] Ibrahim F Haddad. Pvfs: A parallel virtual file system for linux clusters. *Linux Journal*, 2000(80es), 2000.

[33] Jing Han, E Haihong, Guan Le, e Jian Du. Survey on nosql database. *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE, 2011.

[34] Mark D Hill. What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4), 1990.

[35] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, e Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. *USENIX Annual Technical Conference*, volume 8, 2010.

[36] Inktank Inc. Ceph@ONLINE, mare 2015.

[37] Konrad Junemann, Philipp Andelfinger, e Hannes Hartenstein. Towards a basic dht service: Analyzing network characteristics of a widely deployed dht. *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*. IEEE, 2011.

[38] Flavio Junqueira e Benjamin Reed. *ZooKeeper: Distributed Process Coordination*. "O'Reilly Media, Inc.", 2013.

[39] Flavio P Junqueira e Benjamin C Reed. The life and times of a zookeeper. *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009.

[40] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, e Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997.

[41] Ankur Khetrapal e Vinay Ganesh. Hbase and hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, 2006.

[42] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, e Nectarios Koziris. On the elasticity of nosql databases over cloud management platforms. *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM, 2011.

[43] Ajay D Kshemkalyani e Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.

[44] Avinash Lakshman e Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2), 2010.

[45] Andrew W Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, e Ethan L Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. *FAST*, volume 9, 2009.

[46] Eliezer Levy e Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys (CSUR)*, 22(4), 1990.

[47] Bernadette Farias Lóscio, Hélio Rodrigues de OLIVEIRA, e Jonas César de Sousa PONTES. Nosql no desenvolvimento de aplicações web colaborativas. *VIII Simpósio Brasileiro de Sistemas Colaborativos, Brasil*, 2011.

[48] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, e Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[49] Marshall K McKusick, William N Joy, Samuel J Leffler, e Robert S Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3), 1984.

[50] Nimrod Megiddo e Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. *FAST*, volume 3, 2003.

[51] Justin J Miller. Graph database applications and concepts with neo4j. *Proceedings of the Southern Association for Information Systems Conference*, 2013.

[52] James H Morris, Mahadev Satyanarayanan, Michael H Conner, John H Howard, David S Rosenthal, e F Donelson Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3), 1986.

[53] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, e Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*. IEEE, 2009.

[54] John K Ousterhout, Herve Da Costa, David Harrison, John A Kunze, Mike Kupfer, e James G Thompson. *A trace-driven analysis of the UNIX 4.2 BSD file system*, volume 19. 1985.

[55] João Paiva e Luís Rodrigues. Policies for efficient data replication in p2p systems. *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*. IEEE, 2013.

[56] João Paiva e Luís Rodrigues. On data placement in distributed systems. *ACM SIGOPS Operating Systems Review*, 49(1), 2015.

[57] João Paiva e Luís Rodrigues. On Data Placement in Distributed Systems. *ACM SIGOPS Operating Systems Review*, 49, 2015.

[58] João Paiva, Pedro Ruivo, Paolo Romano, e Luís Rodrigues. Auto Placer. *ACM Transactions on Autonomous and Adaptive Systems*, 9, 2014.

[59] Joao Paiva, Pedro Ruivo, Paolo Romano, e Luís Rodrigues. A uto p lacer: Scalable self-tuning data placement in distributed key-value stores. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(4), 2015.

[60] Anita Pal e Madhumangal Pal. Interval tree and its applications. *Advanced Modeling and Optimization*, 11(3), 2009.

[61] L Sudha Rani, K Sudhakar, e S Vinay Kumar. Distributed file systems: A survey. *International Journal of Computer Science & Information Technologies*, 5(3), 2014.

[62] Eduardo A. Ribas, Roney Uba, Ana Paula Reinaldo, Arion de Campos Jr, Davi Arnaut, e Carmem Hara. Layering a dbms on a dht-based storage engine. *Journal of Information and Data Management*, 2(1), 2011.

[63] Eduardo A Ribas, Roney Uba, Ana Paula Reinaldo, Arion de Campos Jr, Davi Arnaut, e Carmem Hara. Layering a dbms on a dht-based storage engine. *JIDM*, 2(1), 2011.

[64] Robert B Ross, Rajeev Thakur, et al. Pvfs: A parallel file system for linux clusters. *Proceedings of the 4th annual Linux showcase and conference*, 2000.

[65] S Saritha. Google file system. 2010.

[66] Thorsten Schütt, Florian Schintke, e Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*. ACM, 2008.

[67] Sogand Shirinbab, Lars Lundberg, e David Erman. Performance evaluation of distributed storage systems for cloud computing. *IJ Comput. Appl.*, 20(4), 2013.

[68] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, e Robert Chansler. The hadoop distributed file system. *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010.

[69] Konstantin V Shvachko. Hdfs scalability: The limits to growth. *login*, 35(2), 2010.

[70] Abraham Silberschatz, Peter Galvin, e Greg Gagne. *Applied operating system concepts*. John Wiley & Sons, Inc., 2001.

[71] Stephen Skeirik, Rakesh B Bobba, e Jose Meseguer. Formal analysis of fault-tolerant group key management using zookeeper. *IEEE CCGrid*, 2013.

[72] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, e Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4), 2001.

[73] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, e Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Networking, IEEE/ACM Transactions on*, 11(1), 2003.

[74] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005.

[75] Andrew S Tanenbaum e Maarten Van Steen. *Distributed systems*. Prentice-Hall, 2007.

[76] Viet-Trung Tran. *Scalable data-management systems for Big Data*. Tese de Doutorado, École normale supérieure de Cachan-ENS Cachan, 2013.

[77] Bogdan George Tudorica e Cristian Bucur. A comparison between several nosql databases with comments and notes. *Roedunet International Conference (RoEduNet), 2011 10th*. IEEE, 2011.

[78] Richard van Heuven van Staereling, Raja Appuswamy, David C van Moolenbroek, e Andrew S Tanenbaum. Efficient, modular metadata management with loris. *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*. IEEE, 2011.

[79] Feiyi Wang, Mark Nelson, Sarp Oral, Scott Atchley, Sage Weil, Bradley W Settlemyer, Blake Caldwell, e Jason Hill. Performance and scalability evaluation of the ceph parallel file system. *Proceedings of the 8th Parallel Data Storage Workshop*. ACM, 2013.

[80] Klaus Wehrle, Stefan Götz, e Simon Rieche. 7. distributed hash tables. *Peer-to-Peer systems and applications*. Springer, 2005.

[81] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, e Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006.

[82] Sage A Weil, Scott A Brandt, Ethan L Miller, e Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006.

[83] Sage A Weil, Andrew W Leung, Scott A Brandt, e Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*. ACM, 2007.

[84] Dawei Xiao, Cheng Zhang, e Xiaodong Li. The performance analysis of glusterfs in virtual storage. *2015 International Conference on Advances in Mechanical Engineering and Industrial Informatics*. Atlantis Press, 2015.

[85] Shen Yin e Okyay Kaynak. Big data for modern industry: Challenges and trends [point of view]. *Proceedings of the IEEE*, 103(2), 2015.

[86] Gae-Won You, Seung-Won Hwang, e Navendu Jain. Ursa: Scalable load and power management in cloud storage systems. *ACM Transactions on Storage (TOS)*, 9(1), 2013.

[87] Hao Zhang, Yonggang Wen, Haiyong Xie, e Nenghai Yu. A survey on distributed hash table (dht): Theory, platforms, and applications. 2013.

[88] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, e Ioan Raicu. Fusionfs: Toward supporting data-intensive

scientific applications on extreme-scale high-performance computing systems. *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014.

[89] Qing Zheng, Haopeng Chen, Yaguang Wang, Jiangang Duan, e Zhiteng Huang. Cosbench: A benchmark tool for cloud object storage services. *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012.

PATRICK ANDJASUBU BUNGAMA

UM REPOSITÓRIO CHAVE-VALOR COM GARANTIA DE LOCALIDADE DE DADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Prof. Dra. Carmem Satie Hara

CURITIBA

2016