O Apache ZooKeeper como Estratégia de Monitoramento Ativo para manter o Balanceamento de Réplicas no HDFS

Rhauani Weber Aita Fazul[1], **Patrícia Pitthan Barcelos**[1]

[1]Pós-Graduação em Ciência da Computação (PGCC)
Universidade Federal de Santa Maria (UFSM)
Santa Maria – RS – Brasil

{rwfazul, pitthan}@inf.ufsm.br

Abstract. *Apache ZooKeeper is a scalable and highly reliable service for coordination in distributed environments. Following a shared namespace model based on a znode tree, ZooKeeper presents itself as an efficient solution to actively manage configuration information. In this work, we analyzed a perspective of using ZooKeeper and its znodes as a strategy for maintaining the balance in the distribution of data in HDFS: a distributed file system that operates based on data replication. By monitoring the cluster utilization in real-time, there is no need to manually trigger the execution of the HDFS native balancer, thus automating the decisions regarding the process of replica balancing in the system.*

Resumo. *O Apache ZooKeeper é um serviço escalável e altamente confiável para coordenação em ambientes distribuídos. Seguindo um modelo de namespace compartilhado baseado em uma árvore de znodes, o ZooKeeper apresenta-se como uma solução eficiente para o gerenciamento ativo de informações de configuração. Nesse trabalho, analisamos uma possibilidade de uso do ZooKeeper e seus znodes como uma estratégia para a manutenção do balanceamento na distribuição de dados no HDFS: um sistema de arquivos que opera com base na replicação. Através do monitoramento em tempo real do cluster, removeu-se a necessidade de execução manual do balanceador nativo do HDFS, automatizando assim a tomada de decisão no processo de balanceamento de réplicas.*

1. Introdução

Com o aumento da complexidade dos sistemas computacionais, a construção de aplicações distribuídas apresenta-se como um grande desafio. Para garantir uma execução correta e otimizada, os processos em um sistema distribuído devem possuir meios eficientes para a comunicação, de forma que exista algum tipo de acordo global para a realização das tarefas necessárias. Serviços como o Apache ZooKeeper[1] surgem para facilitar o processo de coordenação distribuída e confiável. Através de seu poderoso serviço centralizado, o ZooKeeper auxilia na criação de soluções robustas e escaláveis.

O ZooKeeper segue um modelo de armazenamento compartilhado e hierárquico similar a um sistema de arquivos [Foundation 2020]. As abstrações providas pelo serviço, permitem que os registradores de dados ($znodes$) sejam manipulados de forma simples e eficiente. Diversas tecnologias utilizam o modelo de dados do ZooKeeper a fim de

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

implementar primitivas de coordenação e sincronização avançadas, tais como Apache Hadoop, Kafka, Solr e HBase [Junqueira and Reed 2013].

Para garantir alto desempenho e escalabilidade em operações realizadas sobre seu *namespace*, o ZooKeeper implementa um mecanismo de $watches$, que permite notificar clientes – de forma segura e otimizada – sobre mudanças na árvore de $znodes$ [Haloi 2015]. Com tais recursos, o ZooKeeper torna-se uma solução altamente confiável e disponível para a manutenção de informações ativas em ambientes distribuídos.

Neste trabalho, avaliamos o uso do ZooKeeper como estratégia para suportar o processo de balanceamento de dados em sistemas distribuídos. Para o estudo de caso, consideramos um consolidado sistema de arquivos que utiliza a replicação de dados como um mecanismo de tolerância a falhas, o HDFS. Neste sistema, pertencente à família Apache Hadoop, o equilíbrio de réplicas no *cluster* é obtido através da execução sob demanda de um balanceador. A estratégia desenvolvida utiliza-se de uma estrutura de coordenação ativa, baseada em $znodes$ e no mecanismo de $watches$ do ZooKeeper, para suportar o processo de tomada de decisão em tempo real. Desse modo, a solução proposta permite automatizar o uso e a configuração do balanceador de réplicas no HDFS, contribuindo com a preservação da alta confiabilidade e disponibilidade dos dados no sistema.

O trabalho está organizado em 7 seções. A Seção 2 apresenta o ZooKeeper, com foco no modelo de dados e no mecanismo de $watch$ implementados pelo serviço. A Seção 3 é dedicada ao HDFS e ao seu mecanismo de replicação de dados. A Seção 4 discute as causas do desequilíbrio de réplicas e o funcionamento do balanceador nativo do HDFS. A Seção 5 descreve a estratégia desenvolvida para automatizar o balanceamento. A Seção 6 faz um relato de experiência prática com o uso da solução proposta em um ambiente distribuído. Por fim, a Seção 7 aponta as considerações finais e os trabalhos futuros.

2. Apache ZooKeeper

O Apache ZooKeeper [Foundation 2020] é um projeto *open source* com funcionalidades de coordenação confiável para auxiliar no desenvolvimento de sistemas distribuídos. Fundamentado em conceitos-chave da computação distribuída, o ZooKeeper foi projetado para executar como um serviço centralizado, similar ao DNS ou outros serviços baseados em um repositório central. A ferramenta expõe um conjunto simples de primitivas, acessíveis através de interfaces flexíveis, sobre as quais aplicações podem ser construídas a fim de implementar serviços de alto nível, como gerenciamento e configuração de *clusters*, descoberta de dispositivos, sincronização, controle de presença e grupos [Junqueira and Reed 2013]. O serviço em si é distribuído e altamente disponível.

Sendo amplamente utilizado por um grande número de organizações, tais como o Yahoo! Inc., Netflix, Twitter e Facebook [Haloi 2015], o ZooKeeper é implementado em Java, porém possui ligações para diversas linguagens de programação. A ferramenta pode operar em dois modos: *standalone* ou replicado [White 2015]. No modo de operação *standalone*, utilizado principalmente para desenvolvimento e testes, existe um único servidor e o estado do ZooKeeper não é replicado. Em um ambiente de produção, por sua vez, o ZooKeeper opera em modo replicado sobre um grupo de servidores – denominado ZooKeeper *ensemble* – que replicam o estado (configuração e dados) e, conjuntamente, atendem às requisições dos clientes. O número mínimo recomendado de servidores em

um *ensemble* é três[2], sendo cinco *hosts* o mais comum.

Todavia, se o cliente precisar que todos os servidores do *cluster*, normalmente dispostos em múltiplos *racks* através de diferentes *switches* de rede, efetivamente armazenem os dados antes de continuar com a execução, o atraso gerado pode deixar de ser aceitável. Sendo assim, utiliza-se uma estratégia de *quorum*: número mínimo de servidores que precisam estar disponíveis e com estado sincronizado para que o ZooKeeper funcione. Um *ensemble* de tamanho n pode possuir um *quorum* de $n - i$ servidores. Desse modo, no momento em que os $n - i$ servidores efetivarem o armazenamento, o cliente é informado de que os dados estão seguros e poderá continuar a executar as demais operações. Os outros i servidores, eventualmente, irão sincronizar com o estado do *quorum* mais recente e persistir os dados em seu armazenamento local.

No modo de operação replicado, as instâncias do ZooKeeper seguem uma arquitetura *1-master–n-followers*. O ZooKeeper, de forma transparente, elimina o ponto único de falha (*SPOF – single point of failure*) de modo que, no advento de uma falha do servidor mestre (detectada a partir de mensagens *heartbeat*), uma nova eleição de líder acontece e uma instância, até então seguidora, assume a função de líder [Haloi 2015]. Isso reforça a tolerância a falhas da ferramenta, uma vez que, enquanto a maioria dos servidores estiver executando, o serviço estará sempre disponível. Além disso, uma ordem estrita das transações é garantida, o que permite a implementação de primitivas e receitas de sincronização avançadas. Os clientes podem se conectar a qualquer membro do *ensemble* para realizar operações (um balanceamento de carga mínimo é realizado nativamente).

Em geral, processos em um sistema distribuído podem se comunicar através de duas formas principais: troca de mensagens através da rede ou realizando operações de escrita e leitura em algum armazenamento compartilhado [Junqueira and Reed 2013]. Podendo ser visto como um sistema de arquivos simplificado que implementa abstrações voltadas à coordenação [White 2015], o ZooKeeper segue um modelo de *namespace* compartilhado, organizado hierarquicamente em forma de árvore.

Os registradores de dados, chamados de $znodes$ no contexto do ZooKeeper, podem armazenar dados até 1MB, tipicamente em formato de *bytes*. Os $znodes$ podem possuir outros nodos associados (filhos). O caminho da raiz da árvore até um $znode$ é canônico e absoluto. Existem dois tipos de $znodes$ que devem ser especificados no momento da criação [Haloi 2015]: (i) persistentes, que devem ser deletados explicitamente; e (ii) efêmeros, que são deletados automaticamente quando a sessão do cliente é encerrada, independente do motivo (esses $znodes$ não podem possuir filhos na árvore). Além disso, um qualificador sequencial pode ser associado aos $znodes$, de forma que um número incremental – mantido no $znode$ pai – seja concatenado ao final de seu caminho.

O ZooKeeper é utilizado para o armazenamento de pequenos volumes de dados destinados à coordenação, como informações de configuração, *status* ou localização. Em contraste com sistemas de arquivos convencionais, os dados do ZooKeeper são mantidos em memória [Foundation 2020]. Assim, alto desempenho no atendimento das requisições é garantido pela ferramenta, em especial nas operações de leitura, que são processadas localmente no servidor ZooKeeper ao qual o cliente está conectado no momento. As

[2]É recomendado que o *ensemble* seja formado por um número ímpar de servidores do ZooKeeper, reduzindo assim as chances de erros e inconsistências de estados, e.g., cenários de *split-brain* [Haloi 2015].

solicitações de escrita, por sua vez, são encaminhadas ao líder e passam por um consenso majoritário antes que uma resposta seja retornada ao cliente (há a garantia de atomicidade e durabilidade). Por ser normalmente utilizado como um serviço remoto, acessar um $znode$ toda vez que um cliente precisar verificar seu conteúdo poderia trazer um custo elevado [Junqueira and Reed 2013]. Ao invés disso, o modelo de dados do ZooKeeper dispõe de um mecanismo de notificações, conforme aborda a Seção 2.1.

2.1. ZooKeeper Watches

Para fornecer alto desempenho e escalabilidade, o ZooKeeper necessita de estratégias eficientes para que as requisições ao seu serviço centralizado possam ser atendidas de forma otimizada. A ideia é evitar que clientes realizem acessos recorrentes a fim de verificar se os dados mantidos na árvore de $znodes$ foram modificados (*polling*) [Haloi 2015]. Neste sentido, o ZooKeeper implementa um mecanismo de $watch$, o qual permite que os clientes se registrem para receber notificações quando um $znode$ sofrer alguma alteração.

Os $watches$ são setados em $znodes$ específicos a partir de operações realizadas pelo próprio cliente. Quando um $znode$ é alterado, uma notificação é despachada para o cliente que registrou o $watch$. Os $watches$ são disparados apenas uma única vez e devem ser registrados novamente pelo cliente a cada nova notificação[3], a fim deste continuar sendo notificado sobre possíveis alterações futuras. Os $watches$ permitem observar as seguintes condições [White 2015]: (i) criação ou exclusão de um $znode$ em um determinado caminho; (ii) mudanças nos dados mantidos por um $znode$; e (iii) criação ou exclusão de um dos filhos na sub-árvore de um determinado $znode$.

O ZooKeeper garante que uma notificação de alteração em um $znode$ seja entregue antes que qualquer outra alteração seja feita no $znode$ em questão. Com isso, mesmo em casos de lentidão na propagação, as notificações preservam a ordem de atualizações de estado observadas pelos clientes (i.e., ordenação global, baseada no método FIFO, de $watches$ e notificações) [Junqueira and Reed 2013]. Vale ressaltar que as operações de leitura e escrita realizadas no *namespace* são atômicas. Assim, leituras retornam todos os *bytes* associados com o $znode$ e escritas substituem todos os dados [Foundation 2020].

Com mecanismos otimizados e confiáveis como os $watches$, o ZooKeeper auxilia na resolução de problemas complexos de um modo simples e organizado, aliviando os desenvolvedores das dificuldades de implementar tais serviços desde o início. Com um modelo de programação elegante e poderoso, o ZooKeeper acaba por ser empregado na *stack* de tecnologias de uma variedade de sistemas distribuídos, sendo inclusive parte do ecossistema do Apache Hadoop [White 2015]. É importante notar que, embora muitos projetos relacionados ao Hadoop utilizem o ZooKeeper (e.g., Apache Flume, HBase e HDFS), seu serviço possui um baixo acoplamento com tecnologias específicas.

Neste trabalho, utilizamos o ZooKeeper como um elemento-chave para a implementação de uma estratégia de monitoramento em tempo de execução e tomada de decisão automatizada no contexto de sistemas de arquivos distribuídos. Para o estudo de caso, consideramos o problema do desbalanceamento na distribuição de dados no conhecido motor de armazenamento de arquivos da família Hadoop: o HDFS.

[3] A versão 3.6.0 do ZooKeeper, lançada em 4 de março de 2020, trouxe o conceito de $watches$ permanentes e recursivos, que não são removidos quando disparados pela primeira vez [Foundation 2020].

3. HDFS

O *Hadoop Distributed File System* (HDFS) é um sistema de arquivos distribuído e escalável voltado ao armazenamento de volumes massivos de dados. Por oferecer um ambiente confiável e otimizado para a manutenção e gerência de *big data*, o HDFS é incorporado como camada de armazenamento por diferentes *frameworks* de processamento paralelo, tais como Apache Spark, Storm e Samza [White 2015], além de ser compatível com tecnologias de banco de dados como Apache Phoenix e HBase.

No HDFS segue-se uma arquitetura mestre-escravo baseada em 1 `NameNode` (NN) e n `DataNodes` (DN). O NN é o servidor mestre responsável por manter o *namespace* do sistema e controlar o acesso e a distribuição dos arquivos, enquanto os DNs são os *workers* responsáveis pelo armazenamento dos dados. O HDFS adota uma estrutura de armazenamento em blocos, onde os arquivos inseridos no sistema são segmentados em blocos de dados de tamanho fixo (128MB, por padrão) e distribuídos entre os DNs. Um *cluster* HDFS é escalável até milhares de servidores e geralmente é constituído por *hardware* de baixo custo (i.e., não confiável) [Srinivasa and Muppalla 2016]. Assim, ao invés de depender de *hardware* para fornecer alta disponibilidade e confiabilidade, o HDFS necessita de mecanismos eficientes para a detecção e prevenção de falhas.

Dentre os principais mecanismos de tolerância a falhas implementados pelo HDFS estão [White 2015]: (i) o estabelecimento de *checkpoints* periódicos sobre o *namespace* do NN, a fim de evitar a recomputação excessiva em casos de falhas do servidor mestre; (ii) as mensagens *heartbeat* para o envio de sinais periódicos, que permitem ao NN detectar falhas operacionais em DNs; (iii) o modo de alta disponibilidade do NN (HDFS--HA), que mantém NNs sincronizados em *stand-by* para eliminar a condição de SPOF do sistema[4]; e (iv) a replicação de blocos, que estabelece redundância com o intuito de evitar que dados sejam perdidos em situações de falhas. Como este trabalho tem seu foco voltado à replicação, a Seção 3.1 detalha o funcionamento deste mecanismo no HDFS.

3.1. Replicação de Dados no HDFS

A replicação baseia-se na criação de cópias dos blocos de dados criados a partir do particionamento do arquivo original (réplicas). Os blocos replicados são armazenados em diferentes DNs, assim, caso um nodo falhe, seus blocos continuam disponíveis em um ou mais DNs que mantenham suas réplicas. A quantidade de réplicas a ser criada para cada bloco é configurada, por arquivo, através do Fator de Replicação (FR). Uma aplicação pode especificar o FR no momento da criação de um arquivo, sendo possível modificá-lo posteriormente através de utilitários do sistema [Srinivasa and Muppalla 2016]. A replicação garante que, com um FR de n (por padrão, $n = 3$), os dados armazenados no HDFS permaneçam seguros mesmo na ocorrência de $n - 1$ falhas simultâneas de DN.

Para garantir alta disponibilidade em cenários propensos a falhas consecutivas, o NN precisa controlar o número de réplicas ativas de cada bloco e, quando necessário, conduzir o processo de re-replicação. Em decorrência de alguma falha, os DNs podem perder a comunicação com o servidor mestre. Caso o NN não receba mensagens *heartbeat* de um

[4]O próprio ZooKeeper é utilizado para implementar esta funcionalidade. O NN ativo do HDFS mantém um $znode$ efêmero na árvore de $znodes$ do ZooKeeper. Em caso de falha, a conexão do NN irá expirar e o $znode$ em questão será excluído automaticamente, disparando uma notificação via $watch$. Essa notificação, por sua vez, resulta em uma nova eleição para a função de NN ativo dentre os NNs em espera.

DN dentro de um período de tempo pré-estipulado, o DN é marcado como inativo e o FR dos blocos nele armazenados é decrementado. Se o número de réplicas de um determinado bloco for inferior ao FR definido, a re-replicação pode ser iniciada pelo NN através das cópias excedentes mantidas em DNs ativos. Assim, restaura-se a conformidade com o FR e a disponibilidade esperada dos dados caso novas falhas venham a ocorrer.

A forma de posicionamento dos blocos – tanto os re-replicados quanto os gerados a partir da distribuição inicial – é um aspecto importante para um bom funcionamento do HDFS, assegurando alta confiabilidade e disponibilidade. Atuando como o nodo mestre, o NN utiliza uma Política de Posicionamento de Réplicas (PPR) como referência durante a escolha dos DNs para o recebimento dos dados. A PPR, seguindo um modelo de distribuição *rack-aware*, estabelece que: (i) a primeira réplica seja armazenada no mesmo nodo do cliente ou, para clientes executando fora do *cluster*, em um DN escolhido arbitrariamente (o sistema evita, na medida do possível, escolher nodos que apresentem um alto tráfego de comunicação); (ii) as duas réplicas seguintes sejam armazenadas em DNs distintos em um mesmo *rack* remoto, diferente do *rack* onde a primeira réplica foi armazenada; (iii) caso o FR seja maior que o padrão, as demais réplicas sejam posicionadas de forma aleatória, porém mantendo o número de réplicas por *rack* abaixo do limite determinado por $((\text{réplicas} - 1) \ / \ \textit{racks}) + 2$ [Foundation 2019].

A estratégia da PPR assegura alta confiabilidade e disponibilidade dos dados mesmo em caso de falha de um *rack* por completo (as réplicas de um mesmo bloco não ficam armazenadas em um único *rack*). Ainda, por manter uma réplica no nodo local e as duas réplicas seguintes em um mesmo *rack*, é possível reduzir o custo do tráfego de dados necessário para o armazenamento. Adicionalmente, de forma a otimizar o processo de escrita das réplicas e diminuir a sobrecarga, aplica-se um *pipeline* [Foundation 2019], onde DNs podem, simultaneamente, receber e encaminhar dados. A replicação de dados também é utilizada para otimizar o desempenho do HDFS, onde é possível explorar a redundância espacial dos blocos para suprir altas demandas de acesso e tornar a leitura dos dados mais rápida, i.e., operações de leitura podem tirar proveito da maior disponibilidade dos dados e utilizar a largura de banda de múltiplos *racks*.

3.2. Localidade dos Dados

Muitas arquiteturas de HPC (*High-Performance Computing*) tradicionais possuem os nodos de computação separados dos nodos de armazenamento, sendo esses conectados por enlaces de rede de alta velocidade [Guo et al. 2012]. Em sistemas paralelos como o HDFS, por sua vez, cada nodo é responsável tanto pelo armazenamento quanto pelo processamento dos dados. Sendo um sistema baseado no modelo de acesso WORM (*write once*, *ready many*), muitos dos esforços do HDFS são voltados a maximizar a vazão durante operações de leitura dos dados [Achari 2015]. Assim, o posicionamento dos blocos no *cluster* é um fator crítico para a confiabilidade do sistema e para o desempenho de aplicações voltadas a entrada e saída (E/S) [Shvachko et al. 2010]. Um posicionamento adequado das réplicas permite reduzir o tráfego de dados entre *switches*, que pode se tornar um gargalo de desempenho em sistemas de computação intensiva.

Neste sentido, um dos pilares para o processamento no Hadoop é mover a aplicação até os dados, evitando mover os dados propriamente ditos [Foundation 2019]. Levar a computação para perto dos dados armazenados no HDFS é conhecido como localidade dos dados (*data locality*) [White 2015]. Esta funcionalidade permite aprimorar

a eficiência da plataforma no processamento de grandes *datasets*, já que, por ser local, o acesso aos dados torna-se mais rápido e evita transferências desnecessárias. Caso um processamento local não seja possível, selecionam-se os nodos que tenham um caminho de rede mais rápido para os DNs que mantenham os blocos necessários para a operação.

Como cada bloco é replicado, por padrão, em três DNs diferentes, a probabilidade de que uma tarefa de computação consiga processar a maioria dos blocos *in-place* é alta [Achari 2015]. Entretanto, uma distribuição de réplicas desequilibrada pode fazer com que a localidade dos dados deixe de ser explorada de forma otimizada. A medida que o desbalanceamento se agrava, o número de transferências intra-*rack* ou *off-rack* aumenta, consumindo de forma elevada a largura de banda do *cluster*. Além disso, o desbalanceamento pode acarretar em sobrecarga para os DNs com alta utilização (nodos que possuem mais blocos armazenados) e em ociosidade para nodos com baixa utilização, degradando ainda mais o desempenho de aplicações que realizem grande número de operações de E/S. O desequilíbrio no posicionamento dos blocos inerente à PPR e a importância do balanceamento de réplicas no HDFS são discutidos a seguir, na Seção 4.

4. Balanceamento de Réplicas no HDFS

O HDFS foi projetado para suportar e operar sobre um volume massivo de dados. O armazenamento de arquivos maiores, naturalmente, implica na criação de uma maior quantidade de blocos, os quais devem ser distribuídos através do *cluster*. O posicionamento das réplicas seguindo a PPR, garante um balanceamento mínimo, onde as réplicas de um determinado bloco não recaem em um mesmo DN. Embora aumente a confiabilidade do sistema em caso de falhas pela redundância de dados em *racks* distintos e contribua com uma melhor utilização dos recursos computacionais do *cluster*, a PPR não distribui os blocos de forma totalmente igualitária entre os DNs [Foundation 2019].

A PPR pode favorecer o desbalanceamento de réplicas em dois sentidos. Considerando o FR padrão, um *rack* é escolhido para manter dois terços das réplicas de um bloco, o que contribui com o desequilíbrio inter-*rack*. Já o desbalanceamento intra-*rack* (inter-DN) é agravado pela arbitrariedade na escolha dos nodos. Em geral, qualquer DN do *cluster* que satisfaça as restrições impostas pela política é eletivo ao armazenamento da réplica, sendo responsabilidade do NN a decisão final.

Outros aspectos também contribuem com o desequilíbrio de réplicas no HDFS, tais como [Hortonworks 2019]: (i) a adição de um novo DN ao *cluster*, já que esse irá competir igualmente com outros nodos para o recebimento dos blocos replicados, resultando em um período de subutilização significativo [Turkington 2013]; (ii) o comportamento da aplicação do cliente que, caso execute diretamente em um DN do *cluster*, faz com que esse, de acordo com a PPR, sempre mantenha uma das réplicas localmente; e (iii) o processo de re-replicação, que está sujeito à mesma política da replicação inicial.

Uma possível estratégia para minimizar o problema do desbalanceamento seria adaptar a PPR padrão do HDFS, de forma que esta passe a considerar a utilização dos DNs do *cluster* como critério para distribuição dos dados [Ibrahim et al. 2016]. Contudo, existem situações onde o desbalanceamento é inevitável (e.g., adição de novos DNs), o que faz com que abordagens reativas/corretivas tornem-se necessárias. A solução oficial para o balanceamento de réplicas reativo no Hadoop é o `HDFS Balancer`.

4.1. HDFS Balancer

O `HDFS Balancer` [Shvachko et al. 2010] é um utilitário, integrado na distribuição do Hadoop, destinado ao balanceamento de réplicas entre dispositivos de armazenamento no HDFS. A partir da análise do posicionamento dos blocos presentes no sistema de arquivos, a ferramenta opera iterativamente movimentando blocos de DNs que apresentarem uma alta utilização (origem) para DNs que possuírem um menor volume de dados armazenado (destino). A *daemon* do `HDFS Balancer` foi projetada para operar sem afetar as atividades normais do *cluster* ou interferir com os clientes e suas aplicações [White 2015].

A operação de balanceamento é guiada por um *threshold*, que é passado como parâmetro para a execução do `HDFS Balancer`. Representado como uma porcentagem no intervalo de 0% a 100%, o *threshold* limita a diferença máxima que a utilização (proporção do espaço em uso para a capacidade total) dos DNs e a utilização média do *cluster* pode assumir [White 2015]. Quando a utilização de cada DN estiver dentro desse limite, que tem o valor padrão de 10%, o *cluster* é considerado balanceado. Ao reduzir o *threshold* aumenta-se o equilíbrio do *cluster*, porém maior será o esforço – em termos de processamento e de transferência de dados – necessário para realizar o balanceamento.

O algoritmo padrão empregado pelo `HDFS Balancer` é composto por quatro fases principais que são executadas iterativamente, sendo elas [Hortonworks 2019]: classificação dos dispositivos, pareamento dos grupos, agendamento das movimentações e transferência dos dados. Na fase de classificação, os dispositivos de armazenamento do *cluster* são divididos em grupos de acordo com seus tipos (e.g., disco rígido e SSD). Tomando i como um DN qualquer e t como um tipo de dispositivo de armazenamento, define-se: $U_{i,t}$ como a porcentagem representando a utilização do grupo dos dispositivos do tipo t configurados no DN i; e $U_{\mu,t}$ como a porcentagem representando a média de utilização de todos os dispositivos do tipo t do *cluster*. Com isso, cada grupo pode ser classificado em: (i) superutilizado, quando $U_{i,t} > U_{\mu,t} + threshold$; (ii) acima da média, quando $U_{\mu,t} + threshold \geq U_{i,t} > U_{\mu,t}$; (iii) abaixo da média, quando $U_{\mu,t} \geq U_{i,t} \geq U_{\mu,t} - threshold$; ou (iv) subutilizado, quando $U_{\mu,t} - threshold > U_{i,t}$.

Se o *cluster* HDFS não possuir nenhum grupo subutilizado ou superutilizado, ele é considerado balanceado. Caso contrário, a execução do `HDFS Balancer` continua na fase de pareamento dos grupos, onde formam-se pares origem-destino entre os grupos de dispositivos classificados anteriormente. Um grupo que extrapole o *threshold* pode resultar na criação de um ou mais pares na mesma iteração em uma relação $1 - N$, sendo ele o grupo origem (caso tenha sido classificado como superutilizado ou acima da média) ou o grupo destino (caso tenha sido classificado como subutilizado ou abaixo da média). A estratégia de pareamento procura, inicialmente, grupos de um mesmo *rack*.

Na fase de agendamento de movimentação dos blocos, selecionam-se os blocos para serem redistribuídos entre cada par origem-destino definido. Um bloco do grupo de um DN origem é eletivo ao movimento se: (i) estiver armazenado em um dispositivo do mesmo tipo que o destino; (ii) não possuir uma réplica já existente no destino; (iii) não estiver em processo de movimentação, nem for um dos blocos já movimentados na iteração corrente; e (iv) após a movimentação, continuar em concordância com a PPR. Após a definição do bloco a ser realocado, o `HDFS Balancer` esforça-se em diminuir o tráfego de dados através do *cluster* selecionando o DN mais próximo do destino que possuir uma réplica do bloco a ser movimentado como *proxy*. Assim, na etapa de transferência dos

dados, o DN destino faz a cópia do bloco mantido no DN *proxy* para seu armazenamento local. Ao final, ele envia um alerta para o NN, que dispara a operação de exclusão da réplica armazenada no DN origem. Se, após todas as movimentações da iteração serem completadas, o *cluster* ainda não estiver equilibrado, uma nova iteração será iniciada.

Embora a arquitetura do HDFS seja compatível com esquemas de rebalanceamento para movimentação automática de dados entre DNs, o sistema de arquivos do Hadoop não implementa tal funcionalidade de forma nativa [Foundation 2019]. Desse modo, a execução do balanceador precisa ser disparada de forma manual pelo administrador do *cluster*. O uso sob demanda do balanceador pode tornar-se em um problema, resultando em gargalos de desempenho ao ficar propenso ao esquecimento, a configurações ineficientes e/ou a escolhas inapropriadas para o momento do disparo de sua execução.

Ferramentas como o ZooKeeper podem auxiliar nesse aspecto. O uso de $watches$, por exemplo, permite a criação de serviços de configuração ativos, em que os clientes interessados podem ser notificados sobre alterações de configuração e realizar as ações necessárias [White 2015]. No entanto, o ZooKeeper expõe apenas primitivas, e cabe aos desenvolvedores utilizar tais primitivas – seguindo construtores de alto nível (receitas) – para resolver os problemas de coordenação em suas aplicações distribuídas. Neste trabalho, utilizamos o ZooKeeper para implementar uma solução de configuração e disparo automático do balanceamento de réplicas no HDFS, conforme aborda a Seção 5.

5. Estratégia para Manutenção do Balanceamento de Réplicas

Com o objetivo de analisar o ambiente computacional e o estado de utilização do *cluster*, adotou-se uma estratégia de monitoramento dos nodos em tempo de execução com base no modelo agente-servidor. Neste modelo, cada um dos nodos – que também executa um processo DN do HDFS – possui um $agente$ que coleta informações sobre o estado de seus dispositivos de armazenamento e da instância do HDFS em sua máquina.

Os $agentes$ reportam suas observações para um servidor – endereçado neste trabalho como $gerente$ – que, tendo uma visão global do estado *cluster*, reúne e processa as informações de todos os nodos. Se o $gerente$ julgar necessário uma ação corretiva (i.e., balanceamento do sistema), ele irá notificar um processo dedicado ao disparo da execução do `HDFS Balancer`. Este processo, chamado de $controlador$, impede que operações desnecessárias sejam efetuadas e registra os resultados da operação de balanceamento.

A comunicação entre os módulos ($agentes$, $gerente$ e $controlador$) é realizada através do *namespace* hierárquico e compartilhado do Apache ZooKeeper. A Figura 1 apresenta a visão global da árvore de $znodes$ utilizada para a coordenação e controle do processo de balanceamento de réplicas no HDFS. Para simplificar a visualização, omitiu-se parte do caminho absoluto dos $znodes$ (por exemplo, é exibido */disk-01* ao invés de */datanodes/dn-01/disks/disk-01*). Além disso, por ser uma árvore n-ária, utiliza-se o identificador n em um determinado nível para denotar um número não limitado de nodos. Cada módulo manipula uma sub-árvore em específico e pode realizar observações sobre as demais. O detalhamento dos módulos e suas interações é feito a seguir.

5.1. Agentes

Os $agentes$ são executados para cada elemento do ambiente computacional que necessite de monitoramento (i.e., DNs), de modo que informações sobre seus recursos sejam cole-

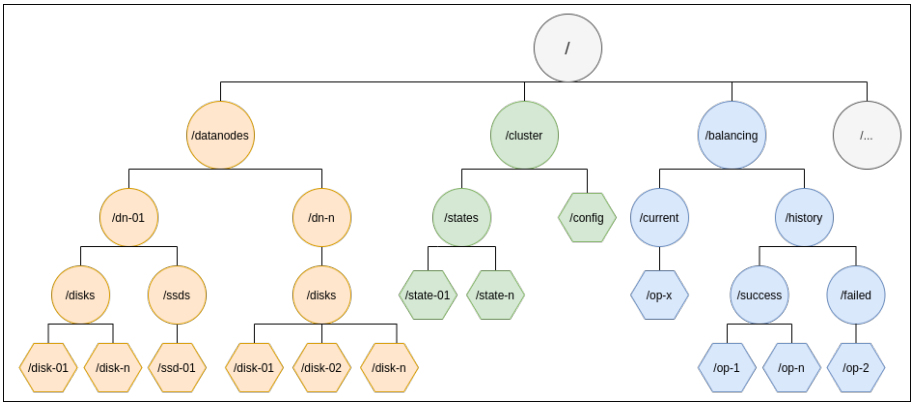

Figura 1. Árvore no ZooKeeper para controle do balanceamento no HDFS.

tadas periodicamente em tempo real. Na Figura 1, os $agentes$ manipulam a sub-árvore */datanodes*. Em específico, cada $agente$ relaciona-se com um $znode$ (persistente e sequencial) que representa o DN que executa em seu nodo (e.g., */dn-01*). Para permitir uma identificação posterior, esse $znode$ armazena o *hostname* do DN em questão.

Informações de todos os dispositivos de armazenamento dos DNs do *cluster* são coletadas. Se o $agente$ percebe uma alteração significativa, ele atualiza o conteúdo de um dos nodos mantidos como filho em seu respectivo $znode$. Para tal, os dispositivos são separados de acordo com seus tipos (e.g., */disks* e */ssds*). Essa estrutura é flexível e pode ser alterada em tempo de execução, conforme necessário. Cada $znode$ que engloba um tipo de dispositivo possui $znodes$ filhos que representam o dispositivo em específico (e.g., */ssd-01*) e armazenam as respectivas informações coletadas, tais como os espaços de armazenamento total, disponível e ocupado neste dispositivo. Posteriormente, essas informações serão consumidas pelo módulo $gerente$.

5.2. Gerente

O $gerente$ é responsável pelo processamento e análise das informações de utilização do ambiente. Os resultados obtidos são armazenados na sub-árvore */cluster* da Figura 1. De modo a estar sempre atualizado sobre o estado mais recente do sistema, o $gerente$ mantém um $watch$ no $znode$ */datanodes* e assim é automaticamente notificado sobre qualquer mudança significativa no ambiente que for observada pelos $agentes$.

Com uma visão global do sistema, é responsabilidade do $gerente$ verificar se o balanceamento de réplicas faz-se necessário no *cluster*. Neste trabalho, aplicou-se um cálculo similar ao processo realizado internamente pelo `HDFS Balancer` (vide fase de classificação dos grupos apresentada na Seção 4.1), ou seja, a divergência da utilização dos dispositivos de armazenamento de um tipo t ($U_{i,t}$) para a média de utilização do *cluster* ($U_{\mu,t}$). Entretanto, é importante observar que, por se tratar de um cálculo a parte do balanceador nativo do HDFS, não há limitação quanto às fórmulas a serem empregadas. A porcentagem de utilização do *cluster* e os demais valores analisados são armazenados em um histórico a partir de filhos do $znode$ */states*.

Se o $gerente$ perceber um desequilíbrio de réplicas acentuado no *cluster*, inicia-se uma rotina destinada à definição de atributos de balanceamento (parâmetros a serem passados para a execução do balanceador). Diferentes atributos podem ser adaptados em

tempo de execução conforme necessário, tais como: largura de banda máxima a ser consumida pela operação do `HDFS Balancer`, DNs a serem incluídos ou excluídos na redistribuição das réplicas, quantidade limite de *threads* a serem executadas no *cluster* e volume máximo de dados a ser movimentado durante uma iteração de balanceamento.

Para este trabalho, a definição dos atributos foi realizada com base nos modos de balanceamento *background* e *fast*, detalhados em [Hortonworks 2019]. Todavia, cabe ressaltar que, a medida que novas notificações são recebidas e processadas pelo $gerente$, é possível realizar uma análise das informações históricas do *cluster* ($znode$ */states*) e das operações anteriores do `HDFS Balancer` para uma melhor adaptação dos atributos. Isso possibilita previsões acuradas sobre o estado do sistema. Independente do método empregado para a definição dos atributos de balanceamento, esses devem ser armazenados no $znode$ */config*, o que gerará uma notificação para o módulo $controlador$.

5.3. Controlador

O $controlador$ é o módulo responsável pela manipulação da sub-árvore */balancing* exibida na Figura 1. Toda vez que o módulo $gerente$ julgar que uma nova operação de balanceamento de réplicas é necessária no *cluster*, o $controlador$ é notificado através de um $watch$ registrado no $znode$ */config*. Ao receber a notificação, cabe ao $controlador$ decidir se a operação deve ou não ser efetivada.

Para este trabalho, consideramos duas validações primárias: (i) nenhuma outra operação de balanceamento deve estar em execução (i.e., $znode$ */current* não possui nodo filho no momento); e (ii) a última operação de balanceamento registrada no $znode$ */history* deve ter sido realizada há pelo menos n segundos, sendo n um valor customizável, com valor padrão de 600 (equivalente a 10 minutos) para dispositivos de armazenamento do tipo disco e 30 para dispositivos do tipo SSD. Modelos mais sofisticados para a tomada de decisão também podem ser empregados, a fim de reforçar a função do $controlador$. Por exemplo, levar em consideração o nível de sobrecarga no *cluster* para decidir se a notificação vinda do $gerente$ deve ser atendida de imediato ou postergada para algum momento no futuro, de modo a causar menor impacto no desempenho do HDFS.

Se as validações forem satisfeitas, uma nova operação de balanceamento é iniciada. Assim, o $controlador$ armazena a operação corrente no $znode$ */current* e dispara a execução do `HDFS Balancer` – passando os parâmetros de execução definidos no $znode$ */config* – pelo comando *balancer* acessível através do *script bin/hdfs* do Hadoop. O $controlador$ espera a *daemon* do balanceador terminar sua execução a fim de coletar informações da operação e armazená-las na sub-árvore */history*. O *exit status* retornado informa se o balanceamento falhou ou teve sucesso. Em caso de falha, o *log* da operação é salvo como um $znode$ filho de */failed*. Esses registros podem ser utilizados para análises futuras. Já em caso de sucesso, os seguintes dados são obtidos: (i) *timestamp* ao final da execução; (ii) tempo gasto com a operação do balanceador (i.e., tempo de execução); (iii) volume total de dados movimentado; e (iv) número de iterações de balanceamento realizadas. Estes dados, em conjunto com as configurações utilizadas na respectiva operação de balanceamento, dão origem a um novo filho no $znode$ */success*.

Com a estratégia relatada nesta Seção, o balanceamento de réplicas passa a ser conduzido com base no monitoramento em tempo real do ambiente computacional, removendo assim a necessidade de disparo manual do `HDFS Balancer`. Através da

manutenção ativa do equilíbrio na distribuição dos dados no *cluster*, tende-se a aprimorar a confiabilidade e a disponibilidade dos dados, contribuindo para o bom funcionamento do HDFS. Um experimento prático utilizando a solução proposta é apresentado na Seção 6.

6. Relato de Experiência Prática

A experimentação foi realizada na plataforma GRID'5000[5], com o Apache ZooKeeper (versão 3.6.0) executando em modo replicado sobre uma distribuição Debian GNU/Linux 10 (*buster*). Foram configurados 10 nodos (modelo Dell PowerEdge R630) no *cluster ecotype* do site *Nantes*, cada um com 2 CPUs Intel Xeon E5-2630L v4 (Broadwell, 1,80GHz, 10 cores/CPU), 128GB de memória RAM, 400GB de capacidade de armazenamento SSD (SAS Toshiba PX04SMB040) e 2 conexões Ethernet de 10GB.

Nesse ambiente, o sistema de arquivos do Apache Hadoop (versão 2.9.2) foi configurado com 1 NN e 10 DNs (um por nodo). Para realizar a carga dos dados, utilizou-se o `TestDFSIO` [White 2015]: um *benchmark* distribuído que testa o desempenho do HDFS através de operações paralelas de leitura e escrita intensivas (I/O *bound*). Foram escritos 25 arquivos de 30GB com o FR padrão (3 réplicas por bloco). Ao total, o volume de dados gerado foi de aproximadamente $2{,}2$TB (18000 blocos de 128MB cada), o que fez com que, ao final da escrita, a utilização média do *cluster* ($U_{\mu,t}$) ficasse por volta de $75{,}20\%$.

Conforme detalhado na Seção 5, a estratégia de monitoramento baseia-se no mecanismo de $watches$ do ZooKeeper, possibilitando que o balanceador seja disparado de forma automática a medida que o desequilíbrio na distribuição de dados no *cluster* se agrava. Desse modo, consideramos dois cenários de teste: (i) sem o balanceamento de réplicas; e (ii) com o uso da estratégia para manutenção do estado de equilíbrio do HDFS.

6.1. Resultados

A Tabela 1 exibe, em cada cenário de teste, a ocupação em GB ($O_{i,SSD}$) e a porcentagem de utilização ($U_{i,SSD}$) de cada DN do *cluster* após o *benchmark* finalizar a escrita. No cenário sem balanceamento, percebe-se um grande desequilíbrio na distribuição dos dados, com um desvio padrão (σ) na ocupação e utilização dos DNs de, respectivamente, $57{,}34$GB e $17{,}82\%$. Já com a estratégia para manutenção do balanceamento de réplicas proposta neste trabalho, permite-se manter a utilização dos DNs dentro de um intervalo controlado, dado pelos limites inferior ($U_{\mu,t} - threshold$) e superior ($U_{\mu,t} + threshold$) adotados pelo `HDFS Balancer`. O equilíbrio alcançado é evidenciado pelo desvio padrão reduzido na ocupação e utilização dos DNs do *cluster*: $3{,}51$GB e $1{,}09\%$, respectivamente.

Para permitir tal equilíbrio, a *daemon* do `HDFS Balancer` foi disparada 6 vezes pelo $controlador$ durante a escrita dos dados, com um *threshold* configurado em 7%. O tempo de balanceamento, quantidade de iterações (it) e volume de dados movimentado em cada uma das 6 operações do balanceador foram, respectivamente: (i) 355s, $28it$ e $265{,}88$GB; (ii) 50s, $3it$, $38{,}88$GB; (iii) 62s, $4it$, $49{,}63$GB; (iv) 84s, $6it$, $63{,}75$GB; (v) 74s, $5it$, $58{,}88$GB; e (vi) 61s, $4it$, $43{,}75$GB. Isso demostra que, inicialmente, houve uma espera maior para o disparo do balanceador, de modo que a primeira operação moveu um volume de dados superior às demais. A partir disso, operações mais rápidas e com um menor

[5]Grid'5000 é uma plataforma para experimentos apoiada por um grupo de interesses científicos hospedado por Inria e incluindo CNRS, RENATER e diversas Universidades, bem como outras organizações (mais detalhes em `https://www.grid5000.fr`).

Tabela 1. Estado final do HDFS após a escrita dos arquivos.

DataNode	sem balanceamento		estratégia proposta	
	$O_{i,SSD}$ (GB)	$U_{i,SSD}$ (%)	$O_{i,SSD}$ (GB)	$U_{i,SSD}$ (%)
DN_{01}	181,28	56,35	223,37	69,43
DN_{02}	176,62	54,90	227,40	70,68
DN_{03}	304,24	94,57	220,98	68,69
DN_{04}	298,44	92,76	227,87	70,83
DN_{05}	181,16	56,31	227,45	70,70
DN_{06}	169,94	52,82	234,46	72,88
DN_{07}	222,73	69,23	224,88	69,90
DN_{08}	247,67	76,98	227,27	70,64
DN_{09}	304,36	94,60	227,15	70,60
DN_{10}	181,16	56,31	227,40	70,68
σ	57,34GB	17,82%	3.51GB	1,09%

número de blocos sendo redistribuído entre os DNs do *cluster*, foram suficientes para manter o balanceamento de réplicas no HDFS.

Executar o `HDFS Balancer` durante a escrita dos arquivos, entretanto, fez com que a aplicação do `TestDFSIO` levasse $761{,}44$s para ser concluída. Em contraste, a escrita no cenário sem balanceamento levou $703{,}22$s. A variação percentual dos valores foi de $8{,}28\%$, o que indica a sobrecarga gerada pela estratégia proposta. Embora exista degradação de desempenho durante a escrita, manter o equilíbrio na distribuição dos blocos permite otimizar as futuras operações de E/S realizadas no sistema de arquivos, uma vez que o processamento no HDFS é dependente da localidade dos dados.

Para analisar possíveis otimizações providas pela manutenção do balanceamento, realizaram-se, em cada cenário de teste, 40 rodadas de execução do `TestDFSIO` destinadas à leitura total dos dados armazenados no sistema. A média aritmética dos tempos de execução do *benchmark* nessas rodadas foi de $311{,}51$s no cenário sem balanceamento e de $272{,}17$s no cenário com a estratégia proposta, o que equivale a uma variação percentual de $-12{,}63\%$, indicando redução no tempo necessário para a leitura dos dados. Adicionalmente, o *throughput* de leitura foi de $173{,}61$MB/s e $201{,}88$MB/s, resultando em um aumento de $16{,}28\%$ no *throughput* da aplicação. Já a taxa média de E/S foi de $206{,}38$MB/s e $233{,}58$MB/s, correspondendo a um aumento de $13{,}18\%$ na taxa de transferência.

Desse modo, os resultados atestam que o desequilíbrio na distribuição dos dados é uma condição prejudicial à usabilidade geral do HDFS, degradando significativamente o desempenho de aplicações voltadas a E/S. O uso da solução apresentada neste trabalho permitiu automatizar o processo de tomada de decisão para o balanceamento de réplicas, eliminando a necessidade de execução e configuração manual do `HDFS Balancer` e aprimorando o funcionamento do sistema de arquivos.

7. Considerações Finais

Este trabalho analisou uma possibilidade de uso do ZooKeeper para auxílio no processo de balanceamento de dados em sistemas de arquivos distribuídos. Através do *namespace* compartilhado para armazenamento de informações de coordenação do ZooKeeper, uma estratégia de monitoramento em tempo real do estado do *cluster* foi implementada. Essa estratégia baseia-se em $watches$: um poderoso mecanismo de notificações que permite

observar mudanças na árvore de $znodes$ do ZooKeeper. Para o estudo de caso, consideramos um sistema de arquivos que faz uso da replicação de dados, o HDFS.

A abordagem nativa para o balanceamento de réplicas nesse sistema é o `HDFS Balancer`, que precisa ser executado sob demanda – e configurado manualmente – pelo administrador do *cluster*. Com a estratégia proposta, foi possível substituir a forma de execução atual do balanceador por uma solução baseada em monitoramento ativo, que permite a configuração e o disparo automático do `HDFS Balancer`. Ao realizar um experimento prático, verificou-se a efetividade da estratégia em automatizar o processo de tomada de decisão para manter o balanceamento na distribuição dos dados no HDFS.

Trabalhos futuros envolvem a implementação de abordagens de aprendizagem sobre o histórico de informações do sistema e das operações de balanceamento realizadas no *cluster*. Com a análise em tempo de execução desse histórico, mantido na árvore de $znodes$ do ZooKeeper, será possível prever o estado do HDFS para realizar a escolha dos parâmetros de balanceamento de forma otimizada. Além disso, pretende-se adaptar a solução apresentada neste trabalho para outros sistemas de arquivos distribuídos que podem se beneficiar da automatização do processo de balanceamento de réplicas.

Referências

Achari, S. (2015). *Hadoop Essentials*. Packt Publishing Ltd, Birmingham, 1st edition.

Foundation, A. S. (2019). "HDFS Architecture". `hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign`. Novembro.

Foundation, A. S. (2020). "ZooKeeper: A Distributed Coordination Service for Distributed Applications". `https://zookeeper.apache.org/doc/r3.6.0/zookeeperOver.html#ch_DesignOverview`. Janeiro.

Guo, Z., Fox, G., and Zhou, M. (2012). Investigation of data locality in mapreduce. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 419–426, Ottawa. IEEE Computer Society.

Haloi, S. (2015). *Apache Zookeeper Essentials*. Packt Publishing Ltd, 1st edition.

Hortonworks (2019). "Balancing data across an HDFS cluster". `https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.4/data-storage/content/balancing_data_across_hdfs_cluster.html`. Dezembro.

Ibrahim, I. A., Dai, W., and Bassiouni, M. (2016). Intelligent data placement mechanism for replicas distribution in cloud storage systems. In *IEEE International Conference on Smart Cloud (SmartCloud)*, pages 134–139, New York. IEEE.

Junqueira, F. and Reed, B. (2013). *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, Inc., 1st edition.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Symposium on Mass Storage Systems and Technologies*, pages 1–10. IEEE.

Srinivasa, K. and Muppalla, A. K. (2016). *Guide to High Performance Distributed Computing*. Springer, Swindon, 1st edition.

Turkington, G. (2013). *Hadoop Beginner's Guide*. Packt Publishing Ltd, 1st edition.

White, T. (2015). *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition.