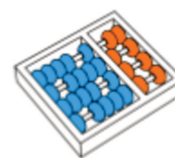




André Oriani

“Uma Solução de Alta Disponibilidade para o Sistema de Arquivos Distribuído do Hadoop”

CAMPINAS
2013



Universidade Estadual de Campinas
Instituto de Computação

André Oriani

“Uma Solução de Alta Disponibilidade para o Sistema de Arquivos Distribuído do Hadoop”

Orientador(a): Prof.^a Dr.^a Islene Calciolari Garcia

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

ESTE EXEMPLAR CORRESPONDE À VERSÃO
FINAL DA DISSERTAÇÃO DEFENDIDA POR
ANDRÉ ORIANI, SOB ORIENTAÇÃO DE
PROF.^a DR.^a ISLENE CALCIOLARI GAR-
CIA.

A handwritten signature in black ink, reading "Islene Calciolari Garcia", positioned above a horizontal line.

Assinatura do Orientador(a)

CAMPINAS
2013

FICHA CATALOGRÁFICA ELABORADA POR
MARIA FABIANA BEZERRA MULLER - CRB8/6162
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Or4s Oriani, André, 1984-
Uma solução de alta disponibilidade para o sistema de arquivos distribuído do Hadoop / André Oriani. – Campinas, SP : [s.n.], 2013.

Orientador: Islene Calciolari Garcia.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Organização de arquivos (Computação). 2. Tolerância a falha (Computação). 3. Processamento eletrônico de dados - Processamento distribuído. I. Garcia, Islene Calciolari, 1971-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em inglês: A high availability solution for the Hadoop Distributed File System

Palavras-chave em inglês:

File organization (Computer science)

Fault-tolerant computing

Electronic data processing - Distributed processing

Área de concentração: Ciência da Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora:

Islene Calciolari Garcia [Orientador]

Alfredo Goldman vel Lejbman

Luiz Eduardo Buzato

Data de defesa: 25-02-2013


Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 25 de Fevereiro de 2013, pela
Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Alfredo Goldman vel Lejbman
IME / USP



Prof. Dr. Luiz Eduardo Buzato
IC / UNICAMP



Prof.ª Dr.ª Islene Calciolari Garcia
IC / UNICAMP

Uma Solução de Alta Disponibilidade para o Sistema de Arquivos Distribuído do Hadoop

André Oriani

25 de fevereiro de 2013

Banca Examinadora:

- Prof.^a Dr.^a Islene Calciolari Garcia (Orientadora)
- Prof. Dr. Alfredo Goldman vel Lejbman
Instituto de Matemática e Estatística - USP
- Prof. Dr. Luiz Eduardo Buzato
Instituto de Computação - UNICAMP
- Prof. Dr. Gustavo Maciel Dias Viera (Suplente)
Departamento de Computação - UFSCar Campus Sorocaba
- Prof. Dr. Edmundo Roberto Mauro Madeira (Suplente)
Instituto de Computação - UNICAMP

Abstract

System designers generally adopt cluster-based file systems as the storage solution for high-performance computing environments. That happens because they provide data with reliability, consistency and high throughput. But most of those file systems employ a centralized architecture which compromises their availability. This work focuses on a specimen of such systems, the Hadoop Distributed File System (HDFS). A hot standby for the master node of HDFS is proposed in order to bring high availability to the system. The hot standby was achieved by (i) extending the master's state replication performed by its checkpointer helper, the Backup Node; and by (ii) introducing an automatic fail-over mechanism. Step (i) took advantage of the message duplication technique developed by other high availability solution for HDFS named AvatarNodes. Step (ii) employed ZooKeeper, a distributed coordination service. That approach resulted on small code changes, around 0.18% of the original code, which makes the solution easy to understand and to maintain. Experiments showed that the overhead implied by replication did not increase the average resource consumption of system nodes by more than 11% nor did it diminish the data throughput compared to the original version of HDFS. The complete transition for the hot standby can take up to 60 seconds on workloads dominated by I/O operations, but less than 0.4 seconds when there is predominance of metadata requisitions. Those results show that the solution developed on this work achieved the goals of producing a high availability solution for the HDFS with low overhead and short reaction time to failures.

Resumo

Projetistas de sistema geralmente optam por sistemas de arquivos baseados em *cluster* como solução de armazenamento para ambientes de computação de alto desempenho. A razão para isso é que eles provêm dados com confiabilidade, consistência e alta vazão. Porém a maioria desses sistemas de arquivos empregam uma arquitetura centralizada, o que compromete sua disponibilidade. Este trabalho foca especificamente em um exemplar de tais sistemas, o Hadoop Distributed File System (HDFS). O trabalho propõe um *hot standby* para o nó mestre do HDFS a fim de conferir-lhe alta disponibilidade. O *hot standby* é implementado por meio da (i) extensão da replicação de estado do mestre realizada por seu *checkpoint helper*, o Backup Node; e por meio da (ii) introdução de um mecanismo automático de *failover*. O passo (i) aproveitou-se da técnica de duplicação de mensagens desenvolvida por outra técnica de alta disponibilidade para o HDFS chamada Avatar Nodes. O passo (ii) empregou ZooKeeper, um serviço distribuído de coordenação. Essa estratégia resultou em mudanças de código pequenas, cerca de 0,18% do código original, o que faz a solução ser de fácil estudo e manutenção. Experimentos mostraram que o custo adicional imposto pela replicação não aumentou em mais de 11% o consumo médio de recursos pelos nós do sistema nem diminuiu a vazão de dados comparando-se com a versão original do HDFS. A transição completa para o *hot standby* pode tomar até 60 segundos quando sob cargas de trabalho dominadas por operações de E/S, mas menos de 0,4 segundos em cenários com predomínio de requisições de metadados. Estes resultados evidenciam que a solução desenvolvida nesse trabalho alcançou seus objetivos de produzir uma solução de alta disponibilidade para o HDFS com baixo custo e capaz de reagir a falhas em um breve espaço de tempo.

Agradecimentos

Ao meu pai Antonio Carlos, à minha mãe Bernadete e à minha irmã Cristiani pelo apoio.

Aos meus colegas Marco Aurélio Kohara, Guilherme Batista Leite e Felipe Wolff Ramos por terem dividido apartamento durante o tempo do mestrado e colaborado com meus estudos.

Ao Departamento de Computação da UFSCar - São Carlos, minha alma mater. Agradecimentos mais que especiais ao Professor Doutor Jander Moreira, meu orientador de iniciação científica, que me ensinou o que é desenvolver uma pesquisa em Computação.

Ao Instituto de Computação da Unicamp por ter-me aceito como mestrando, mesmo na condição de aluno em tempo parcial. Agradeço a honra ter recebido o conhecimento de pessoas tão destacadas em seus campos de pesquisa.

Ao Rodrigo Schmidt — mestre em Ciência da Computação por esse Instituto de Computação — por ter proposto o tema desta dissertação, ter-me orientado e colaborado com discussões técnicas no início do projeto.

À Motorola Industrial Ltda., minha empregadora no período de fevereiro de 2006 a agosto de 2012, por ter cedido quatro horas semanais para dedicar-me ao mestrado.

Aos membros da banca examinadora por suas prestimosas críticas e sugestões para o texto final desta presente dissertação. Ao Professor Doutor Alfredo Goldman vel Lejbman que prontamente aceitou avaliar meu trabalho. Ao Professor Doutor Luiz Eduardo Buzato e ao Professor Doutor Edmundo Roberto Mauro Madeira que acompanham este trabalho desde seus primórdios, tendo sido membros também da banca do Exame de Qualificação. E por fim ao Professor Doutor Gustavo Maciel Dias Viera por aceitar ser o suplente externo.

Por último, e portanto o agradecimento mais especial, à minha orientadora, a Professora Doutora Islene Calciolari Garcia. Muitíssimo obrigado por ter-me orientado, aconselhado, auxiliado, compartilhado minhas apreensões e dificuldades, dedicado-me tempo mesmo em face de compromissos familiares ou férias, revisado a exaustão meu textos e ter provido valiosos comentários e sugestões. Foste determinante para o êxito deste mestrado.

“A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable”

Leslie Lamport em um email de 28 de maio de 1987 para quadro de avisos do SRC DEC.

Sumário

Abstract	ix
Resumo	xi
Agradecimentos	xiii
1 Introdução	1
1.1 Contribuições	3
1.2 Uso de termos estrangeiros e convenções	4
1.3 Organização da dissertação	5
2 Fundamentos	7
2.1 Sistemas de arquivos distribuídos baseados em <i>cluster</i> centralizados	8
2.2 Alta disponibilidade em SADBcs centralizados	10
2.2.1 <i>Journaling</i>	10
2.2.2 Par ativo-passivo	11
2.2.3 Particionamento do espaço de nomes	12
2.2.4 Replicação de processos	14
2.3 Quadro comparativo	17
3 O Sistema de arquivos distribuído do Hadoop	19
3.1 Características	20
3.2 Espaço de nomes e blocos	21
3.3 Arquitetura	22
3.3.1 Cliente	22
3.3.2 DataNode	22
3.3.3 NameNode	24
3.3.4 Backup Node	27
3.4 Operações	28
3.4.1 Leitura	28

3.4.2	Escrita	29
3.4.3	Remoção	32
3.5	Escalabilidade	32
3.6	Considerações finais	33
4	Alta disponibilidade para o HDFS	35
4.1	Propostas de alta disponibilidade para o HDFS	36
4.1.1	Linux-HA e DRBD	37
4.1.2	Solução da IBM China Research	39
4.1.3	NameNode Cluster	41
4.1.4	Avatar Nodes	43
4.1.5	Solução oficial	45
4.1.6	UpRight HDFS	46
4.2	Quadro comparativo de soluções	48
5	O <i>Hot Standby Node</i>	51
5.1	Interações entre NameNode e Backup Node	52
5.1.1	Arquitetura do NameNode e Backup Node	52
5.1.2	Sincronização inicial e <i>checkpointing</i>	53
5.1.3	Propagação de mudanças no espaço de nomes	54
5.2	Construção do Hot Standby Node	56
5.2.1	Replicação do estado <i>soft</i>	56
5.2.2	Mecanismo automático de <i>failover</i>	60
5.2.3	Mudanças para captura de métricas	66
5.3	Arquitetura geral e ciclo de vida	66
5.4	Limitações do protótipo construído	67
5.5	Comparação com outras soluções	69
6	Experimentos	71
6.1	Ambiente experimental	72
6.2	Experimentos de desempenho	73
6.2.1	Programa de geração de carga de trabalho	74
6.2.2	Procedimento experimental	74
6.2.3	Coleta de dados	75
6.2.4	Resultados e análise	76
6.3	Experimentos de <i>failover</i>	91
6.3.1	Programa de geração de carga de trabalho	91
6.3.2	Procedimento experimental	92
6.3.3	Coleta de dados	93

6.3.4	Resultados e análise	93
6.4	Considerações finais	97
7	Conclusão e trabalhos futuros	99
7.1	Iterações	99
7.2	Projeto final e resultados	100
7.3	Trabalhos futuros	101

Lista de Tabelas

2.1	Técnicas de alta-disponibilidade usadas por diferentes SADB Cs centralizados.	17
3.1	Limites práticos do HDFS.	20
4.1	Quadro comparativo de propostas de alta disponibilidade para o HDFS. . .	50
5.1	Exemplos de tamanhos de mensagem para o HDFS 0.21 e para a solução Hot Standby Node.	59
5.2	Divisão do esforço de código em linhas.	66
6.1	Configurações do programa de geração de carga de trabalho para os cenários.	74
6.2	Passos do experimento de desempenho.	75
6.3	Duração média em segundos do período de análise.	76
6.4	Comparação entre o tráfego médio gerando por RPC e pelo ZooKeeper. . .	83
6.5	Número médio de execuções do coletor de lixo.	90
6.6	Intervalos para a análise do tempo de <i>failover</i>	94
6.7	Duração dos intervalos considerados para a análise do tempo de <i>failover</i> . .	94

Lista de Figuras

2.1	Arquitetura de SADBDBC centralizado.	8
2.2	Par ativo-passivo.	12
2.3	Divisão de espaço de nomes melhorando disponibilidade do sistema.	13
2.4	Exemplo de execução de replicação ativa sem falhas.	15
2.5	Exemplo de execução de replicação ativa com uma falha.	15
2.6	Exemplo de execução de replicação passiva sem falhas.	16
2.7	Exemplo de execução de replicação passiva com uma falha.	16
3.1	Arquitetura do HDFS.	23
3.2	Diagrama de sequência para a leitura de um arquivo de dois blocos no HDFS.	29
3.3	Distâncias na rede para o Hadoop.	30
3.4	Diagrama de sequência para a escrita de um arquivo de um bloco no HDFS	31
4.1	Solução de alta disponibilidade para o HDFS da ContextWeb.	37
4.2	Modos de sincronização da solução desenvolvida pela IBM China.	40
4.3	Arquitetura do NameNode <i>Cluster</i>	42
4.4	Arquitetura do Avatar Nodes.	44
4.5	Arquitetura do UpRight HDFS.	47
5.1	Diagrama arquitetural de blocos para o NameNode.	52
5.2	Sincronização inicial do Backup Node.	54
5.3	Propagação de alterações no espaço de nomes.	55
5.4	Possibilidades para replicar informações sobre blocos e DataNodes.	57
5.5	Exemplo de árvore do ZooKeeper.	61
5.6	Ciclo de vida do Failover Manager.	64
5.7	Arquitetura do Hot Standby Node.	67
5.8	Ciclo de vida da solução Hot Standby Node.	68
6.1	Tráfego médio na interface Ethernet dos DataNodes.	77
6.2	Tráfego médio na interface Ethernet dos DataNodes normalizado.	78
6.3	Tráfego RPC médio para o NameNode.	79

6.4	Tráfego RPC médio para o NameNode normalizado.	80
6.5	Tráfego RPC médio para o Backup Node e o Hot Standby Node.	81
6.6	Tráfego RPC médio para o Backup Node e o Hot Standby Node normalizado.	81
6.7	Consumo médio de CPU pelos DataNodes.	84
6.8	Consumo médio de CPU pelo NameNode.	84
6.9	Consumo médio de CPU pelo Backup Node e Hot Standby Node.	85
6.10	Uso médio de memória <i>heap</i> pelos DataNodes.	87
6.11	Uso médio de memória <i>non-heap</i> pelos DataNodes.	87
6.12	Uso médio de memória <i>heap</i> pelo NameNode.	88
6.13	Uso médio de memória <i>non-heap</i> pelo NameNode.	88
6.14	Uso médio de memória <i>heap</i> pelo Backup Node e Hot Standby Node.	89
6.15	Uso médio de memória <i>non-heap</i> pelo Backup Node e Hot Standby Node.	89
6.16	Vazão média de leitura e escrita para clientes.	91
6.17	Intervalos para a análise do tempo de <i>failover</i>	95
6.18	Tempo gasto para transição em relação ao número de blocos.	96

Capítulo 1

Introdução

Big Data, como é chamada pela indústria a geração e o processamento de gigantescos volumes de dados, já é uma realidade. Para corroborar tal afirmação podemos citar algumas estatísticas. Um estudo da Universidade da Califórnia São Diego [90] estimou que em 2008 os servidores de todo o mundo processaram mais de 10 milhões de petabytes. No mesmo ano o Google afirmou processar mais de 20 PB por dia em um artigo sobre seu *framework* de processamento distribuído, o MapReduce [32]. Neste mesmo ano ainda o CERN estimava que o LHC iria produzir 15 PB de dados por ano [109], porém a taxa real está atualmente em 25 PB por ano [14]. O Amazon S3 [7], o serviço de armazenamento da Amazon, observou um aumento no número de objetos armazenados de 2,9 bilhões em 2006 para 762 bilhões em 2011 [6]. Para finalizar a lista, o YouTube recebe a cada minuto 600 vídeos [113], os quais tem ser processados antes de serem disponibilizados na Web.

Todo esse universo de *Big Data* somente pode ser viabilizado se existirem sistemas de armazenamento escaláveis que forneçam dados de forma confiável a altas taxas. Na prática uma das soluções utilizadas são sistemas de arquivos distribuídos baseados em *cluster*. Nesses sistemas um arquivo é dividido entre vários nós de armazenamento, o que os habilita a escalar horizontalmente. Esse fato também permite-lhes fornecer dados com alta vazão pois arquivos podem ser lidos de vários nós simultaneamente. A maioria desses sistemas adota uma arquitetura mestre-escravos. Embora essa arquitetura simplifique a implementação e garanta bom desempenho, seu caráter centralizador fragiliza a disponibilidade do sistema. E conforme os sistemas de *Big Data* e de computação de alto desempenho migram de ambientes de processamento em lote para ambientes de tempo real 24x7, o quesito disponibilidade vem se tornando cada vez mais crucial.

Um dos expoentes dessa classe de sistema de arquivos é o Hadoop Distributed File System, mais conhecido por sua sigla HDFS [46, 92]. O HDFS faz parte do projeto Apache Hadoop [10], uma importante comunidade de software livre que tem por objetivo prover implementações de código aberto de *frameworks* para computação e armazenamento

confiáveis, escaláveis e distribuídos. O HDFS é o componente do Hadoop responsável pelo armazenamento de dados. Como sistema de arquivos é bastante escalável. O maior *cluster* HDFS em termos de armazenamento pertence ao Facebook e é capaz de armazenar 30 PB [66]. O Yahoo detém o recorde no quesito número de nós com um *cluster* HDFS de 4 mil servidores [91]. Métricas obtidas em *cluster* do Yahoo dão conta que o HDFS é capaz de fornecer uma vazão média de 60 MB/s para leitura e de 40 MB/s para escrita [91]. Assim como outros sistemas de arquivos baseados em *cluster* o HDFS adota uma arquitetura centralizada. O mestre do HDFS configura-se como um ponto único de falha por ser o ponto de entrada para clientes e o único nó a ter informações sobre quais arquivos compõem o sistema. Portanto não é de se estranhar que a comunidade Hadoop tenha iniciado discussões acerca de como melhorar a disponibilidade do HDFS [69].

Para contribuir com tal discussão descrevemos nas páginas que se seguem o resultado de uma pesquisa sobre como conferir alta disponibilidade ao HDFS. Trata de uma proposta de um *hot standby*, uma réplica do nó mestre do HDFS que pode em um curto espaço de tempo assumir a posição do mestre quando este falha. O protótipo baseia-se na versão 0.21 do HDFS e reutiliza o nó responsável por criar *checkpoints* para o mestre. Este nó é informado sobre toda alteração nos metadados dos arquivos de forma que lhe é possível reconstituir a árvore do sistema. Para complementar-lhe o estado, de forma a torna-se uma réplica, o *hot standby* assim formado passou a receber, assim como já acontecia com o mestre, mensagens de estado dos escravos. Para completar a solução, desenvolvemos um mecanismo capaz detectar a falha do mestre e iniciar o processo de transição. Essas decisões de projeto renderam uma implementação compacta de 1392 linhas que está disponível ao público como software livre.

A fim de validar a solução desenvolvida fizemos experimentos que pretendiam elucidar duas questões que devem ser respondidas por qualquer solução de alta disponibilidade: (i) qual o custo adicional imposto ao sistema original? (ii) qual o tempo de reação perante uma falha do mestre? Quanto a (i) os experimentos mostraram que tanto mestre quanto escravos não observaram aumentos superiores a 11% em nenhum dos três parâmetros básicos de análise de sistemas distribuídos: processamento, memória e rede. Quanto a (ii), obtivemos respostas diferentes de acordo com o tipo da carga a que o sistema estava submetido. Em cenário dominado por operações de leitura e escrita o *hot standby* pôde levar até 60 segundos para tratar sua primeira requisição após a detecção da falha do mestre. Em um cenário dominado por operações sobre metadados, como criação de arquivos e listagem de diretórios, esse intervalo pôde ser reduzido para menos de 0,4 segundos. Em todos os casos houve sobreposição entre os desvios para as vazões obtidas com a solução e o sistema original.

1.1 Contribuições

Além da presente dissertação a pesquisa produziu as seguintes contribuições:

- Relatório técnico sobre a então situação dos trabalhos envolvendo alta disponibilidade no HDFS:

André Oriani, Islene Calciolari Garcia, e Rodrigo Schmidt. The Search for a Highly-Available Hadoop Distributed Filesystem. Relatório Técnico IC-10-24, Instituto de Computação, Universidade Estadual de Campinas, agosto de 2010. Em Inglês, 27 páginas

O relatório contou com a participação de Rodrigo Schmidt, ex-aluno de Mestrado do Instituto de Computação-Unicamp e gerente de engenharia do Facebook. Rodrigo sugeriu o tema da pesquisa e colaborou com as discussões iniciais.

- Contribuições para o projeto Hadoop por meio de:
 - envio de *patches*:
 - * *Enhance the webUi to list a few of the corrupted files in HDFS.*
<https://issues.apache.org/jira/browse/HDFS-1031>
 - * *Extend DFSck with an option to list corrupt files using API from HDFS-729.*
<https://issues.apache.org/jira/browse/HDFS-1032>
 - relato de defeitos:
 - * *Backup Node fails when log is streamed due checksum error.*
<https://issues.apache.org/jira/browse/HDFS-2090>
 - * *Calling addEntry on a LedgerHandler obtained using openLedger does not throw an exception.*
<https://issues.apache.org/jira/browse/BOOKKEEPER-8>
- Contribuição para outros projetos de software livre.

O psutil [84] é um módulo Python para acessar informações de sistema e de processos em várias plataformas. Este módulo foi utilizado na coleta de métricas para a avaliação de desempenho. Submeteu-se um rascunho de *patch* para incorporar o *stolen time* (tempo de CPU cedido a outras máquinas virtuais) às APIs que lidam com tempo de CPU. O *patch* foi reaproveitado para a implementação do tíquete “Issue 361: Provide virtualization-related CPU stats on Linux” (<http://code.google.com/p/psutil/issues/detail?id=361>).

- Uma implementação em andamento do protocolo de *broadcast* atômico Zab. O Zab [55, 86] é usado pelo ZooKeeper [11], o coordenador distribuído do Hadoop. As funcionalidades que ainda necessitam ser codificadas são o registro das propostas aceitas e a recuperação de nós faltosos. Pretendia-se usar essa implementação para criar um protocolo de replicação que seria usado em uma linha de pesquisa que foi posteriormente abandonada.

O código se encontra disponível em <https://github.com/aoriani/zab>

- A implementação da solução de alta disponibilidade que é o objeto deste texto. O código está sob licença livre Apache 2 e se encontra disponível nos repositórios:

- <https://github.com/aoriani/hadoop-common>
- <https://github.com/aoriani/hadoop-hdfs>

- Apresentação do trabalho no VII WTD e consequente publicação do resumo expandido:

André Oriani e Islene Calciolari Garcia. A High Availability Solution for The Hadoop Distributed File System. Em *Anais do VII Workshop de Teses, Dissertações e Trabalhos de Iniciação Científica em Andamento do IC-UNICAMP*, páginas 6–7. Instituto de Computação, Universidade Estadual de Campinas, junho de 2012

- Artigo aceito no SRDS'12:

André Oriani e Islene Calciolari Garcia. From Backup to Hot Standby: High Availability for HDFS. Em *Proceedings of 31st International Symposium on Reliable Distributed Systems*, SRDS'12, páginas 131–140. IEEE Computer Society, outubro de 2012

O Qualis da Capes classificou o SRDS como uma conferência de nível A2 em 2012 [29]. O SRDS'12 recebeu 114 artigos dos quais apenas 28 foram considerados para o simpósio, resultando em uma taxa de aceitação de 25% [2].

1.2 Uso de termos estrangeiros e convenções

Tentou-se, na medida do possível, limitar o presente texto ao uso de verbetes da Língua Portuguesa, evitando neologismos ou anglicismos. Todavia, por ser um campo de pesquisa novo, nem todos os termos possuem alguma tradução consagrada. Por outro lado, mesmo em face da disponibilidade de termos portugueses com sinonímia, deu-se preferência ao termo inglês se o uso desse é o mais comum, o mais difundido. O racional é facilitar a

assimilação do texto, deixando-o acessível a pessoas de diferentes áreas que nem sempre estão familiarizados com a versão brasileira das palavras. Assim preferiu-se, por exemplo, usar *cluster*, *log* e *checkpoint* ao invés de agrupamento, registro e ponto de salvaguarda.

A nomenclatura usada para os elementos do HDFS obedece a convenção usada nos artigos publicados por engenheiros do Facebook e Yahoo [20, 92], a qual usa *UpperCamelCase*. A única discordância se dá no nomes das mensagens trocadas entre mensagens entre mestre e escravos. Os nomes dessas mensagens são palavras compostas que pela convenção são separadas por espaço. Preferiu-se separá-las com hífen para enfatizar a unidade do substantivo composto.

1.3 Organização da dissertação

O restante dessa dissertação se encontra organizado da seguinte forma:

- Capítulo 2 introduz sistemas de arquivos distribuídos baseados em *cluster* e discute a disponibilidade desses sistemas apresentando técnicas para melhorá-la;
- Capítulo 3 expõe o HDFS em detalhes;
- Capítulo 4 lista e descreve outras propostas que já foram feitas para conferir alta disponibilidade ao HDFS;
- Capítulo 5 delinea a solução desenvolvida, apontando e justificando decisões de projeto;
- Capítulo 6 apresenta e discute os experimentos juntamente com os resultados;
- Capítulo 7 explicita conclusões e trabalhos futuros.

Capítulo 2

Fundamentos

Sistemas de arquivos distribuídos, como bem definido por Coulouris et al. [30], são sistemas que *“permitem aos programas armazenarem e acessarem arquivos remotos exatamente como se fossem locais, possibilitando que os usuários acessem arquivos a partir de qualquer computador em uma rede”*.

Sistemas de arquivos distribuídos podem ser simétricos ou assimétricos. Nos sistemas simétricos os nós exercem funções similares sendo tanto responsáveis pelo acesso como pelo armazenamento de arquivos. É o caso de sistemas de arquivos baseados em tecnologias *peer-to-peer* tais como o Ivy [67] e o Oceanstore [57]. Porém o mais comum são sistemas assimétricos, os quais adotam uma arquitetura cliente-servidor com clientes acessando arquivos armazenados nos servidores. O exemplo clássico de tais sistemas de arquivos é o Sun Network File System (NFS) [88]. No NFS uma subárvore do sistema de arquivos local do servidor pode ser compartilhada com vários clientes.

Entretanto logo ficou claro que sistemas assimétricos como o NFS tinham limitações e não escalariam o suficiente para suprir a necessidade crescente de armazenamento. Nesse contexto surgiram os sistemas de arquivos distribuídos baseados em *cluster*. Os dados agora estão espalhados entre vários servidores que se apresentam ao cliente como um grande e único repositório de dados. Tais sistemas podem ser descentralizados ou centralizados. No primeiro caso clientes devem colaborar entre si, apoiando-se em sistemas de travas distribuídas para garantir a consistência de dados e *cache*. Os servidores de armazenamento são geralmente vistos como um grande disco virtual. Nessa linha podemos destacar os sistemas de arquivos Frangipani [97] e o xFS [9].

O segundo caso, sistemas centralizados, vem se tornando mais popular, principalmente após a publicação do artigo sobre o Google File System [41]. A popularidade também se deve ao sucesso de outros sistemas de arquivos comerciais e de código aberto em lidarem com as necessidades de ambientes de computação de alto desempenho e de processamento de grandes volumes de dados. Tais ambientes requerem sistemas de armazenamento es-

caláveis, confiáveis e que forneçam dados com boa vazão. Esses requisitos são preenchidos pelos sistemas de arquivos distribuídos baseados em *cluster* centralizados, os quais serão o objeto de estudo desse capítulo.

2.1 Sistemas de arquivos distribuídos baseados em *cluster* centralizados

Os sistemas de arquivos distribuídos baseados em *cluster* centralizados (SADBCs) adotam uma arquitetura mestre-escravos como pode ser apreciado na Figura 2.1.

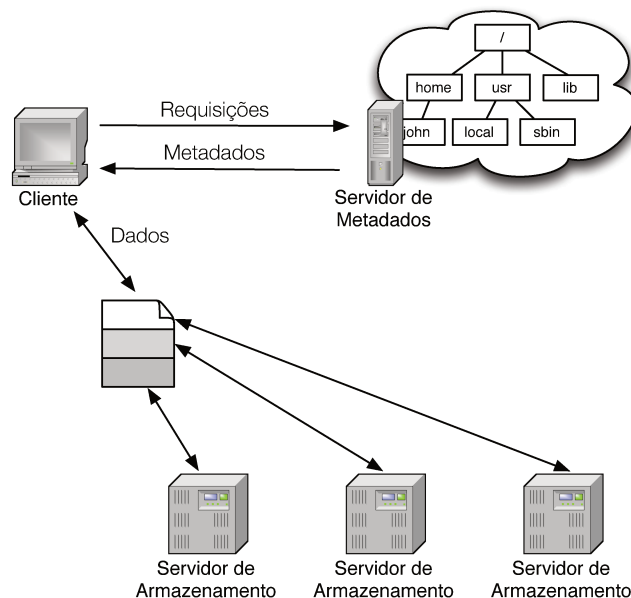


Figura 2.1: Arquitetura de SADBC centralizado.

Os servidores de armazenamento, os escravos dessa arquitetura, como o próprio nome evidencia, são responsáveis por armazenar os dados dos arquivos. O servidor de metadados, o mestre, é responsável por gerir o espaço de nomes e tratar as requisições de clientes. Também constam entre suas atribuições definir quais servidores de armazenamento receberão os dados de um determinado arquivo. Por ser único no sistema, o servidor de metadados também detém o poder de árbitro, controlando o acesso a arquivos e resolvendo conflitos entre clientes, garantindo assim a consistência do sistema de arquivos. Essa divisão de papéis também colabora com a escalabilidade, pois basta adicionar mais servidores de armazenamento para aumentar a capacidade do sistema.

Essa arquitetura também é uma arquitetura de sistema de arquivos paralelos. Os

arquivos são divididos em tiras (*strips*) que são distribuídas entre os servidores de armazenamento usando esquemas RAID [80]. Outra opção é dividir os arquivos em blocos de tamanhos iguais, os quais são distribuídos e replicados entre vários servidores de armazenamento. Nesta opção *checksums* são usados para detectar corrupção de blocos. Em ambos os casos clientes poderão ler simultaneamente de vários servidores de armazenamento gerando um grande vazão agregada. Outra vantagem dessa estratégia é que ela contribui para a alta disponibilidade dos dados. Em caso de corrupção ou perda, dados podem ser reconstruídos por meio de técnicas RAID ou a partir de uma réplica saudável do bloco.

Ainda no tocante a armazenamento dos blocos ou tiras, embora sejam sistemas de arquivos, os SADCs não trabalham com entidades de baixo nível tais como blocos ou setores de disco. O armazenamento de blocos e tiras é delegado ao sistema de arquivos local dos servidores de armazenamento ou a dispositivos de armazenamento baseados em objetos [73]. No primeiro caso os blocos ou tiras são armazenados simplesmente como arquivos locais. No segundo caso os dispositivos de armazenamento baseados em objetos provêm uma interface de acesso de alto nível baseada em objetos (contêineres de dados de tamanho flexível) deixando o gerenciamento do espaço e armazenamento de baixo nível para o dispositivo.

Este projeto de sistema de arquivos distribuído promove o desacoplamento entre o espaço de nomes e os dados. Ao mesmo tempo em que os clientes se comunicam com o servidor de metadados para operações sobre metadados, eles são livres para contactar diretamente os servidores de armazenamento para ler e escrever dados. As operações sobre metadados são interações curtas, ao mesmo passo que transferências de dados podem tomar um tempo considerável. Se o servidor de metadados fosse responsável tanto pelo gerenciamento de metadados como pela transferência de dados aos clientes, esta última dominaria todo o tempo de resposta do servidor tornando-o um gargalo para um ambiente altamente distribuído. Essa divisão faz com as transferências de dados sejam distribuídas pelo servidores de armazenamento, utilizando-se a vazão de todo o *cluster*, gerando ganhos de desempenho e de escalabilidade.

O fato de ser uma arquitetura centralizada simplifica a implementação e garante bom desempenho já que não é necessário comunicar-se com outros nós e atingir consenso sobre o resultado de uma determinada operação. Porém arquiteturas centralizadas também são conhecidas por problemas de disponibilidade. O servidor de metadados, por ser o ponto de entrada para os clientes e o único nó a deter informações sobre quais arquivos compõem o sistema, configura-se como um ponto único de falha. Sua queda significa a queda de todo o sistema de arquivos, visto que clientes não terão como fazer suas solicitações.

2.2 Alta disponibilidade em SADCs centralizados

Como vimos na seção anterior, a alta disponibilidade de dados em SADCs centralizados é conseguida por meio de técnicas RAID e replicação de blocos. No caso de replicação de blocos *checksums* são empregados para garantir a consistência e confiabilidade dos dados. Já no tocante à disponibilidade do sistema como um todo, vimos que ela é comprometida pela presença de um ponto único de falha, o servidor de metadados.

Neste seção veremos que técnicas são empregadas por importantes SADCs centralizados para melhorar a disponibilidade do sistema. Tratam-se de SADCs utilizados em aplicações reais de computação de alto desempenho e de processamento distribuído de grande volumes de dados. A maioria desses sistemas iniciou-se como projetos de pesquisa da academia e acabaram se tornando produtos comerciais ou projetos de código aberto. Os sistemas tratados aqui serão:

- **Ceph** [64, 104] foi inicialmente criado durante o doutorado de Sage Will [103] em 2007 e hoje é um projeto de código aberto com suporte comercial da Inktank Storage, Inc. [25].
- **Google File System (GFS)** [41] é o sistema de arquivos utilizado pelo Google para armazenar os grandes volumes de dados gerados por suas aplicações. É a base sobre a qual opera o BigTable [26], um banco de dados orientado a colunas, e o MapReduce [32], o *framework* de processamento distribuído para grande conjuntos de dados.
- **Lustre** [62, 63, 96] iniciou-se como um projeto de pesquisa de Peter Braam na Carnegie Mellon University em 1999. Entre 2007 e 2010 a Sun Microsystems vendia o Lustre como parte de seus sistemas de computação de alto desempenho. Hoje o Lustre tem seu código disponível e mantido pela OpenSFS [74].
- **Panasas** [68, 106] é um sistema de arquivos distribuído escalável desenvolvido e comercializado por uma empresa de mesmo nome.
- **PVFS2** O Parallel Virtual File System [24] foi criado em 1993 por Walt Ligon e Eric Blumer como um sistema de arquivos paralelo para um projeto financiado pela NASA. O PVFS2 [79] é um versão reescrita do zero do PVFS para deixá-lo mais modular, flexível e extensível.

2.2.1 *Journaling*

Uma técnica bastante empregada em sistema de arquivos de disco para garantir recuperação após falhas é *journaling* [93]. Nessa técnica alterações no sistema de arquivos são

primeiramente aplicada às estruturas de dados em memória e registradas sequencialmente em um *log*. Uma vez escrita para o *log*, a operação é tida como efetivada e o controle pode voltar para o programa de usuário que a requisitou. Em paralelo as entradas do *log* são aplicadas às estruturas de dados do sistema de arquivos em disco e removidas do *log*. As escritas sequências no *log* são mais rápidas que as escritas aleatórias nas estruturas em disco. Com isso o sistema de arquivos consegue aumentar sua vazão de operações. Em caso de falha podem restar ainda algumas entradas no *log*. Durante a recuperação estas entradas serão aplicadas as estruturas de dados do sistema de arquivos com o fito de que ele possa voltar ao seu último estado consistente anterior à falha. SADBcs também usam essa técnica para assegurar sua capacidade de recuperação perante quedas do servidor de metadados. Os servidores de metadados registram toda alteração no espaço de nomes para um *log* em arquivo. Por questões de confiabilidade o *log* é gravado para vários discos, incluindo discos remotos.

Alguns SADBcs como o GFS mantêm as estruturas de dados do sistema apenas em memória principal por questões de desempenho. Nesse caso as entradas do *log* não são removidas. O *log* torna-se vital para reconstituir todo o sistema de arquivos. Para que ele não se torne longo causando um demasiado tempo de reinício do sistema, o GFS cria *checkpoints* — versões seriadas para arquivo de suas estruturas de dados em memória. Desta forma, os *logs* do GFS representam deltas de alterações no espaço de nomes desde o último *checkpoint*. Ao iniciar, o servidor do GFS carrega o último *checkpoint* e aplica as entradas do último *log* a fim de reconstituir a árvore de arquivos e diretórios.

2.2.2 Par ativo-passivo

O servidor de metadados ativo pode gravar suas estruturas de dados e *log* para um dispositivo de armazenamento compartilhado com outro nó, o servidor passivo (Figura 2.2(a)). Um serviço de monitoramento pode detectar a falha do servidor de metadados corrente e iniciar um novo no servidor passivo (Figura 2.2(b)). Esse novo servidor de metadados se utilizará das estruturas de dados e *logs* do armazenamento compartilhado para recuperar o estado do sistema de arquivos. Clientes serão redirecionados para o novo servidor.

O servidor passivo pode ser mantido ocioso ou, como no caso do Lustre, ser o servidor de metadados de outro sistema de arquivos. No que tange ao serviço de monitoramento, ele pode ser interno próprio ou, como no caso do PVFS2 [1], usar um software externo como o gerenciador de recursos de *cluster* Heartbeat do projeto Linux-HA [47]. PVFS2 usa Heartbeat em conjunto com STONITH [95]. STONITH é um software voltado para prevenir o acesso (*fencing*) de recursos compartilhados por meio do desligamento do nó suspeito de falha. Isso evita o cenário em que ambos os nós pensem estar ativos e tentem simultaneamente escrever para o dispositivo de armazenamento compartilhado, corrompendo-o.

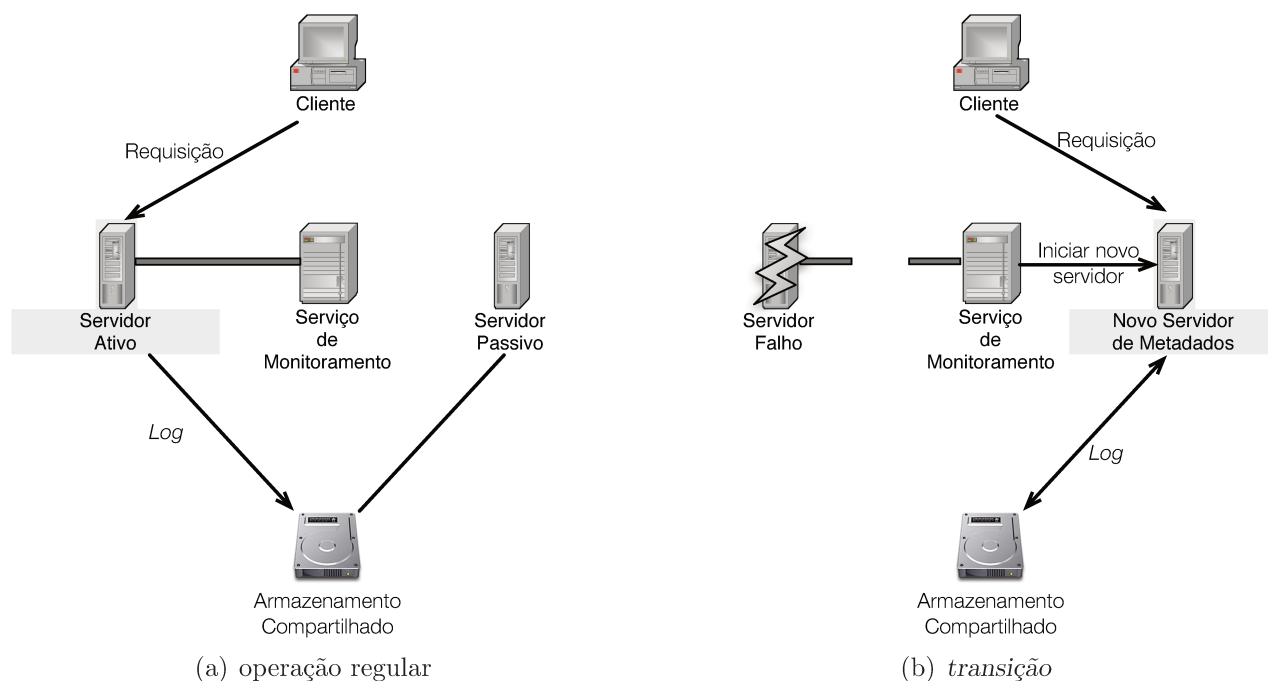


Figura 2.2: Par ativo-passivo.

O redirecionamento do cliente pode se feito atuando-se na rede ou por meio de algum serviço que forneça informações sobre o servidor de metadados atual. No GFS clientes acessam o servidor de metadados por meio de um *alias* DNS que pode ser ajustado para o servidor corrente. O PVFS2 transfere o IP do servidor falho para o novo servidor. Clientes do Panasas e do Lustre devem consultar um serviço de configuração para saber qual é o endereço do servidor de metadados. O serviço de configuração do Panasas é um banco de dados replicado entre vários servidores por meio de Paxos [59].

2.2.3 Particionamento do espaço de nomes

Conforme o número de arquivos cresce, chegar-se-á a um nível no qual um único servidor não será suficiente para armazenar todos os metadados do sistema de arquivos. A solução será distribuir o espaço de nomes entre dois ou mais servidores de metadados. Embora o objetivo primordial do particionamento do espaço de nomes seja aumentar a escalabilidade, não se pode negar que ele melhora a disponibilidade do sistema de arquivos. O particionamento cria domínios independentes de falha. A queda de um servidor de metadados deixa agora apenas *parte* da árvore de arquivos e diretórios indisponível.

Por exemplo, no Panasas e no GFS o sistema de arquivos está dividido em volumes. Cada volume é mantido por um servidor de metadados e é de fato um sistema de arquivos

independente. Os servidores de armazenamento podem ser compartilhados entre os vários volumes. Os clientes veem os volumes como subdiretórios da raiz de um grande sistema de arquivos virtual. Assim se na Figura 2.3 o servidor de metadados 1 falhar, todos os arquivos debaixo de `/bar` ainda continuam acessíveis.

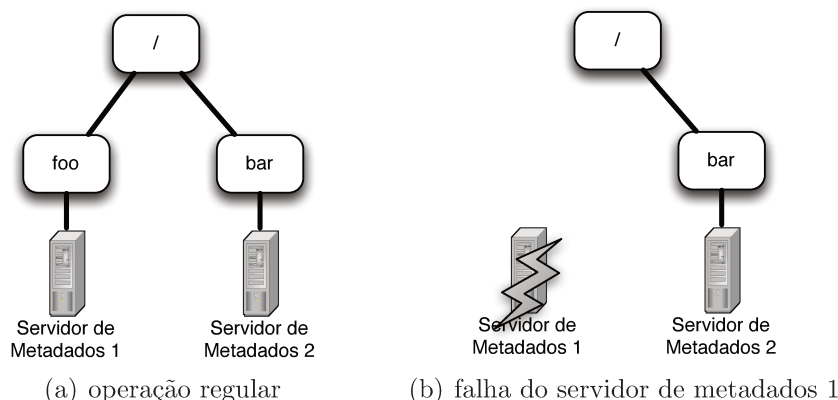


Figura 2.3: Divisão de espaço de nomes melhorando disponibilidade do sistema.

No PVFS2 quando um novo arquivo é criado, um servidor de metadados é escolhido aleatoriamente para armazenar seu *inode*. A seguir, uma entrada no diretório pai do arquivo, que pode estar em outro servidor de metadados, é criada para apontar para o *inode* do novo arquivo. Desta forma, quando o cliente deseja ler o arquivo ele deve olhar por cada segmento de seu caminho completo, do diretório raiz ao seu diretório pai, indo de servidor de metadados a servidor de metadados, até encontrar os metadados do arquivo desejado. O cliente faz *cache* de entradas de diretório e suas localizações para evitar ter de contactar vários servidores para acessar um único arquivo. Todas as entradas de um mesmo diretório ficam armazenadas em um mesmo servidor de metadados.

Ceph vai mais além com a divisão do espaço de nomes. Em Ceph o espaço hierárquico de nomes do sistema de arquivos é particionado em subárvores disjuntas de forma dinâmica e adaptativa entre um conjunto de servidores de metadados de acordo com uma estratégia chamada Particionamento Dinâmico de Subárvores [105]. Esta estratégia move subárvores entre servidores com o objetivo de uniformizar a carga de trabalho. No caso de subárvores frequentemente acessadas pode ocorrer a sua replicação entre vários servidores de metadados. Nessa situação uma das réplicas se torna a autoridade para aquela subárvore. Ela fica responsável por executar as operações que alterem os metadados e por avisar as demais réplicas sobre atualizações. O fato de o particionamento ser em subárvores disjuntas preserva a localidade. Para descobrir que subárvores são populares, os diretórios são dotados de um contador de acesso. Quando um arquivo é acessado, todos os diretórios de seu caminho tem seu contador de acesso incrementado. Os contadores decaem expo-

nencialmente com o tempo para refletir uma diminuição na frequência de acesso. Assim as subárvores populares são aquelas cujo diretório raiz possui um contador de acesso com um valor elevado. Esses contadores são trocados entre os nós, de forma nós sobrecarregados podem identificar nós ociosos e transferir-lhes a autoridade de porções populares do espaço de nomes.

2.2.4 Replicação de processos

Do ponto de vista de hardware, a máquina que hospeda o processo de servidor de metadados deve ser robusta e empregar redundância onde possível e viável: fontes de energia redundantes, enlaces redundantes, etc. Redundância é a técnica chave para tolerância a falhas e alta disponibilidade. Com redundância se um dos componentes redundantes falha, os demais podem continuar o trabalho em seu lugar, gerando o mínimo de interrupção ao serviço. Redundância também pode ser aplicada a software por meio da replicação de processos e seus estados. Existem duas técnicas seminais de replicação de processos: a replicação ativa e a passiva.

Replicação ativa

Na replicação ativa [30, 43] ou replicação por estado de máquina [89], os processos replicados funcionam como máquinas de estado deterministas, ou seja, o estado atual é determinado única e exclusivamente por um estado inicial e uma sequência de entradas. Desta forma, se o processos replicados recebem a mesma sequência de entradas eles atingirão estados idênticos em algum momento.

Assim, a replicação ativa requer que os clientes enviem suas requisições a todas as réplicas por meio de *broadcast* totalmente ordenado. O *broadcast* totalmente ordenado impõe uma ordem total às requisições e garante que todas as réplicas corretas as recebam e as recebam em ordem. Consequentemente todas as réplicas processam as requisições e devolvem uma resposta ao cliente como pode ser observado na Figura 2.4, na qual Σ_1 representa o estado antes do processamento da requisição e Σ_2 depois. O cliente pode usar a primeira reposta que recebe no caso de um modelo de falhas por queda (o processo para de funcionar, mas estava correto até parar), ou aquela que atingir uma maioria entre as repostas recebidas no caso de um modelo de falhas bizantinas (o processo produz repostas arbitrárias em momentos arbitrários).

A grande vantagem da replicação ativa se revela na presença de falhas, como pode ser vista na Figura 2.5. Mesmo com a falha da réplica 2, o sistema pode ainda atingir progresso com as réplicas corretas restantes. Não só isso, a falha foi totalmente transparente para o cliente que nem sequer percebeu alguma latência adicional.

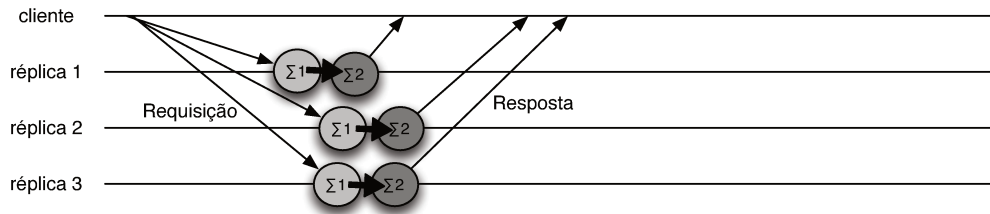


Figura 2.4: Exemplo de execução de replicação ativa sem falhas.

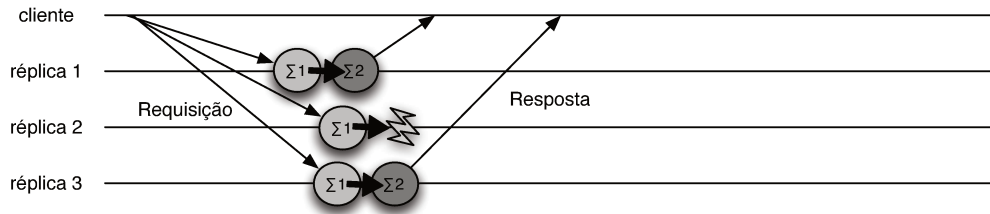


Figura 2.5: Exemplo de execução de replicação ativa com uma falha.

Uma desvantagem da replicação ativa é que por requerer processos deterministas ela restringe o uso de múltiplos fluxos de execução (*multithreading*) e outros mecanismos que podem trazer indeterminismos, mas que podem melhorar o desempenho.

O requisito de ordem total pode ser relaxado se as requisições dos clientes forem comutativas ou idempotentes. Para uma sistema f -tolerante a falhas, são necessárias $f + 1$ réplicas para o modelo de falha por queda. Para o modelo bizantino é necessário $2f + 1$ réplicas a fim que haja sempre uma maioria de pelo menos $f + 1$ para alguma das respostas.

Replicação passiva

A replicação passiva [30], também conhecida como *primary-backup* [23, 43], emprega um modelo assimétrico. Uma das réplicas, a primária, é responsável por atender as requisições dos clientes. Porém, como pode ser visto na Figura 2.6, a réplica primária (réplica 1) envia uma mensagem de atualização de estado para as outras réplicas (os *backups*) antes de devolver alguma resposta ao cliente. O objetivo dessa mensagem de atualização é fazer com que os *backups* atinjam o mesmo estado em que a réplica primária se encontra após a execução da requisição do cliente. Quando todos os *backups* corretos respondem com uma mensagem de reconhecimento é que então a réplica primária envia uma resposta ao cliente.

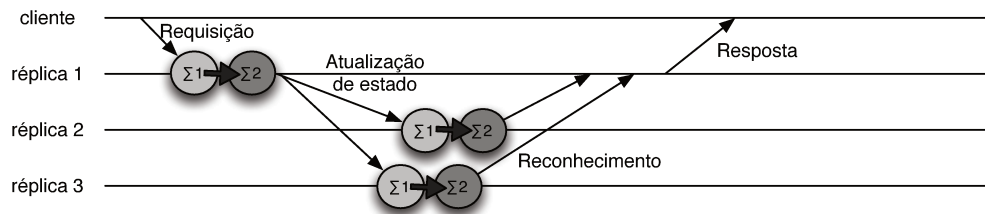


Figura 2.6: Exemplo de execução de replicação passiva sem falhas.

Visto que apenas uma das réplicas processa as requisições, uma ordenação global de todas as requisições é facilmente estabelecida. E como também o estado final dos *backups* é ditado pela réplica primária, esta pode se utilizar de múltiplos fluxos de execução (*multithreading*) e outros elementos não deterministas para o processamento das requisições, embora os *backups* devam processar as mensagens de atualização de forma determinista para garantir a consistência entre as réplicas. Devido ao fato de uma única réplica processar as requisições, a replicação passiva usa menos recursos de processamento do que a replicação ativa.

Porém umas das desvantagens da replicação passiva se revela em um cenário de falhas como o mostrado na Figura 2.7. Quando a réplica primária falha (réplica 1), os *backups* corretos devem organizar uma eleição para eleger uma nova réplica primária (réplica 2). Esta falha não é transparente para o cliente, o qual terá que refazer a requisição. Ao período em que o sistema fica momentaneamente indisponível, entre a falha da réplica primária e a escolha da nova réplica primária, dá-se o nome de *failover*. Quando um dos *backups* falha, não há impacto para o cliente, porém a réplica primária deve saber desse fato para evitar a espera por um reconhecimento do *backup* faltoso. Portanto para replicação passiva é imprescindível o emprego de um protocolo de gerenciamento de grupo, o que torna a implementação da replicação passiva tão complexa quanto o *broadcast* totalmente ordenado demandado pela replicação ativa.

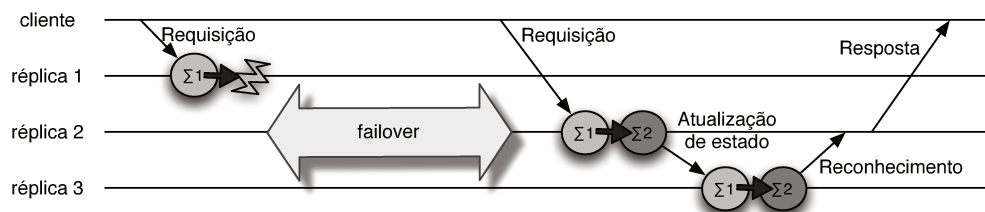


Figura 2.7: Exemplo de execução de replicação passiva com uma falha.

Esse modelo não tolera falha bizantinas, porém com $f + 1$ réplicas ele pode sobreviver até f falhas por queda. Os *backups* podem ser usados para tratar requisições de somente-leitura.

O GFS implementa algo na linha de replicação passiva. O GFS conta com réplicas do servidor de metadados que provêm acesso somente leitura ao sistema de arquivos. O acesso pode ocorrer mesmo quando o servidor de metadados primário não está em funcionamento. Essas réplicas são chamadas “*shadows*” pois elas se atualizam a partir do *log*, o que pode impor uma defasagem em relação à réplica primária. Tanto a réplica primária como as *shadows* consultam os servidores de armazenamento ao iniciar para descobrirem onde estão as réplicas dos blocos.

2.3 Quadro comparativo

A Tabela 2.1 relaciona as técnicas com os sistema de arquivos apresentados aqui. Antecipando-nos ao Capítulo 3, incluímos também o Hadoop Distributed File System.

Sistemas de Arquivos	Journaling	Par Ativo-Passivo	Particionamento do Espaço de Nomes	Replicação
Ceph	✓	✓	Dinâmico	Somente-Leitura Passiva ³
GFS	✓	✓	Estático	
HDFS	✓		Estático ²	
Lustre	✓	✓ ¹		
Panasas	✓	✓	Estático	
PVFS2		✓ ¹	Aleatório	

¹Apenas com suporte de software externo.

²A partir de HDFS-0.23.

³A partir de HDFS-2.0.0-alpha.

Tabela 2.1: Técnicas de alta-disponibilidade usadas por diferentes SADCs centralizados.

Capítulo 3

O Sistema de arquivos distribuído do Hadoop

Em 2002 Mike Cafarella and Doug Cutting iniciaram o projeto Apache Nutch. Este projeto tinha por objetivo implementar um motor de busca para Web de código aberto. Quando o sistema começou a tomar forma, eles logo perceberam que sua arquitetura não escalaria para os bilhões de páginas disponíveis na Web. Entretanto, providencialmente em 2003, engenheiros de software do Google publicaram o artigo intitulado *The Google File System* [41]. O artigo descrevia um sistema de armazenamento para o grande volume de dados gerados nos *data centers* da empresa. Mike e Doug perceberam então que o sistema de arquivos criado pelo Google resolveria o problema de armazenar os grandes arquivos gerados pelo *web crawler* do Nutch, bem como diminuiria o tempo gasto em tarefas administrativas como gerenciar nós de armazenamento. Inspirados nesse artigo, Mike e Doug começaram então a escrever em 2004 o Nutch Distributed File System (NDFS). Em 2006 os desenvolvedores de Nutch, vislumbrando que a combinação do NDFS com uma implementação livre de MapReduce [32] — um modelo de programação distribuída para o processamento de vastos conjuntos de dados — teria potencial para aplicações além de máquinas de busca, iniciaram um novo projeto chamado Apache Hadoop. A partir daquele momento o NDFS teve seu nome mudado para o atual: Hadoop Distributed File System (HDFS).

O HDFS [22, 46, 92, 108] é um sistema de armazenamento confiável e escalável. Ele provê arquivos com alta consistência, vazão e disponibilidade. Essa afirmação é ratificada pela Tabela 3.1, na qual são listados limites práticos do HDFS obtidos de medidas em *clusters* de produção.

Dentro do projeto Apache Hadoop, o HDFS é usado não somente pelo Hadoop MapReduce, como também pelo HBase [45], um banco de dados distribuído versionado e orientado a colunas. Existem relatos sobre o HDFS sendo usado como sistema de ar-

Quesito	Limite
Capacidade de armazenamento	30 PB [66]
Tamanho do <i>cluster</i>	4 mil nós [91]
Número de arquivos	150 milhões [20]
Números de clientes	15 mil clientes [91]
Vazão média de leitura	66 MB/s [91]
Vazão média de escrita	40 MB/s [91]

Tabela 3.1: Limites práticos do HDFS.

mazenamento de propósito geral por aplicações que variam desde servidores multimídia a redes de sensores [18]. Cientistas do CERN usam HDFS para armazenar extensos conjuntos de resultados de experimentos [12].

Neste capítulo descrevemos a versão 0.21 do HDFS, a qual foi objeto desse trabalho. Referências a versões posteriores poderão ser feitas apenas a título informativo e com vistas a situar o trabalho desenvolvido aqui no contexto atual do desenvolvimento do HDFS.

3.1 Características

Assim como outros sistemas de arquivos distribuídos baseados em *cluster*, o HDFS é caracterizado por desacoplar os metadados dos dados. A principal motivação para isso é a escalabilidade do sistema. Enquanto operações sobre metadados costumam ser rápidas, transferências de dados podem tomar um tempo significativo. Se a operação combinada fosse executada por um único servidor como no NFS [88], a parte da transferência de dados iria dominar o tempo de resposta do servidor, tornando-o um gargalo para um sistema altamente distribuído. Assim, operações sobre metadados são concentradas em um único servidor de metadados, enquanto as transferências de dados são repartidas entre os servidores de armazenamento, aproveitando-se desta forma da vazão de todo o *cluster*.

Clusters HDFS são geralmente compostos de dezenas a milhares de nós feitos do que a literatura convencionou chamar de *commodity hardware* — equipamento comum, de custo acessível, comercializado por diversos fabricantes. Esse grande número de elementos desprovidos de grande robustez interconectados entre si cria uma probabilidade não trivial de falhas. De fato, nos *clusters* de produção do Yahoo a taxa de falha é de 2 a 3 nós por dia em um grupo de mil. Nós recém saídos da fábrica [91] apresentam uma taxa de falha três vezes maior. Consequentemente, o HDFS tem de tratar falhas como a norma ao invés de exceção. Portanto o sistema trabalha ativamente para detectá-las. Para isso, ele se utiliza de *checksums* para detectar corrupção de dados e de replicação para se recuperar

quando de sua ocorrência. Além disso nós constantemente trocam mensagens entre si para descobrir quedas.

Projetado para dar apoio ao Hadoop MapReduce, o HDFS é voltado para aplicações que necessitam processar grande volumes de dados. Nesse tipo de aplicação o conjunto de dados é normalmente gerado ou copiado de uma fonte e então várias análises são realizadas. Essas análises envolvem a leitura de boa parte, senão de todo o conjunto de dados. Portanto, para o tempo total da computação é mais importante o tempo gasto em ler todo o conjunto de dados do que o tempo gasto para ler o primeiro registro. Desta forma, o HDFS foca-se em prover alta vazão de dados em detrimento da latência de acesso. Isto é conseguido com o relaxamento de algumas semânticas do padrão POSIX [81] e um modelo de coerência simples. Este modelo de coerência permite que um arquivo tenha vários leitores, porém apenas um escritor. Ele também proíbe escrita aleatórias, permitindo que elas sejam feitas somente ao final do arquivo. Esse conjunto de restrições simplificam o controle de acesso aos arquivos e reduzem as disputas por travas.

Outros aspectos importantes do HDFS são que ele é um sistema de arquivos codificado na linguagem de programação Java [54] e que executa no espaço de usuário. Isto permite que ele seja empregado em plataformas de hardware e software heterogêneos.

3.2 Espaço de nomes e blocos

O HDFS possui um espaço de nomes hierárquico organizado como uma árvore de arquivos e diretórios. Arquivos e diretórios são representados por estruturas semelhantes a *inodes*, os quais guardam atributos tais como nome, permissões de acesso, dono, grupo, fator de replicação, tempos de acesso e de modificação. Inclusive *links* simbólicos e algum tipo de cotas são implementados.

Assim como em sistemas de arquivos de disco, arquivos no HDFS são compostos por blocos de tamanho fixo — a unidade básica para transferências. Porém, diferentemente de seus equivalentes para discos, blocos no HDFS são grandes, da ordem de 64 MB e 128 MB. A razão por trás disso é minimizar o custo de recuperação de blocos (*seek time*). Se o tamanho do bloco é tal que sua transferência a partir do disco toma um tempo consideravelmente maior que a latência para ler seus primeiros bytes, a transferência do arquivo pode ocorrer na taxa de transferência do disco, garantindo assim boa vazão. Este tamanho de bloco grande torna o armazenamento de arquivos pequenos custoso. Entretanto, o tamanho típico de arquivos encontrados no HDFS é da magnitude de gigabytes, senão terabytes. A abstração de blocos implica em outros benefícios. Ela permite que um arquivo seja maior que qualquer disco no sistema, uma vez que os blocos de um arquivo podem ser espalhados entre vários discos. Blocos também simplificam o subsistema de armazenamento do HDFS pois por serem de tamanho fixo permitem determinar de

antemão quantos deles caberão em um certo disco.

Blocos no HDFS são replicados como medida básica para garantir alta disponibilidade de dados e tolerância a falhas. Se uma réplica se torna corrompida ou indisponível, outra pode ser usada em seu lugar. O fator de replicação é definível por arquivo, sendo por padrão três. Outra técnica usada por sistemas de arquivos distribuídos para oferecer as mesmas garantias é RAID [80]. Em RAID 0, a técnica mais comum para aumentar desempenho, as operações de leitura e escrita são limitadas pela velocidade do disco mais lento do arranjo de discos. Já na replicação as operações de disco são independentes, portanto em média elas são mais rápidas que o disco mais lento envolvido no processo. Replicação tem também a vantagem de aliviar o fardo de leitura, visto que leituras de um mesmo arquivo podem ser feitas a partir de diferentes réplicas.

3.3 Arquitetura

O HDFS possui uma arquitetura mestre-escravos. Um único nó mestre, o NameNode, é responsável por gerenciar o espaço de nomes e blocos. O NameNode controla outros vários nós servidores, denominados DataNodes, que são incumbidos de armazenar as réplicas dos blocos. O Backup Node é outro nó servidor subserviente ao NameNode que tem por função produzir *checkpoints*.

Toda a comunicação entre os elementos é feita por meio de um protocolo de chamada de procedimento remoto (RPC) próprio. O protocolo é baseado em TCP/IP e conta com seu próprio mecanismo de empacotamento para tipos de dados não primitivos. Os elementos do HDFS serão detalhados a seguir e podem ser vistos na Figura 3.1 juntamente o fluxo de dados que são trocados entre eles.

3.3.1 Cliente

O cliente é a interface do HDFS com outros sistemas. Embora ele apresente uma API muito próxima do POSIX [81], aplicativos que desejem se utilizar do HDFS devem usar uma biblioteca especializada, a qual abstrai a comunicação com o *cluster*. Todas as requisições geradas pelo cliente são encaminhadas ao NameNode. Entretanto, para que esse nó não se torne um gargalo do sistema ele apenas processa operações nos metadados. Todas as transferências de dados são realizadas diretamente entre clientes e DataNodes. Aliás, o cliente usa *caches* locais para melhorar o desempenho de entrada e saída.

3.3.2 DataNode

Os DataNodes são nós servidores que cumprem duas funções:

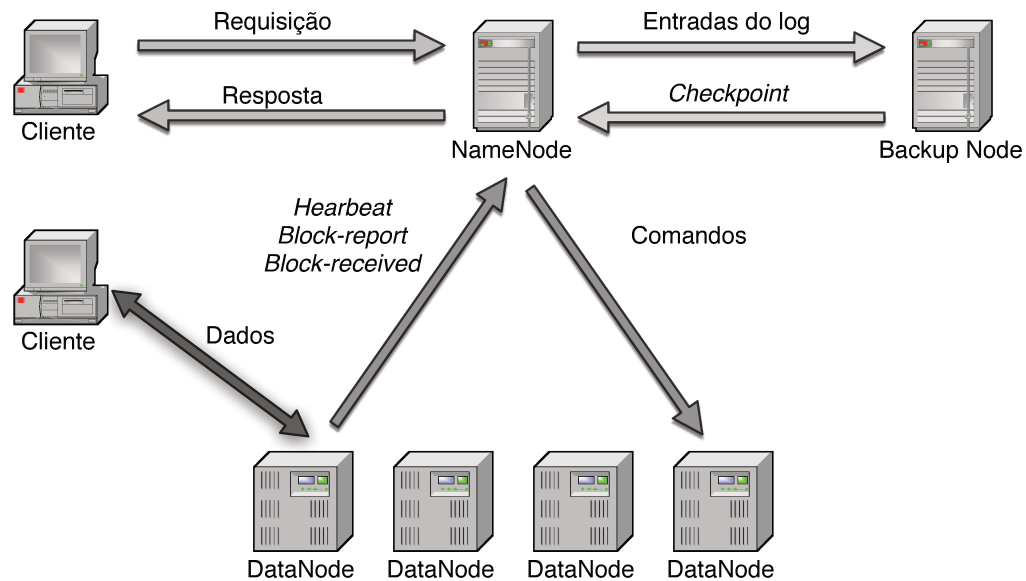


Figura 3.1: Arquitetura do HDFS.

- armazenar réplicas de blocos;
- fornecer e receber blocos provenientes dos clientes.

Os DataNodes guardam um bloco usando dois arquivos locais. O primeiro armazena os dados propriamente ditos e é tão grande quanto o conteúdo do bloco. O segundo arquivo armazena *checksums* para cada seção de 512 bytes do bloco. Como o algoritmo usado é o CRC-32 [21], apenas quatro bytes são usados por *checksum*, gerando um custo adicional de armazenamento de menos de 1%. Os DataNodes apenas conhecem os identificadores dos blocos que armazenam. É ignorado a que arquivos eles pertencem.

Após terminar sua iniciação, o DataNode se registra com NameNode e a partir desse momento os dois passam a trocar uma série de mensagens entre si que visam passar ao NameNode um visão sobre o estado do DataNode. Tais mensagens podem ser:

Heartbeat As mensagens de heartbeat servem para indicar para o NameNode que o DataNode ainda está ativo. Elas também são usadas para passar informações gerais sobre o estado do DataNode, tais como a capacidade total e usada de armazenamento e o número de transferências de dados em curso. Essas informações auxiliam o NameNode em suas decisões sobre alocação de espaço e balanceamento de carga do sistema. O NameNode utiliza as respostas a essas mensagens para enviar instruções aos DataNodes. Essas instruções pode ser:

- registrar-se novamente;
- replicar blocos para outros DataNodes;
- remover réplicas;
- desligar-se;
- e enviar um novo block-report (a ser detalhado a seguir).

Por padrão, as mensagens de heartbeat são enviadas a cada três segundos. DataNodes cujos heartbeats não são ouvidos há mais de dez minutos são considerados inativos e os blocos que eles armazenavam são dados como perdidos. Este aparente descompasso entre a frequência dos heartbeats e o *timeout* tem sua razão. Como os comandos do NameNode pegam carona nas resposta dos heartbeats, eles devem ser frequentes para garantir a capacidade do NameNode de reagir rapidamente a eventos no *cluster*. Por outro lado o valor conservador de 10 minutos visa evitar detecção incorreta de falhas, o que causaria replicação desnecessária de dados, impactando a capacidade de armazenamento e a banda do *cluster*.

Block-report As mensagens de block-report são listas de blocos saudáveis que o DataNode pode oferecer. Por serem mensagens mais longas, demandam maior processamento. Desta forma, um block-report é enviado logo após o registro com o NameNode e então depois a cada período de aproximadamente uma hora. O envio de block-reports é aleatorizado para evitar que o NameNode seja inundado por ondas de block-reports.

Block-received Do ponto de vista de desempenho, enviar um novo block-report após receber com sucesso um novo bloco seria inconveniente. Assim, nesse caso uma mensagem mais curta de block-received é enviada para informar ao NameNode sobre blocos recém-recebidos e que não constaram em mensagens de block-report ou block-received enviadas anteriormente.

Além do envio dessas mensagens outra tarefa de manutenção do DataNode é executar periodicamente o block-scanner. O block-scanner varre os blocos conferindo-os contra seus *checksums*. Com isso, o DataNode consegue identificar réplicas corrompidas, as quais são prontamente notificadas ao NameNode.

3.3.3 NameNode

O NameNode é o principal elemento do sistema sem o qual o HDFS não pode funcionar. É o arbitrador e o mantenedor do espaço de nomes. Esse nó tem por responsabilidades:

- **Atender às requisições de clientes**

O NameNode é um processo *multithread* que processa requisições de vários clientes simultaneamente. O NameNode realiza as mudanças nos metadados em nome dos clientes e coordena as interações destes com os DataNodes, seja alocando DataNodes para receberem novos blocos de dados durante escritas ou fornecendo listas de DataNodes que contêm réplicas de blocos de um certo arquivo. Maiores detalhes sobre a interação do NameNode com clientes serão vistos na Seção 3.4.

- **Controlar o acesso a arquivos**

Além do tradicional controle de acesso do padrão POSIX [81] por meio de permissões de usuário e grupo, o NameNode precisa garantir que haja no máximo um escritor por arquivo. Esse controle é feito por meio de *leases*. *Lease* é “um contrato que dá a seu portador direitos específicos sobre o recurso por um período limitado de tempo” [42]. *Lease* é um mecanismo particularmente útil em ambientes sujeitos a falhas, uma vez que garante que a posse sobre um recurso possa vir a ser revogada caso seu atual detentor falhe ou seja desconectado do *cluster* antes de liberar o recurso.

No caso específico do HDFS quando um cliente deseja escrever para um arquivo ele deve obter uma *lease*. Ela deve ser periodicamente renovada para evitar que expire. Quando o arquivo é fechado, a *lease* é automaticamente revogada. A duração da *lease* é definida por um limite *soft* e um *hard*. O limite *soft* é curto, um minuto, pois visa garantir a consistência da escrita. Durante esse período o detentor da *lease* tem acesso exclusivo para escrever no arquivo. Expirado o limite *soft*, inicia-se o limite *hard*. O limite *hard*, mais longo, uma hora, visa proteger o sistema de arquivos contra falhas no cliente que causariam o “sequestro” do acesso ao arquivo além de permitir que o cliente gaste algum tempo com processamento e volte a escrever sem ter de pedir uma nova *lease*. Durante o limite *hard* o atual detentor da *lease* ainda terá acesso exclusivo de escrita, porém quaisquer outros clientes estarão livres para fazer preempção da *lease*. Caso o possuidor da *lease* não a renove antes que o período *hard* se finde, o NameNode assumirá que o cliente terminou a escrita ou falhou, e fechará automaticamente o arquivo, revogando a *lease*. O fato de um cliente deter uma *lease* não impede outros clientes de lerem o arquivo. Arquivos podem ser lidos enquanto são escritos.

- **Administrar réplicas de blocos**

O NameNode é o único nó que tem conhecimento sobre a localização de todas as réplicas de bloco no *cluster*. É com esse conhecimento que ele tenta garantir que todos os arquivos tenham o nível de replicação desejado e que a ocupação dos DataNodes mantenha-se balanceada. Esse conhecimento é adquirido por meio das

mensagens que são enviadas pelos DataNodes. Através das mensagens de heartbeat o NameNode sabe quais DataNodes estão ativos e quais são suas atuais cargas de trabalho e capacidade livre de armazenamento. Isto fornece subsídios ao NameNode em suas decisões de onde armazenar novos blocos. Por meio das mensagens de block-report e block-received o NameNode fica ciente de quais DataNodes possuem réplicas de um determinado bloco. Se um bloco tem réplicas em excesso o NameNode instruirá a um dos DataNodes que contém o bloco para remover sua réplica. Para escolha do DataNode será preferido manter o número de armários de rede com cópias de bloco e remover a cópia do DataNode com menor capacidade livre, a fim de manter o balanceamento sem comprometer a disponibilidade do bloco. No caso de um bloco que esteja sub-replicado o NameNode instruirá DataNodes a copiarem blocos entre si. O critério para a escolha do DataNode que receberá o bloco é similar ao adotado para o posicionamento de réplicas de novos blocos, o qual será visto na Subseção 3.4.2.

- **Gerenciar alterações no espaço de nomes**

O NameNode controla e realiza todas as alterações na árvore de arquivos e diretórios bem como em seus atributos. Por razões de desempenho o NameNode mantém todo seu estado em memória principal. Isso inclui os *inodes* e as estruturas de dados que gerenciam réplicas e *leases*. Para evitar que todo o sistema de arquivos se perca em caso de falha do NameNode, faz-se o *journaling* do sistema por meio de dois arquivos. O primeiro funciona como um *checkpoint*, contendo uma versão seriada da árvore do sistema de arquivos e dos metadados. O segundo é um *log* no qual são registradas todas as mudanças ocorridas no espaço de nomes desde o último *checkpoint*. Um cliente executando uma operação no sistema de arquivos receberá uma resposta sobre sua requisição apenas quando o registro da operação no *log* tiver sido *flushed* e *synced* (transferidos dos *buffers* da aplicação para os discos), garantido então a durabilidade da operação. As informações sobre *leases* e sobre localização de réplicas não são registradas no *log* por serem voláteis. *Leases* são válidas por um período curto de tempo. E informações sobre réplicas podem mudar rapidamente, visto que DataNodes falham a taxas consideráveis, conforme pode ser observado na estatística sobre taxas de falhas nos *clusters* do Yahoo mencionada na Seção 3.1. Portanto quaisquer informações sobre *leases* e localização de réplicas são perdidas no caso de falha do NameNode. Tanto o *checkpoint* como o *log* podem ser armazenados em vários discos ou até mesmo em partilhas NFS a fim de evitar que sejam perdidos no caso de uma falha catastrófica do nó que hospeda o processo do NameNode.

Quando o NameNode inicia suas operações ele carrega o último *checkpoint* para a memória e aplica as entradas do *log* para recuperar o último estado do espaço de nomes. Em seguida ele gera um novo *checkpoint*, trunca o *log* e entra em um estado chamado de *safe-mode*. Durante o *safe-mode* o NameNode pode apenas atender requisições de leitura dos metadados e não pode instruir DataNodes a removerem ou replicarem blocos. O objetivo do *safe-mode* é dar tempo aos DataNodes de contactarem o NameNode e permitir que a informação sobre a localização das réplicas seja reconstruída. Quando é sabido que para 99,9% dos blocos está disponível o número mínimo permitido de réplicas, o NameNode pode sair do *safe-mode* e entrar no seu modo padrão de operação. Com este valor de 99,9% o NameNode consegue atender a praticamente qualquer requisição de leitura e não fica retido em *safe-mode* por causa de DataNodes que não conseguiram executar ou cuja iniciação está tomando mais tempo que o normal.

É importante notar que o NameNode é o único elemento do sistema que atende requisições de usuários do HDFS e que mantém informações sobre quais arquivos estão armazenados no sistema, que blocos os compõem e onde estão tais blocos. A queda do NameNode implicará na queda de todo o sistema de arquivos. Ou seja, o NameNode é um ponto único de falha para o HDFS.

3.3.4 Backup Node

A implementação do NameNode cria *checkpoints* apenas durante sua iniciação. Criá-los em paralelo ao tratamento de requisições dos clientes acarretaria em perda de desempenho e eventualmente de consistência, pois estar-se-ia registrando um estado do NameNode ao mesmo passo que esse estado é modificado. No entanto, com o sistema de arquivos executando por período prolongados e sem reinícios, o *log* poderá crescer de forma arbitrária, aumentando as chances de que ele seja corrompido ou até mesmo perdido. Por outro lado, um *log* grande impacta consideravelmente o tempo de iniciação do NameNode. O *log* de toda uma semana pode levar uma hora para ser processado [92].

A fim de remediar essa situação o NameNode conta com um *checkpoint helper*, o Backup Node. O Backup Node produz novos *checkpoints* para o NameNode a intervalos regulares, permitindo que o *log* seja truncado. O Backup Node implementa uma estratégia eficiente de *checkpointing*, visto que ele mantém sua própria visão dos metadados sincronizada com a do NameNode. Isto evita a necessidade de periodicamente interromper o NameNode para descarregar seu estado completo como ocorreria em técnicas tradicionais de *checkpointing*. O uso do Backup Node também evita a necessidade de travas de granularidade fina para as estruturas de dados do NameNode ou de estratégias de *copy-on-write* [108].

Quando o Backup Node inicia, ele se registra no NameNode e lhe requisita as cópias

mais recentes de seu *checkpoint* e *log* a fim de criar um novo *checkpoint*. Mas o Backup Node requisita o *log* somente desta vez, apenas com o objetivo de sincronizar-se com o NameNode. Uma vez sincronizado o Backup Node passa a ser visto como local extra para o armazenamento do *log*. Assim, quando o NameNode escreve uma nova entrada no *log*, essa entrada é propagada para o Backup Node. Tão logo é recebida, ela é aplicada pelo Backup Node às suas próprias estruturas de dados em memória, replicando as alterações no espaço de nomes. Consequentemente o Backup Node é capaz de manter uma visão do espaço de nomes sincronizada com o NameNode. E portanto, fazer o *checkpoint* de seu próprio estado equivale fazer a *checkpoint* do estado do NameNode. Isto é possível porque tanto o NameNode como o Backup Node compartilham boa parte de seus códigos. De fato, em termos de implementação, o Backup Node é uma subclasse do NameNode. O Backup Node pode ser visto como um NameNode somente-leitura. Ele contém todos os metadados do sistema exceto pelas localizações dos blocos e *leases*.

3.4 Operações

O HDFS implementa muitas das operações de arquivos definidas pelo padrão POSIX [81] tais como criação de arquivos, leitura, escrita, remoção, listagem de diretórios, renomeação, modificação de dono e das permissões de acesso, criação de *links* simbólicos, etc. A seguir serão detalhadas as operações de leitura, escrita e remoção.

3.4.1 Leitura

Um usuário utiliza-se do cliente do HDFS para abrir o arquivo quando deseja lê-lo. O cliente contactará o NameNode para obter a localização das réplicas dos primeiros blocos do arquivo, conforme pode ser visto na Figura 3.2 que mostra o diagrama de sequência para a leitura de um arquivo com dois blocos. O NameNode então envia uma lista de DataNodes ordenada por proximidade ao cliente. O cliente então comunica-se com o primeiro DataNode da lista para obter um bloco. Se a comunicação com o DataNode falha ou é detectada a corrupção do bloco pelo cliente ao validar os *checksums*, o cliente continua a ler do próximo DataNode na lista e notifica o NameNode do ocorrido. Conforme o cliente lê o arquivo, ele pode vir a contactar o NameNode novamente para obter a localização dos próximos blocos.

No contexto do Hadoop, o conceito de proximidade corresponde à largura de banda. Quanto maior a largura de banda entre dois nós, mais próximos entre si eles estarão. Como é difícil medir a largura de banda entre dois nós em um *cluster*, ela é estimada a partir da topologia da rede. O Hadoop supõe que largura de banda diminua na seguinte sequência ilustrada pela Figura 3.3:

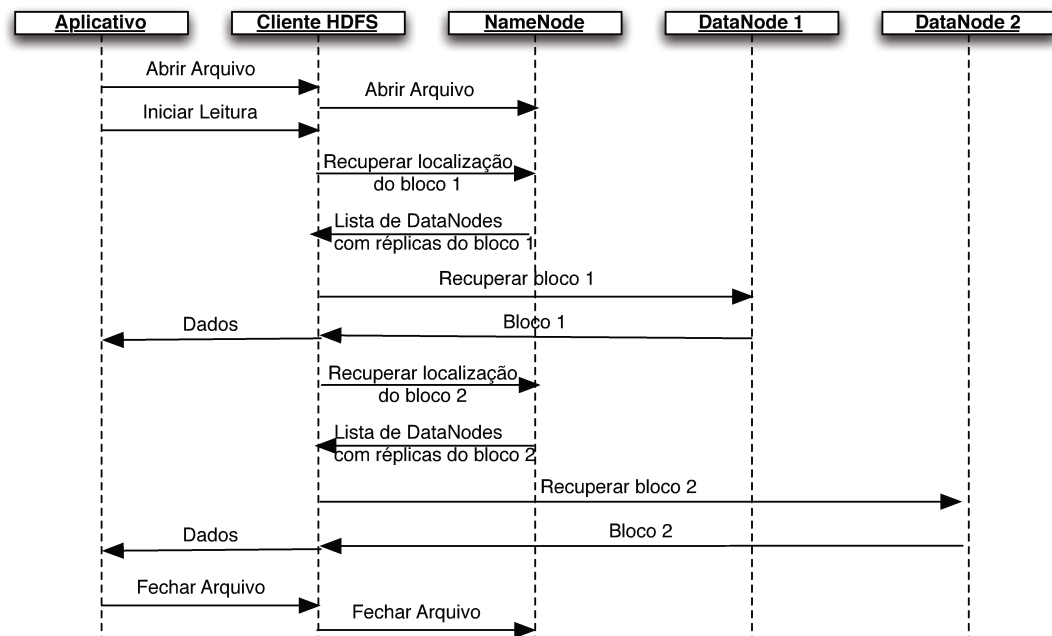


Figura 3.2: Diagrama de sequência para a leitura de um arquivo de dois blocos no HDFS.

- processos no mesmo nó (d_1);
- processos em diferentes nós do mesmo armário de rede (d_2);
- processos em diferentes armários de rede do mesmo *data center* (d_3);
- processos em diferentes *data centers* (d_4).

3.4.2 Escrita

Quando o cliente cria ou abre um arquivo para adicionar dados, o NameNode primeiramente verifica se o arquivo já existe, se o cliente tem permissão e se ninguém detém uma *lease* para o arquivo. Passado nos testes, o NameNode cria um novo arquivo na árvore do sistema de arquivos (se ele não já existir), e concede uma *lease* ao cliente. Conforme os dados vão chegando ao cliente, ele os organiza em pacotes de 64 kB que são postos numa fila. O cliente pede então ao NameNode que aloque um novo bloco. O NameNode devolve um identificador único para o novo bloco, juntamente com uma lista de DataNodes para os quais o cliente deve replicar os dados. Os DataNodes são dispostos em linha e os pacotes de dados começam a ser repassados do cliente para o primeiro DataNode e do primeiro

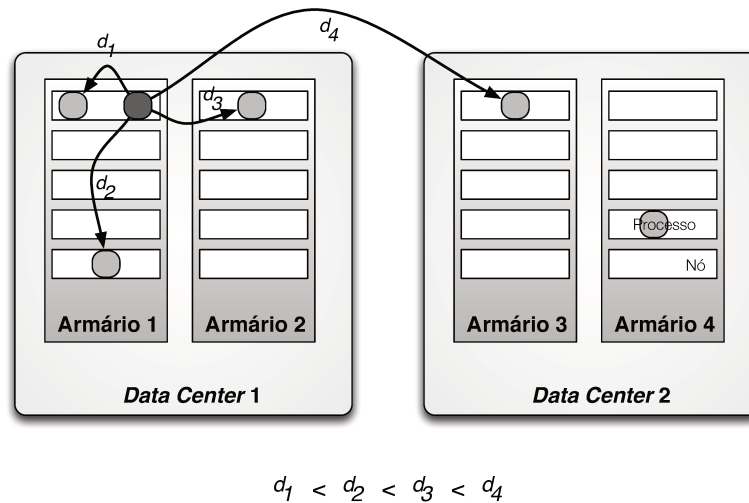


Figura 3.3: Distâncias na rede para o Hadoop.

DataNode para o segundo e assim por diante. Quando o pacote atinge o último DataNode ele manda uma mensagem de reconhecimento do pacote para o cliente através da fila de DataNodes formada. No momento em que o cliente recebe essa mensagem, ele sabe que o pacote foi escrito em todos os DataNodes. Se um bloco é totalmente preenchido o cliente requisita um novo ao NameNode. Quando então o arquivo é fechado, o cliente força que últimos dados sejam enviados para os DataNodes e então manda um mensagem de arquivo completo para o NameNode. O NameNode por sua vez revoga a *lease* concedida e registra no *log* que o arquivo foi fechado e quais são os blocos que o compõem. Isso significa que se o NameNode falhar antes que uma escrita seja terminada, todos os blocos recém-adicionados serão perdidos, visto que os novos blocos não estão registrados no *log*. Um exemplo de criação de um arquivo de um bloco com fator de replicação três pode ser visto na Figura 3.4.

Caso um dos DataNodes falhe enquanto um bloco é escrito, o bloco ganha um novo identificador. Isto evita que o bloco armazenado no DataNode falho seja considerado válido. O processo de escrita segue então da mesma forma com os DataNodes restantes e a partir do primeiro pacote que ainda não foi reconhecido. Mais tarde o NameNode notará a réplica faltante e tomará atitudes para retomar o nível normal de replicação do bloco. Vale também lembrar que o cliente é responsável por calcular os *checksums* que são enviados juntamente com o dados para os DataNodes.

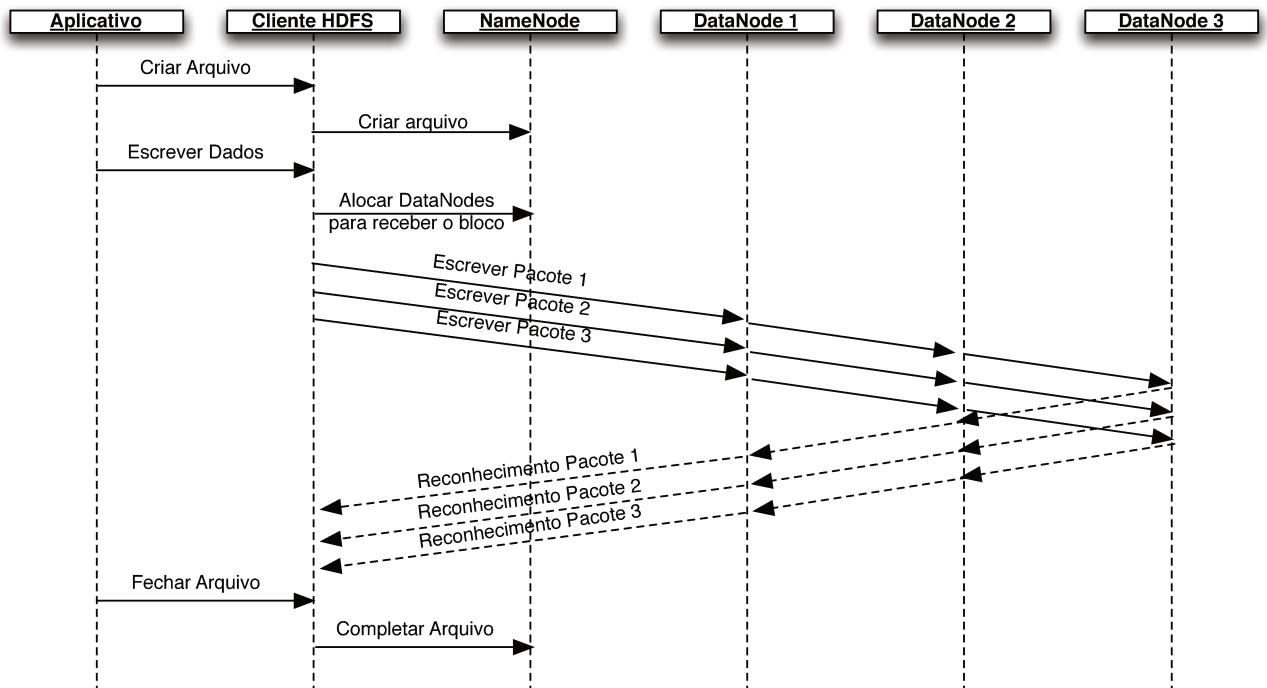


Figura 3.4: Diagrama de sequência para a escrita de um arquivo de um bloco no HDFS

Modelo de coerência

O padrão POSIX diz “que após um `write()` para um arquivo feito com sucesso, qualquer `read()` executado com sucesso de cada posição em bytes no arquivo que foi modificada por aquela escrita deve devolver os dados especificado pelo `write()` para aquela posição até que tais posições em bytes sejam modificadas novamente” [110]. O HDFS relaxa essa imposição do padrão visando melhorar o desempenho. No modelo de coerência adotado pelo HDFS um arquivo é visível para outros leitores logo após ser criado. No entanto, com relação ao conteúdo, todos os blocos são visíveis para os outros leitores, exceto pelo bloco que está sendo escrito no momento. Para forçar que todos os *buffers* do cliente sejam sincronizados para os DataNodes, o cliente conta com o método `hflush()`. Após uma execução com sucesso de `hflush()` é garantido que todo dado escrito até o momento esteja visível para os leitores. Por meio de `hflush()`, um usuário pode definir o equilíbrio entre desempenho e visibilidade dos dados.

Posicionamento de réplicas

A política padrão de posicionamento de réplicas do HDFS coloca a primeira réplica no mesmo nó que o processo escritor. Se o nó do processo escritor não for um `DataNode`, um `DataNode` é escolhido aleatoriamente. A segunda réplica é posta em um nó diferente da primeira, mas que está no mesmo armário de rede. A terceira réplica é armazenada em um nó que esteja em um armário de rede diferente das outras duas réplicas. Se o nível de replicação do arquivo é maior que o usual de três, as demais réplicas são postas em `DataNodes` escolhidos de forma aleatória, porém tomando-se o cuidado de não colocar muitas réplicas no mesmo armário de rede. Com esse arranjo tenta-se atingir um bom equilíbrio entre:

- confiabilidade, pois blocos são armazenados em pelo menos dois armários de rede distintos;
- largura de banda para escritas, pois dados tem de trafegar um único comutador de rede;
- e desempenho de leitura, pois há pelo menos duas escolhas de armário de rede para se obter o bloco.

3.4.3 Remoção

O HDFS possui uma funcionalidade de “lixeira” que se ligada não apaga o arquivo imediatamente, mas move o arquivo para uma pasta especial e posterga sua real remoção por um período definido pelo administrador do sistema. Durante esse período o arquivo ainda pode ser recuperado. Findo o período, o `NameNode` remove o arquivo da árvore do sistema e comanda todo `DataNode` que tenha réplicas de blocos do arquivo a apagá-las.

3.5 Escalabilidade

Visto que todos os *inodes* ficam armazenados na memória principal do `NameNode`, o número de arquivos que uma instalação de HDFS comporta é limitado pela extensão do referido recurso. Para driblar tal limitação a versão 0.23 lançada em novembro de 2011 introduziu uma nova funcionalidade denominada HDFS Federation, que visa aumentar a escalabilidade do sistema de arquivos, principalmente no que tange o número de arquivos. O HDFS Federation adota solução semelhante a do sistema de arquivos Panasas, a qual foi apresentada na Subseção 2.2.3. Cada `NameNode` é responsável por um volume, um sistema de arquivos independente. Cabe ao cliente do HDFS fornecer às aplicações uma visão única, transformando os volumes em subdiretórios da raiz de um sistema de

arquivos virtual. No caso específico do HDFS Federation esse sistema de arquivos virtual é denominado ViewFs.

Dentro dessa nova funcionalidade DataNodes podem servir a mais de um NameNode. Portanto eles se registram e enviam mensagens de estado a todos os NameNodes ao qual são subservientes.

3.6 Considerações finais

Neste capítulo fizemos uma descrição detalhada do HDFS destacando as interações entre seus elementos. Desta forma deve ter ficado evidente a criticalidade do NameNode para a disponibilidade do sistema de arquivos. Mesmo com o HDFS Federation, essa criticalidade não é de toda diminuída, pois nessa nova configuração a queda de um NameNode implica na indisponibilidade de todo um volume.

O capítulo que se segue se incumbirá de discutir essa criticalidade bem como apresentar algumas propostas que foram feitas objetivando minimizar o impacto da queda do NameNode.

Capítulo 4

Alta disponibilidade para o HDFS

Após detalharmos o funcionamento do Hadoop Distributed File System, neste capítulo iremos focar nossa discussão na questão da disponibilidade do sistema. Até o momento a alta disponibilidade de dados do HDFS parece ser um problema resolvido. Ela é alcançada por meio da replicação de blocos. O algoritmo de disposição de réplicas do HDFS garante que réplicas de um mesmo bloco sempre estarão em nós distintos e que réplicas estarão em pelo menos dois armários de rede. Assim, se um nó falha os dados podem ser lidos de outro nó. O caso em que um armário de rede inteiro é danificado também não é problema. Nesta situação o cliente pode conectar-se ao armário de rede sobrevivente e continuar seu trabalho. Se algum usuário prevê uma alta demanda pela leitura de um arquivo, ele pode definir um valor maior para o nível de replicação deste arquivo. Isto habilita os vários leitores a trabalharem em paralelo, obtendo seus dados de réplicas diferentes.

Estas estratégias se mostram eficazes na prática. Nos *clusters* de produção do Yahoo a taxa de falha é de 2 a 3 nós por dia em um grupo de mil, e essa taxa é três vezes maior para nós recém saídos da fábrica [91]. Se a falha ocorrer em um DataNode de 12 TB, ele pode ser re-replicado em 7 minutos [39]. Durante o ano de 2009, o Yahoo descobriu que nos seus 10 *clusters* HDFS executando Apache Hadoop 0.20.3 (um total de 20 mil nós), apenas 650 blocos foram relatados como perdidos de um total de 329 milhões de blocos[31]. Desses blocos, 533 foram incorretamente relatados como perdidos por causa de um defeito quando na verdade foram abandonados devido a falhas de escrita. 98 blocos foram criados explicitamente sem replicação. Os restantes 19 blocos (ou apenas 0,00000578% do total de blocos) foram perdidos devido a 7 diferentes defeitos no HDFS. Diante desses valores, vê-se que a impossibilidade de executar uma operação sobre os dados de um arquivo é uma ocasião rara no HDFS.

Por outro lado a disponibilidade do sistema como um todo carece de melhorias. A estratégia de um único mestre adotada pelo HDFS 0.21 concentra todas as informações sobre o espaço de nomes em um único ponto, o NameNode. O NameNode faz uso de

checkpoints e *logs* armazenados em vários lugares para ter essas informações persistidas. Todavia isso apenas previne todos os arquivos de serem perdidos completamente, porque em nenhum outro lugar existe a informação de que blocos compõem um certo arquivo ou até mesmo de que arquivos estão no sistema. Mesmo assim, se o NameNode está se recuperando de uma falha, ele ainda tem de esperar por block-reports de todos os DataNodes antes de ser capaz de processar requisições de clientes. O que pode levar mais de meia hora para *clusters* grandes [19, 20]. Para se ter uma ideia da confiabilidade do NameNode, de fevereiro de 2010 a agosto de 2011 — portanto 18 meses — ocorreram 22 falhas entre 25 *clusters* HDFS no Yahoo [31]. Ou seja, 0,58 falhas por *cluster* por ano. Entre maio de 2008 e outubro de 2009, o *cluster* HDFS da ContextWeb, uma empresa de anúncios na Web, experimentou 3 falhas, ou 2 falhas por ano. Embora duas falhas por ano ou uma falha a cada dois anos possa parecer pouco para o usuário comum, é um número bastante considerável para *clusters* grandes pois o tempo necessário reiniciá-los é longo. Uma falha em um *cluster* como do Facebook, com de 2 mil nós, 21 PB e 150 milhões de arquivos pode deixá-lo inoperante por pelo menos 45 minutos [20].

Mas a questão de alta disponibilidade do NameNode não se remete apenas a falhas. Eventualmente será necessário realizar manutenções preventivas no NameNode bem como atualizações ou trocas de componentes de hardware. Consequentemente o número de vezes em que NameNode estará fora de serviço será maior ainda. Independentemente do motivo da parada do NameNode é necessário manter o sistema de arquivo funcionando, pois as paradas longas significam máquinas ociosas, muitas tarefas a serem reexecutadas e acordos de nível de serviço (*SLA*) comprometidos. Ou seja, incorrerá em prejuízos financeiros que poderiam ser facilmente evitados se uma solução de alta disponibilidade estivesse possível. Demonstra-se portanto a importância do desenvolvimento de tal solução.

4.1 Propostas de alta disponibilidade para o HDFS

Por ser um sistema de arquivos distribuído largamente utilizado não tardou muito para que a questão da disponibilidade começasse a ser discutida pela comunidade de desenvolvedores do Hadoop. De fato as primeiras discussões que se tem registradas sobre o assunto datam de junho de 2008 [69]. Nesse meio tempo várias propostas de como conferir alta disponibilidade ao HDFS foram feitas até culminar na solução oficial integrada no HDFS 2.0.0-alpha em maio de 2012. A seguir descreveremos várias dessas propostas incluindo a oficial. A proposta dessa dissertação será tratada em detalhes no capítulo seguinte.

4.1.1 Linux-HA e DRBD

ContextWeb, uma empresa de anúncios na Web, desenvolveu uma solução simples [15, 34], por não exigir alterações de código, para conferir alta disponibilidade à sua instalação de HDFS. A solução ataca o problema de dois pontos de vista: infraestrutura e software.

Do ponto de vista de infraestrutura tenta-se usar redundância em todo lugar que for possível. Desta forma todos os nós usam fontes de energia redundantes e discos organizados em RAID [80] nível 1 (discos espelhados). Cada nó é ligado ao seu comutador de rede por dois enlaces físicos combinados por meio de LACP/802.3ad. O LACP (*Link Aggregation Control Protocol* - protocolo de controle de enlace agregado) [58] combina dois enlaces físicos em único enlace lógico, agregando suas larguras de banda. Além do aumento da largura de banda, outro benefício do LACP é que se um dos enlaces é danificado os dados podem ainda seguir pelo enlace sobrevivente. Os comutadores são também ligados entre si por LACP.

Do ponto de vista de software usa-se a técnica do par ativo-passivo apresentada na Subseção 2.2.2. Logo, temos um NameNode atuando como servidor de metadados ativo e um outro nó mantido em estado de prontidão para iniciar um novo processo de NameNode caso o primeiro falhe. Essa configuração criada pela ContextWeb pode ser vista na Figura 4.1. Note que o par ativo-passivo é mantido em comutadores de rede distintos a fim de aumentar a confiabilidade do sistema.

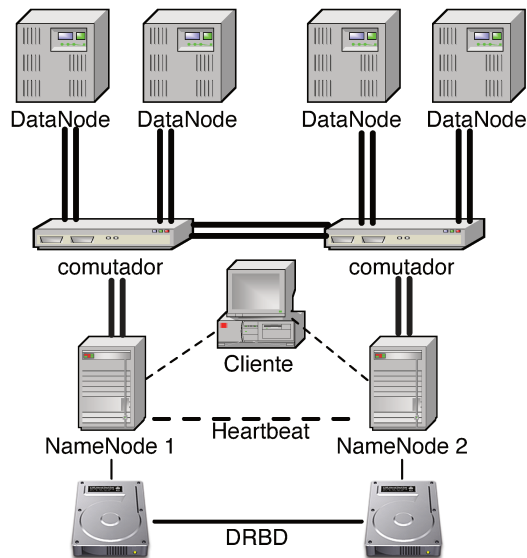


Figura 4.1: Solução de alta disponibilidade para o HDFS da ContextWeb.

Para compartilhar o *log* entre os dois nós do par é empregado o DRDB - *Distributed*

Replicated Block Device (dispositivo de bloco replicado distribuído) [87] que “*pode ser entendido como um RAID 1 baseado em rede*” [35]. O DRBD é implementado como um dispositivo de bloco do núcleo Linux em que qualquer dado nele escrito é espelhado via uma conexão TCP/IP para um outro dispositivo de armazenamento, produzindo assim replicação de dados de maneira transparente para aplicações. Com DRBD a máquina de prontidão tem acesso às versões do *checkpoint* e do *log* correspondentes ao estado do NameNode antes de sua queda. Entretanto a nova instância de NameNode ainda precisará recuperar a localização das réplicas de blocos, o que tomará certo tempo.

A solução apoia-se no Heartbeat do projeto Linux-HA [47] para detectar a falha do NameNode e iniciar o *failover*. O Heartbeat funciona com um *daemon* que monitora seus pares no *cluster* por meio da troca constante de mensagens temporizadas do tipo “*você está aí?*”. Caso o nó do NameNode não responda a tempo e a temporização estoure, ele é tido como falho. Nesse caso o Heartbeat iniciará um novo processo de NameNode na máquina que estava de prontidão.

Para que a transição entre NameNodes seja transparente tanto para clientes como para DataNodes é utilizado IP virtual [99]. IP virtual é uma técnica que permite dinamicamente reatribuir um endereço IP para um novo endereço MAC. Quando a máquina que estava de prontidão torna-se o NameNode ativo, ela faz o *broadcast* de uma resposta do protocolo ARP (*Address Resolution Protocol*) [8] sem que tenha havido alguma consulta por parte de algum outro nó (*gratuitous ARP reply*). Essa resposta não requisitada tem por efeito fazer com os elementos da rede remapeiem o endereço de rede IP do antigo NameNode para endereço físico do novo servidor de metadados. Assim DataNodes e clientes começam a enviar suas mensagens para o novo NameNode ativo sem que tomem ciência desse fato. Quando um DataNode contactar o novo NameNode, este o pedirá para se registrar novamente, forçando o envio de um *block-report*. Com isso o mapeamento de blocos para DataNodes será reconstituído.

ContextWeb afirma que com essa técnica o tempo de *failover* foi de aproximadamente 15 segundos para um *cluster* de 40 nós [34] e 100 TB de armazenamento. Com essa técnica ele mantiveram seu *cluster* HDFS operacional mesmo em face de 6 *failovers* entre maio de 2008 e outubro de 2009, sendo que apenas 3 foram planejados. Note que essa solução inicia um processo de servidor de metadados do zero, sem estado. O novo processo do NameNode terá de carregar o *checkpoint*, aplicar o *log* e processar *block-reports* de todos os DataNodes. Logo, o tempo de *failover* será uma função do tamanho do *log*, do número de DataNodes e de blocos. O tempo de *failover* foi pequeno porque o *cluster* era pequeno. Para *clusters* maiores o tempo de *failover* pode se aproximar da cifra de 45 minutos, que é o aproximadamente o tempo gasto para reiniciar o *cluster* HDFS de 2 mil nós do Facebook [20]. Ou seja, é uma solução de alta disponibilidade que não escala.

4.1.2 Solução da IBM China Research

Wang et al. no artigo intitulado “*Hadoop High Availability through Metadata Replication*” [100] descrevem uma solução de alta disponibilidade que pode adotar duas diferentes arquiteturas: um par primário-*backup* ou um grupo primário-escravos. A principal diferença entre as duas arquiteturas é que na última é permitido aos escravos executarem requisições do tipo somente-leitura. Os pesquisadores afirmam que ambas as arquiteturas se adaptam bem a cenários típicos de aplicações do MapReduce. Eles também afirmam que uma cópia completa do estado do NameNode primário penalizaria severamente o desempenho. Portanto essa solução replica apenas mudanças nos metadados e nas *leases*. As *leases* são tidas como importantes para manter consistência após o *failover*. O ciclo de vida da solução é dividido em três fases: iniciação, replicação e *failover*.

1. Iniciação

Na iniciação o NameNode primário espera por solicitações de registros dos escravos. Quando um escravo se registra, o primário envia a todos os outros nós uma tabela de endereços IPs de todos os escravos já registrados por meio de *broadcast*. Desta forma, os nós escravos podem tomar ciência de seus pares e serem capazes de organizar eleições no caso de queda do NameNode primário. Após o registro de todos os escravos, eles verificam com o primário se eles possuem uma cópia do último *checkpoint* do sistema. Se não a possuem, o primário lhes enviará. O sistema pode prosseguir para a fase seguinte quando todos os nós estão sincronizados com o NameNode primário.

2. Replicação

Nesta fase as operações sobre o espaço de nomes e *leases* são interceptadas no NameNode primário e suas informações são enviadas aos escravos. Os escravos confirmam a recepção do metadados de acordo com o modo de sincronia vigente. O modo de sincronia é determinado por um módulo de decisão adaptativo. Esse módulo de decisão escolhe entre três possíveis modos baseados em limiares definidos pelo administrador e pelas atuais taxa de transferência e carga de trabalho. Os três modos se diferenciam pelo momento em que as requisições dos clientes são dadas como efetivadas e quando os dados são enviados para os escravos. Desta forma os três modos são:

Modo 1 Primário aplica a operação em memória e em disco antes de propagá-la para escravos. A operação é efetivada para o cliente quando escravos aplicam tanto a operação em memória quanto em disco. (Figura 4.2(a))

Modo 2 Primário aplica a operação em memória e em disco antes de propagá-la

para escravos. A operação é efetivada para o cliente após escravos aplicarem a operação em memória. (Figura 4.2(b))

Modo 3 Primário aplica a operação em memória e a propaga para os escravos. A operação é efetivada para o cliente após ter sido registrada no disco do primário e ter sido aplicado à memória dos escravos. (Figura 4.2(c))

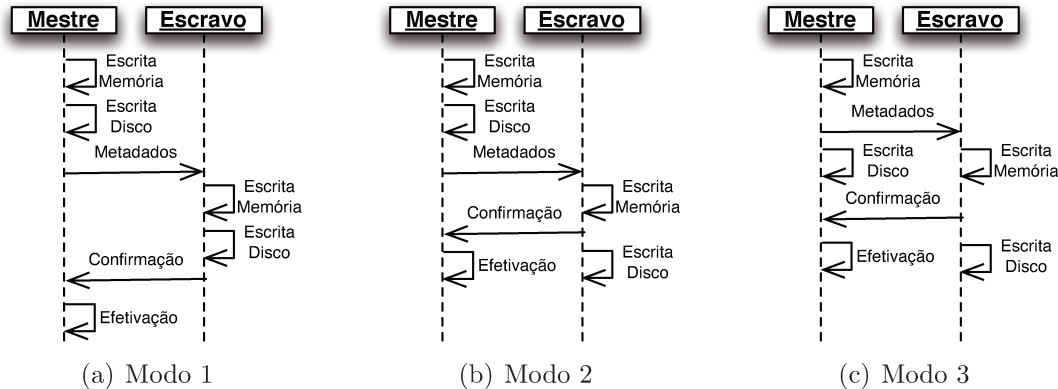


Figura 4.2: Modos de sincronização da solução desenvolvida pela IBM China.

Modos 1 e 2 são recomendados para redes locais e garantem que os metadados estarão armazenados de forma persistente no primário antes de serem enviados aos escravos. Modo 3 é adequado para redes geograficamente distribuídas (WANs) de banda estreita, pois sobrepõe a latência de transmissão da operação com o registro em disco. Para manter a consistência entre nós é empregado um protocolo de efetivação de três fases não bloqueante. Entretanto, a efetivação se dá com a anuência de uma maioria ao invés de todos os nós para evitar degradação de desempenho.

Além dos metadados, primário e escravos trocam mensagens de *heartbeat* a fim de que o escravo possa detectar a falha do primário.

3. *Failover*

Qualquer um dos escravos pode organizar uma eleição se ele suspeitar de falha do primário. Se algum outro nó discordar que o NameNode primário está inativo ele cancelará a eleição. Todavia se a eleição prosseguir, o seu vencedor se torna o novo primário. Assim como na solução desenvolvida pela ContextWeb, o IP do primário anterior é transferido para o novo a fim de que o *failover* seja transparente para DataNodes e clientes. Uma vez que esta transferência é completada, os escravos restantes devem se registrar com o novo primário, reiniciando a primeira fase. Para completar o processo de *failover* o novo primário deverá também esperar ser contatado pelos DataNodes a fim de reconstituir o mapeamento entre blocos e DataNodes,

uma vez que essa informação não consta do processo de replicação dessa solução. Apenas como referência, essa etapa de processamento de block-report leva cerca de 20 minutos para o *cluster* HDFS de 2 mil nós do Facebook [20]

A solução foi testada em um ambiente de cinco nós Pentium 4 com 1,5 GB de memória RAM e *links* de 1 Gbps, sendo um NameNode primário, um escravo e três DataNodes. Devido a essas configurações o modo 1 de sincronização foi usado o tempo todo. Também optou-se por apenas usar arquivos de um bloco e com fator de replicação 3. A base de comparação tomada foi o Hadoop 0.20.

Para o experimento de tempo de *failover* o sistema de arquivos foi populado com um número variável de arquivos. Ao término desta etapa o NameNode era desligado, dando início ao processo de *failover*. Os pesquisadores notaram o tempo de *failover* é proporcional ao número de blocos no sistema e que o tempo de transferência de rede era consideravelmente maior que o tempo de processamento dos block-reports. Para 100 mil blocos o tempo de *failover* foi um pouco mais que 7 segundos.

No que tange o custo de replicação, de novo o tempo de transferência de rede teve o maior impacto, sendo responsável por mais da metade do tempo gasto em replicação dos metadados e *leases*. O número de blocos não pareceu afetar o desempenho, dado que os registros das operações com metadados tinham tamanho constante. No geral o tempo de processamento de operações sobre os metadados triplicou se comparado com a versão original do Hadoop 0.20.

Como nessa solução o tempo de *failover* é dependente do números de blocos, ela também é uma solução de alta disponibilidade não escalável. Comparando-se com a solução da ContextWeb, replicar apenas metadados apenas não parece ser muito vantajoso. Com 100 mil blocos e 5 nós, essa solução tem um tempo de *failover* que é apenas metade do tempo atingido pela solução da ContextWeb que tem 8 vezes mais nós. Este longo tempo de *failover* acaba por não compensar a redução para um terço do desempenho original, ainda mais considerando que apenas mudanças nos metadados estão sendo replicados.

4.1.3 NameNode Cluster

No NameNode Cluster [102] desenvolvido pelo China Mobile Research Institute, um *cluster* de nós organizados numa configuração mestre-escravo substitui o NameNode conforme pode ser visto na Figura 4.3. O mestre é responsável por atender às requisições dos clientes e por propagar a atualizações no sistema de arquivos para os escravos. Para isso o mestre conta com a ajuda do SyncMaster Agent, uma *thread* em segundo plano responsável também por gerenciar os escravos. Reciprocamente cada nó escravo executa o SyncSlave Agent para receber e processar as atualizações.

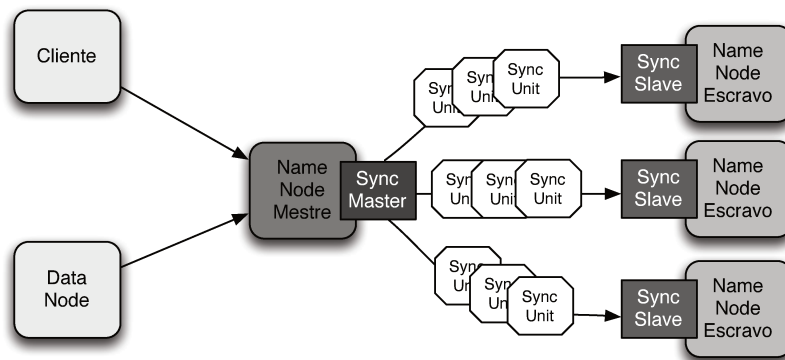


Figura 4.3: Arquitetura do NameNode Cluster.

Quando um novo NameNode se junta ao *cluster* ele tem se registrar junto ao mestre. O mestre então bloqueia todas as suas operações de escrita de metadados. Depois ele transfere a árvore do sistema de arquivos e o mapeamento de blocos para DataNodes para o novo escravo a fim que eles se sincronizem. Ao final desse processo de registro o novo escravo está pronto para começar a receber as mensagens de atualização.

Toda replicação de estado é feita em unidades denominadas SyncUnits. Uma SyncUnit pode carregar, além das informações que já são registradas no *log* do HDFS, dados sobre *leases*, mensagens de block-report, block-received e heartbeat, e comissionamento e descomissionamento de DataNodes. Essas SyncUnits permitem aos escravos possuírem uma cópia completa do estado do NameNode mestre. Para reduzir o custo de replicação as mensagens que vem de vários DataNodes são consolidadas e enviadas em lote para os escravos. O SyncMaster dedica a cada escravo uma *thread* para enviar suas SyncUnits. Se alguma *thread* falha ou estoura a temporização, seu escravo relacionado é marcado como não-sincronizado e terá que se registrar novamente com o mestre. As SyncUnits também são transferidas em ordem para manter consistência. Isto significa que uma nova SyncUnit não é transferida até que se complete a transferência da SyncUnit anterior para todas as *threads*. Para evitar comprometer o desempenho do mestre as atualizações são postas em *buffers*.

Nesta solução também é usado o HeartBeat [47] do projeto Linux-HA para gerenciar o processo de *failover*. Para que os nós do sistema sejam capazes de localizar o atual NameNode mestre é utilizado o Apache ZooKeeper [11], um serviço de coordenação distribuído altamente disponível, no qual os endereços dos nós bem como suas atuais situações são armazenados.

O NameNode Cluster é baseado no Hadoop 0.20. A experiência do China Mobile Research Institute é de uma redução de desempenho de 15% para operações de E/S e

trabalhos de MapReduce e de 85% para operações de escrita que somente operam sobre metadados comparado-se com o a versão original do HDFS. A razão para tal discrepância é que operações de leitura e escrita de arquivos interagem pouco com o NameNode. Já as operações exclusivas sobre metadados, como `mkdir` e `touch`, são impactadas pelo custo adicional da sincronização. O custo da replicação ficou um tanto alto, porém ela não impactou muito as operações de E/S. O custo é compensado pelo fato de haverem várias réplicas do NameNode com todo o seu estado, e que portanto podem prontamente assumir o posto de servidor de metadados primário. O código do NameNode Cluster se encontra disponível em <http://github.com/gnawux/hadoop-cmri>.

4.1.4 Avatar Nodes

No começo de 2010 o Facebook possuía um *cluster* HDFS 0.20 de 1.200 nós, 12 PB de armazenamento e cerca de 50 milhões de arquivos [17]. Seus engenheiros queriam ser capazes de realizar manutenções no NameNode sem que fosse necessário tornar o sistema indisponível. Adotar uma solução tal como a desenvolvida pela ContextWeb não ajudaria muito. Para um *cluster* daquele tamanho o *failover* levaria cerca de 45 minutos. Na configuração do Facebook os *checkpoints* e *logs* do NameNode eram gravados para uma partilha NFS com o fito de garantir a sobrevivência desses dados mesmo em caso de falhas catastróficas. Esse fato, mais o desejo de não realizarem grandes mudanças no projeto do HDFS que poderiam desestabilizar o sistema, levaram os desenvolvedores do Facebook a arquitetar uma solução com uma réplica primária e um *backup* a qual denominaram de Avatar Nodes [17]. Esta solução, cujo código encontra-se disponível em [50], é ilustrada pela Figura 4.4.

Nessa solução temos um Avatar NameNode primário que é responsável por atender as requisições dos clientes. Esse Avatar armazena os *checkpoints* e os *logs* em uma partilha NFS. Essa partilha NFS deve ser exportada por um sistema de armazenamento estável. Um Avatar NameNode passivo mantém-se lendo as entradas do *log* da partilha NFS, aplicando-as a seu estado à semelhança das réplicas *shadows* do Google File System. Desta forma o Avatar passivo pode manter sua visão sobre o espaço de nomes sincronizado com o primário.

Para que o Avatar NameNode passivo também possa saber onde as réplicas dos blocos estão no sistema, os DataNodes foram modificados para registrar-se e enviar suas mensagens de status a ambos Avatar NameNodes. Porém o Avatar NameNode passivo é mantido em *safe mode* para evitar que envie comandos aos DataNodes. Clientes não precisam estar cientes de qual Avatar está ativo no momento, pois IP virtual é usado para tornar essa configuração transparente.

Assim, para fazer alguma manutenção ou atualização no Avatar NameNode primário

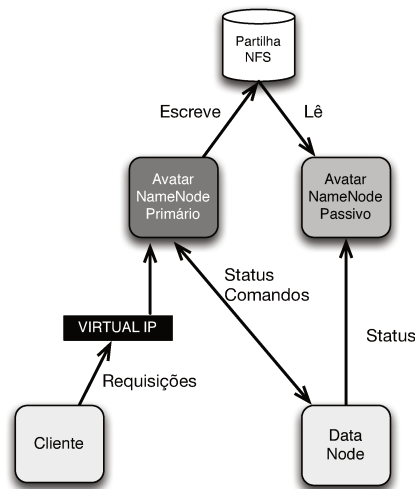


Figura 4.4: Arquitetura do Avatar Nodes.

o administrador deve desativá-lo, transferir o IP deste para o Avatar passivo e instruir esse último a assumir o papel de ativo. Esse processo de *failover*, que dura cerca de um minuto, é realizado de forma totalmente manual. Isto porque a solução não se destina a tratar falhas do NameNode. Por outro lado isso evita o problema de *split-brain*. Antes de comandar o passivo a trocar de papel, o administrador precisava garantir que o primário estava totalmente inativo. Isso torna impossível o cenário em que os dois Avatares pensem ser o ativo e escrevam simultaneamente para a partilha NFS, corrompendo-a. A transição não é sentida pelos clientes realizando leituras, pois eles mantêm as localizações de blocos em *cache*. Porém clientes poderão perder suas escritas, visto que novos blocos são apenas registrados no *log* quando o arquivo é fechado, conforme foi dito na Subseção 3.4.2.

Posteriormente houve várias melhorias à essa solução, as quais são relatadas em “*Apache Hadoop Goes Realtime at Facebook*” [20]. Essas melhorias incluem registrar os blocos no *log* à medida que são alocados, o que permite continuar escritas após o *failover*. Agora cada entrada do *log* é acompanhada por seu *checksum*, o que permite ao Avatar NameNode passivo ter certeza de sua integridade e completude. A informação de qual Avatar está ativo passou a ser armazenada no ZooKeeper [11]. O cliente HDFS pode consultar algum servidor ZooKeeper para saber para qual nó direcionar suas requisições, evitando a necessidade de IPs virtuais. DataNodes também usam essa informação para garantir que ignorarão qualquer comando vindo do Avatar NameNode passivo.

4.1.5 Solução oficial

Como dito no início desta seção, há muito se discute uma solução de alta disponibilidade para o HDFS dentro da comunidade de desenvolvedores do Hadoop. Essas discussões se concentraram no tíquete HDFS-1064 [70] do sistema de rastreamento de defeitos do projeto Apache Hadoop [44]. Desde então houve algumas mudanças paliativas, no sentido de reduzir o tempo de início do HDFS [36, 53, 94]. Nesse meio tempo também desenvolveu-se um sistema altamente disponível de *write-ahead logging* chamado BooKeeper [16]. Foi uma tentativa de fornecer armazenamento estável para o *log* do HDFS. Todavia sua integração ao sistema de arquivos não foi levada a cabo.

Um solução oficial para conferir alta disponibilidade só tomou forma a partir de fevereiro de 2011 com o tíquete HDFS-1623 - “High Availability Framework for HDFS NN” [49]. Após a implementação de 148 subtarefas, a solução entrou na *release* oficial 2.0.0-alpha do HDFS, lançada em maio de 2012. Trata-se essencialmente da mesma solução do Avatar Nodes e portanto não se faz necessário descrevê-la novamente. A escolha por esse projeto pela comunidade se deve muito provavelmente ao seu caráter pouco disruptivo em relação a base de código do HDFS e à experiência dos engenheiros do Facebook com o Avatar Nodes em grandes *clusters* de produção. Entretanto vale a pena destacar alguns aspectos que foram abordados em [108] e [48]. A solução também apenas permite *failover* manual. A implementação de *failover* automático está em progresso sob o tíquete HDFS-3042 [13]. O ZooKeeper não é usado para armazenar a informação sobre o servidor de metadados ativo. O cliente tenta conectar-se ao NameNode primário e ao *backup* em ordem, recebendo um valor de retorno especial no caso do cliente conectar-se ao *backup*. Caso o NameNode falhe no meio de uma requisição, o cliente será incapaz de determinar se ela foi processada com sucesso ou não. Operações idempotentes poderão ser retentadas. Porém no caso das que não são, o erro recebido pelo cliente deverá ser corretamente tratado. A solução tenta também evitar *split-brain*, um cenário que ambos NameNodes pensem ser o primário, atendam requisições e corrompam o armazenamento compartilhado. Nesse caso pode ser empregado uma ferramenta de *I/O fencing* tal como STONITH [95], que é capaz de controlar unidades de distribuição de força cortando a energia do nó a ser desativado.

A solução dos Avatar Nodes e por conseguinte a solução adotada pelo HDFS a partir de sua versão 2.0, seguem o princípio de Engenharia de Software de que em um sistema já estável, tenta-se introduzir o mínimo de perturbação com vista a manter sua estabilidade. Ou seja, elas não reformulam a arquitetura do HDFS com vistas a contemplar alta disponibilidade. São apenas adaptações para atender o novo requisito. Apesar disso apresentam bom desempenho por serem soluções que distribuem as tarefas de replicação de estado do NameNode pelo *cluster*, evitando que se concentrem neste nó.

4.1.6 UpRight HDFS

Clement et al. desenvolveram UpRight [27], uma biblioteca que tenta tornar tolerância a falhas bizantinas uma alternativa simples e viável à tolerância a falhas por queda para serviços baseados em *cluster*. O protocolo de consenso de UpRight combina ideias de protocolos anteriores como a execução especulativa de Zyzzyva [56], as técnicas de robustez de Aarvark [28] e a estratégia de separação entre acordo e execução de requisições de Yin et al. [112]. Upright divide o espaço de falhas bizantinas em falhas por omissão — quando o nó falha em enviar uma ou mais mensagens especificadas pelo protocolo, mas não envia mensagens não especificadas (isso inclui falhas por queda) — e falhas por comissão — quando o nó envia mensagens não especificadas pelo protocolo. Assim a tolerância a falhas de UpRight é especificada por dois parâmetros:

- um sistema UpRight é considerado seguro (“*right*”) até r falhas por comissão e qualquer número de falhas por omissão.
- um sistema UpRight é considerado seguro e ativo (“*up*”) se durante intervalos síncronos suficientemente longos há no máximo u falhas das quais no máximo r são por comissão e as demais por omissão.

UpRight implementa replicação de máquina de estado [89] e tenta isolar as aplicações dos detalhes do protocolo de replicação para que seja mais fácil convertê-las de tolerantes a falhas por queda para tolerantes a falhas bizantinas. Isto requer que os servidores sejam processos deterministas. Para ajudar nesse sentido, toda requisição que chega aos servidores é acompanhada de uma marca de tempo e de uma semente para números aleatórios. Isto habilita fluxos de execução que dependam de tempo ou de números aleatórios a serem deterministas pois todas réplicas recebem os mesmos parâmetros. Processos servidores devem também expor APIs para criar *checkpoints* de seu estado. Clientes por sua vez devem usar a biblioteca UpRight para fazerem todas as suas requisições e receberem suas respostas. As requisições são encaminhadas ao *cluster* UpRight onde são validadas, ordenadas e distribuídas aos servidores para execução.

A fim de comprovar que a biblioteca UpRight cumpria seu objetivo de tornar fácil a adaptação de sistemas tolerantes a falhas por quedas para tolerantes a falhas bizantinas, os pesquisadores resolveram aplicá-la ao HDFS. Desta forma 1750 linhas de código do NameNode foram modificadas para adicionar mais informações de estado para seu *checkpoint* e *log*, bem como para remover não determinismos de seu fluxo de execução. Desta forma, trechos de código que referenciavam números aleatórios ou marcas de tempo físico passaram se utilizar do tempo lógico e da semente presentes nos cabeçalhos das requisições. As várias tarefas periódicas de monitoramento — gerenciamento de *leases*, de *heartbeats*, de níveis de replicação, etc — que rodavam em suas próprias *threads*, passaram agora a ser

agendadas com base em tempo lógico para executar na agora única *thread* de execução do NameNode. Com todas essas modificações pôde-se garantir a consistência entre réplicas.

Em relação ao DataNode os pesquisadores optaram por não replicá-lo, pois poderia alterar as políticas de replicação do sistema de forma indesejada, afetando o posicionamento de blocos. Entretanto eles decidiram calcular *hashes* criptográficos dos blocos para permitir que os clientes possam validá-los. Tais *hashes* são armazenados no NameNode e tem por objetivo prevenir um ataque no qual dados em DataNodes são modificados em conjunto com seus respectivos *checksums*. Nesse caso a versão original do HDFS não seria capaz de notar a alteração não autorizada. Essa medida extra de segurança rendeu outras 946 linhas de código.

Toda comunicação entre os nós que anteriormente era feita pela camada de RPC do Hadoop foi substituída pela biblioteca UpRight, de forma que todas as mensagens sejam redirecionada para o *cluster* UpRight para que elas sejam distribuídas de acordo com seu protocolo bizantino. Com todas essas modificações chegou-se a um solução que implementa replicação ativa do NameNode. A arquitetura do UpRight-HDFS, cujo código está disponível em <http://code.google.com/p/upright/> está ilustrada na Figura 4.5,

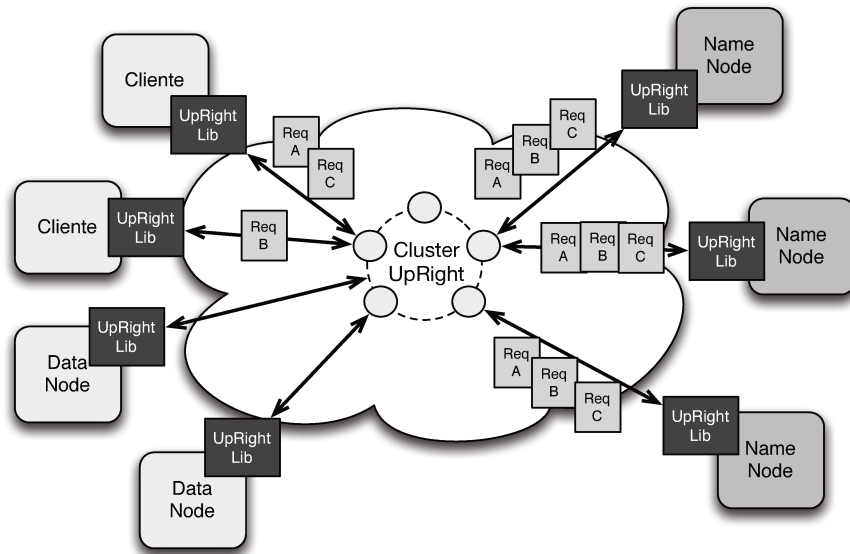


Figura 4.5: Arquitetura do UpRight HDFS.

Para avaliar a solução implementada foram utilizadas 107 instâncias do tipo *small* [4] na Amazon EC2 [3], sendo 50 instâncias para DataNodes e outras 50 para clientes. Os experimentos consistiram basicamente em clientes lendo e escrevendo um conjunto de arquivos de 1 GB. Foram utilizados nos testes a versão original do HDFS (uma versão de desenvolvimento posterior ao Hadoop 0.19) e o UpRight HDFS em duas configurações:

$u = r = 1$ e $u = 1$ $r = 0$. Esta última configuração simula um sistema tolerante a falhas por queda apenas. A vazão para leituras se manteve quase a mesma, havendo ligeira redução conforme o nível de tolerância a falhas aumentava. Nessa situação consumo de CPU aumentou por um fator de 2 para $r = 1$ e 1,6 para $r = 0$. Para escritas o comportamento foi semelhante, porém a queda de desempenho foi mais acentuada. A vazão reduziu em pouco menos de 30% para $r = 1$ e pouco menos de 20% para $r = 0$. Nesse casos o consumo de CPU foi de 2,5 e 1,1 vezes respectivamente.

De todas as soluções apresentadas aqui, o UpRight-HDFS se mostra a mais interessante pois emprega replicação ativa. Isso significa que o *failover* é inexistente: não existirão períodos momentâneos de indisponibilidade enquanto houver réplicas corretas e quaisquer falhas em réplicas são imperceptíveis aos clientes. Por empregar um protocolo bizantino, a solução protege o sistema contra réplicas faltosas e ataques maliciosos, embora a versão de HDFS usada (uma versão de desenvolvimento posterior a HDFS 0.19) não ofereça autenticação de usuários, o que a torna vulnerável. Por outro lado o UpRight-HDFS demanda um consumo bem maior de computação que o sistema original. Também requer determinismo do NameNode, o que limita como suas tarefas podem ser paralelizadas para alcançar melhor desempenho.

4.2 Quadro comparativo de soluções

No decorrer deste capítulo vimos como as diversas propostas tentam replicar o processo do NameNode, visando remover o ponto único de falha do HDFS. O estado de tal processo pode ser caracterizado, tal como feito no relatório técnico “*The Search for a Highly-Available Hadoop Distributed Filesystem*” [77] em dois componentes principais:

- **estado *hard*** Esta componente compreende o espaço de nomes: a árvore de arquivos e diretórios e seus atributos (dono, permissões, tempos de acesso e modificação, blocos que compõe o arquivo, entre outros). É o único componente de estado que é persistido entre uma execução e outra do NameNode. É também um componente mínimo e estável. É mínimo no sentido de ser a informação mínima requerida para trazer o NameNode para um estado operacional. É estável no sentido em que entre a falha e a recuperação do NameNode é imutável, uma vez que depende exclusivamente do NameNode.
- **estado *soft*** Esta componente compreende outra duas subcomponentes: informação sobre *leases* e localização de réplicas. Por ser um estado volátil, que pode alterar-se frequentemente e independentemente do NameNode, não há vantagem em persisti-lo. Como visto no Capítulo 3, *leases* são válidas por um curto espaço de tempo e DataNodes podem falhar, alterando as informações sobre réplicas.

Com base em quando o estado *hard* e o *soft* são ativamente replicados estabeleceu-se mesmo relatório as seguintes categorias para as soluções:

1. **Replicação Passiva:** quando uma variante da técnica de replicação passiva é usada para replicar o NameNode. Dependendo de quanta informação de estado é repassada para os *backups* em tempo de execução podemos definir subcategorias que variam em “temperatura”. Quando maior a “temperatura” de uma solução menor será teoricamente seu tempo total de *failover*. Isso porque haverá menos estado a ser recuperado ou reconstituído, e portanto a réplica que assumiu poderá exercer em plenitude as funções do NameNode em menos tempo. Assim as subcategorias são:
 - (a) **Cold Standby:** Não há replicação do processo do NameNode. Corresponde ao par ativo-passivo descrito na Subseção 2.2.2. O estado *hard* é registrado para armazenamento estável acessível a outros nós do *cluster*. Com a falha do NameNode um novo processo deverá ser iniciado em outro nó a partir estado *hard*. O estado *soft* terá de ser reconstituído.
 - (b) **Warm Standby:** o estado *hard* é continuamente repassado para as réplicas que são processos vivos. Quando uma delas assumir a posição de ativa, ela ainda precisará recuperar o estado *soft*.
 - (c) **Hot Standby:** ambos os estados *hard* e *soft* são continuamente fornecidos às réplicas de modo que a nova réplica ativa possa prontamente assumir todas as funções do NameNode.
2. **Replicação Ativa:** quando uma variante da técnica de replicação ativa é usada para replicar o NameNode.

Introduzida essa classificação, podemos apresentar a Tabela 4.1, a qual compara as soluções em vários aspectos. Adiantamo-nos ao capítulo seguinte e adicionamos a solução proposta por esse trabalho à tabela. A coluna *Categoria* se refere às categorias citadas anteriormente; a coluna *Estratégia* se refere a como os dados são replicados; a coluna *Estado Replicado* se refere a que componentes de estado são replicadas; a coluna *Failover* se refere ao mecanismo de *failover* utilizado; e por fim a coluna *LOC* se refere ao número de linhas alteradas por essas soluções.

Proposta	Categoria	Estratégia	Estado Replicado	Failover	LOC
ContextWeb	Cold Standby	DRBD	espaço de nomes	Linux-HA	zero
IBM China	Warm Standby	mensagens de atualização de estado	espaço de nomes, leases	mecanismo próprio	desconhecido
NameNode <i>cluster</i>	Hot Standby	mensagens de atualização de estado	espaço de nomes, leases, localização de blocos	Linux-HA	5.570 [71]
AvatarNodes	Hot Standby	partilha NFS, mensagens duplicadas de DataNodes	espaço de nomes, leases, localização de blocos	manual	8.120 [38]
HDFS 2.0.0-alpha	Hot Standby	partilha NFS, mensagens duplicadas de DataNodes	espaço de nomes, leases, localização de blocos	manual	15.000+ [49]
Hot Standby Node	Hot Standby	propagação de <i>log</i> , mensagens duplicadas de DataNodes	espaço de nomes, localização de blocos	ZooKeeper	1.392
UpRight HDFS	Replicação Ativa	protocolo bizantino	espaço de nomes, leases, localização de blocos	não aplicável	2.696 [27]

Tabela 4.1: Quadro comparativo de propostas de alta disponibilidade para o HDFS.

Capítulo 5

O *Hot Standby Node*

Neste capítulo descreveremos uma proposta para alta disponibilidade do HDFS. Optou-se por desenvolver um *hot standby* para o NameNode. De maneira mais geral um *hot standby* é um nó servidor que a todo momento tenta manter uma cópia completa e atualizada do estado de seu nó primário. Consequentemente, quando um mecanismo de *failover* o comanda a assumir a posição de ativo, ele pode fazê-lo em um curto espaço de tempo, uma vez que há nenhum ou pouco estado para ser reconstruído. Portanto, com um *hot standby* podemos reduzir ao máximo o tempo em que o sistema de arquivos ficará indisponível após a falha do NameNode conforme a discussão do capítulo anterior. Obviamente esse período poderia ser inexistente se empregássemos replicação ativa para o servidor de metadados. No entanto isso impõe um requisito de determinismo ao NameNode. Isso só não requer muitas modificações no código, como também dificulta a manutenção, pois toda mudança deveria ser verificada por violações ao determinismo. Há também a imposição de restrições a projetos de novas funcionalidades, bem como menor possibilidade de paralelismo.

Durante a pesquisa o Backup Node se projetou como uma grande oportunidade para a implementação de um *hot standby*. Além de já replicar alterações no espaço de nomes, o fato de ser uma subclasse do NameNode lhe confere o potencial de ser capaz de tratar as requisições de clientes. Logo a proposta inclui o uso de uma versão modificada do Backup Node que doravante denominaremos de *Hot Standby Node*. Mas antes de esmiuçarmos as alterações feitas no HDFS para implementar o Hot Standby Node, faremos uma explanação acerca do funcionamento em baixo nível das interações entre o NameNode o Backup Node a fim de identificar e compreender que comportamentos úteis para Hot Standby Node foram herdados. Tal estudo foi também útil para a implementação, pois permitiu identificar que trechos código deveriam ser modificados para se atingir o objetivo.

5.1 Interações entre NameNode e Backup Node

O NameNode e o Backup Node já realizam algumas das interações que foram reaproveitadas para o Hot Standby Node. Antes de delineá-las, faz-se necessário apresentar os componentes arquiteturais do NameNode e do Backup Node, visto que as interações ocorrem entre esses componentes.

5.1.1 Arquitetura do NameNode e Backup Node

O papel e o funcionamento geral do NameNode do HDFS 0.21 (a versão em que se baseia o Hot Standby Node) foram descritos na Subseção 3.3.3. Nesta subseção detalharemos seus componentes e as interações entre eles. Na Figura 5.1 temos um diagrama de blocos da arquitetura do NameNode. Cada retângulo representa uma classe. Um retângulo contido dentro de outro indica que a classe do retângulo externo contém um objeto da classe do retângulo interno.

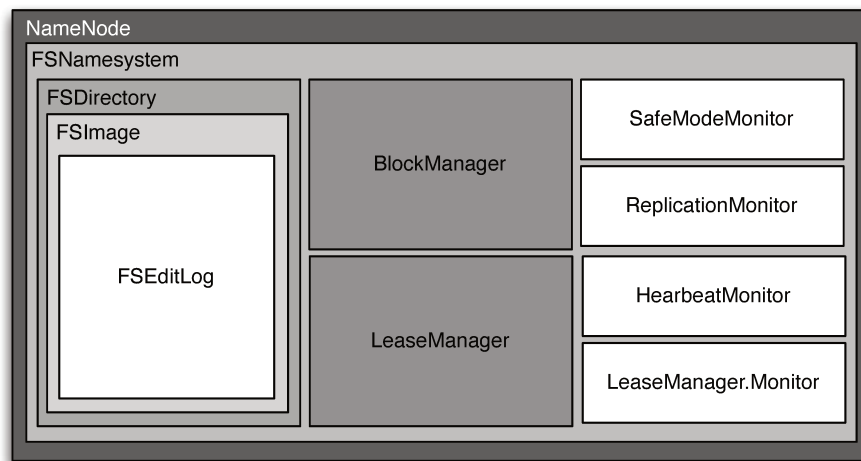


Figura 5.1: Diagrama arquitetural de blocos para o NameNode.

NameNode É a classe responsável por iniciar o processo do NameNode, tratar as chamadas de RPC e criar o **FSNamesystem**. No Backup Node essa classe é substituída por uma subclasse, a **BackupNode**, que provê configurações diferentes e um novo processo de iniciação, fazendo a sincronização inicial com o NameNode. A classe **BackupNode** também cria o *daemon Checkpointer* que executará periodicamente o processo de criação de um novo *checkpoint*.

FSNamesystem É a classe que de fato implementa o servidor de metadados. O **FSNamesystem** faz a validação das pré-condições antes de repassar uma operação para o **FSDirectory** e determina quando o *log* será persistido para o disco. Também é responsável por iniciar um conjunto de *threads* monitoras, como a **LeaseManager.Monitor** que remove *leases* expiradas, a **ReplicationMonitor** que gerencia o nível de replicação dos blocos, a **HeartbeatMonitor** que monitora quais DataNodes estão ativos, e a **SafeModeMonitor** que verifica quando é possível deixar o *safe mode*.

FSDirectory Implementa a árvore do sistema de arquivos e suas operações. É responsável por criar a trava global para serializar operações concorrentes e por registrá-las no *log*.

FSImage É responsável por carregar os *checkpoints* e criá-los a partir da fusão com o *log*.

FSEditLog É uma fachada que permite que o *log* do sistema de arquivos seja escrito para diferentes fluxos. Entre eles o **EditLogFileOutputStream** que guarda o *log* para arquivos e o **EditLogBackupOutputStream** que repassa as entradas para o Backup Node. Ambos são subclasses de **EditLogOutputStream**.

LeaseManager Esta classe gerencia, verifica e concede *leases* para clientes. Conta com a ajuda de **LeaseManager.Monitor** para lidar com *leases* expiradas.

BlockManager Administra réplicas de blocos, escolhendo DataNodes para recebê-las e processando mensagens de block-report e block-received. Conta com a ajuda do **ReplicationMonitor** para manter os níveis de replicação. O **BlockManager** também mantém o mapeamento entre um bloco e os DataNodes que hospedem suas réplicas.

5.1.2 Sincronização inicial e *checkpointing*

Ao iniciar, o Backup Node estabelece sua primeira interação com o NameNode visando sincronizar suas visões sobre o espaço de nomes. Inicialmente o Backup Node envia um *handshake* (Figura 5.2 - passo 1) para o NameNode a fim de verificar se ambos estão executando versões de software compatíveis. Em caso afirmativo o Backup Node procede ao registro (passo 2). O NameNode cria então um **EditLogBackupOutputStream** (passo 3) para propagar mudanças no espaço de nomes (a propagação será explicada na subseção seguinte). O registro fornece informações que permitem ao Backup Node determinar se ele está sincronizado. De qualquer forma o Backup Node inicia seu *daemon*

Checkpoint (passo 4). Caso o Backup Node não esteja sincronizado, o **Checkpoint** sinaliza ao NameNode que irá criar o um novo *checkpoint* (passo 5). Isso faz com que o NameNode passe a escrever em um novo *log* (passo 6), já que o atual será usado no *checkpoint*. O *daemon* de *checkpoint* recupera via HTTP os últimos *checkpoint* e *log* do NameNode e pede ao **FSImage** do Backup Node para combiná-los em um novo *checkpoint* (passos 7 e 8). O novo *checkpoint* é enviado via HTTP (passo 9) para o NameNode. O **Checkpoint** sinaliza então o término do processo, permitindo ao NameNode remover versões antigas do *checkpoint* (passo 10).

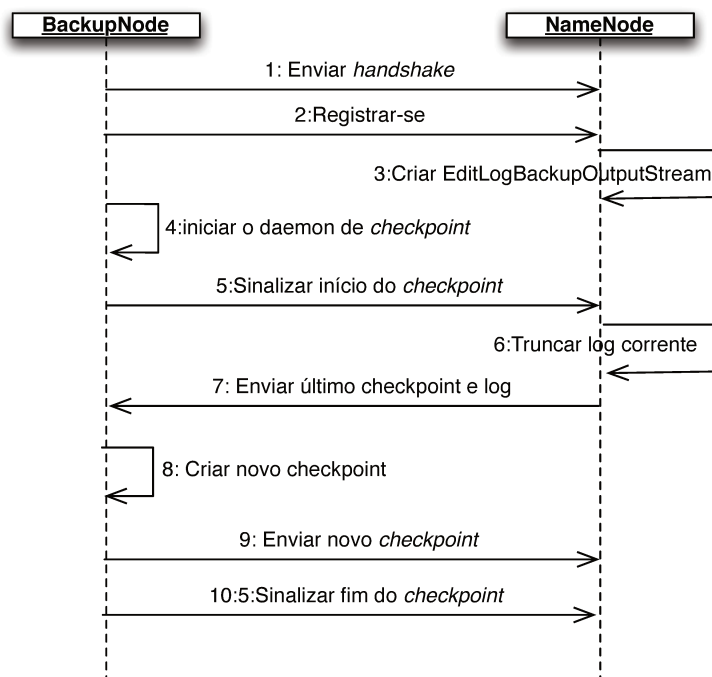


Figura 5.2: Sincronização inicial do Backup Node.

O **Checkpoint** irá executar os passos de 5 a 10 toda vez que perceber que o *log* atingiu um tamanho limite, quatro megabytes por padrão. Porém não executará o passo 7, uma vez que agora poderá usar o último *checkpoint* que gerou e seu próprio *log* para criar um novo *checkpoint*.

5.1.3 Propagação de mudanças no espaço de nomes

Como já mencionado no Capítulo 3, toda mudança feita no espaço de nomes é propagada ao Backup Node, o qual a aplica ao seu estado. Nesta subseção veremos esse processo em mais detalhes a partir do exemplo de uma operação de renomeamento de arquivos

ilustrada pela Figura 5.3. Nessa figura a sequência de chamadas de métodos se desenvolve de cima para baixo, da esquerda para a direita, com setas denotando chamada de métodos remotos.

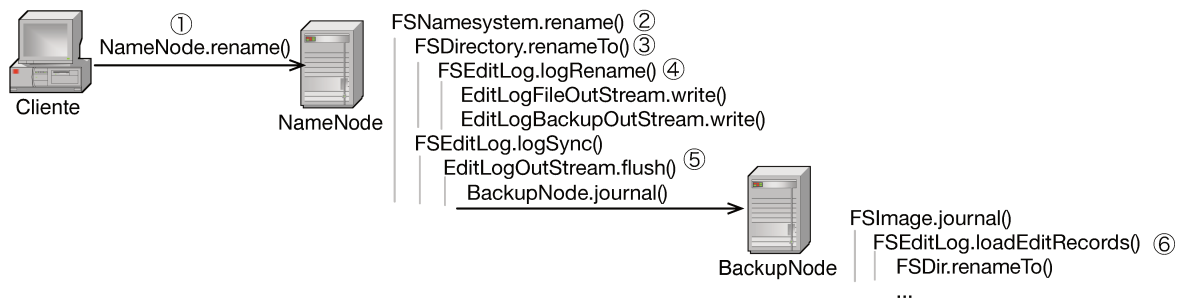


Figura 5.3: Propagação de alterações no espaço de nomes.

Quando o cliente deseja renomear um arquivo, ele faz uma requisição ao NameNode por meio de uma RPC ①. Após as validações de `FSNameSystem.rename()` ②, a requisição segue para árvore do sistema de arquivos: `FSDir.renameTo()` ③ adquire a trava do sistema de arquivos, realiza a alteração de nome no *inode* correspondente, adiciona uma nova entrada ao *log* (`FSEditLog.logRename()`) ④ e libera a trava. No exemplo da Figura 5.3, um arquivo (`EditLogFileOutputStream`) e o Backup Node (`EditLogBackupOutputStream`) são usados para armazenar o *log*. Neste ponto as entradas estão mantidas em *buffer* na memória. Ao retomar o controle, `FSNameSystem.rename()` força (`FSEditLog.logSync()`) as novas entradas a serem escritas para disco (`EditLogFileOutputStream.flush()`) e enviadas ao Backup Node (`EditLogBackupOutputStream.journal()`) ⑤. Isso é feito fora da trava do sistema de arquivos para fazer com que entradas de operações concorrentes sejam enviadas em lote, o que garante um melhor desempenho.

Ao receber o conjunto de entradas do *log* via RPC, o Backup Node irá lê-las e aplicá-las como se estivesse lendo o arquivo de *log* (`FSEditLog.loadEditRecords()`) ⑥. A partir de então o fluxo de execução é o mesmo que no NameNode, inclusive com o registro da operação no *log* próprio do Backup Node. Somente após todos `EditLogOutputStreams` terem confirmado a persistência é que uma reposta é enviada ao cliente. Portanto o mecanismo de propagação para o Backup Node é síncrono, garantindo que a consistência será sempre mantida. Caso um dos `EditLogOutputStreams` falhe, ele é excluído de todas as futuras operações. Para o Backup Node isso é um erro fatal.

5.2 Construção do Hot Standby Node

Reutilizando o Backup Node para a construção do Hot Standby Node herda-se um mecanismo de sincronização e propagação de alterações no espaço de nomes. Esse mecanismo de fato realiza a replicação do estado *hard* conforme definimos no capítulo anterior. Porém isso não é suficiente para o objetivo da solução. O *cluster* HDFS do Facebook é o maior em termos de capacidade de armazenamento com 2 mil nós, 21 PB de capacidade e 150 milhões de arquivos [20, 37]. Estima-se que uma teórica solução de alta disponibilidade usando o Backup Node do jeito que ele é — logo um *warm standby* de acordo com nossa classificação — teria um *failover* de pelo menos 20 minutos [20]. Esse é o tempo necessário para receber e processar os block-reports de todos os DataNodes e reconstituir o mapeamento das localizações dos blocos.

A solução proposta feita neste capítulo evolui o Backup Node para o Hot Standby Node por meio da:

- I - Replicação do estado *soft* para o Backup Node;
- II - Implementação de um mecanismo automático de *failover* que:
 - (a) detecta a falha do NameNode e inicia a transição;
 - (b) dissemina a informação sobre o novo servidor de metadados.

5.2.1 Replicação do estado *soft*

O Backup Node conhece quais arquivos compõem o sistema e seus respectivos blocos. Porém essas informações são insuficientes para que ele possa vir a atender clientes. O Backup Node desconhece quais DataNodes fazem parte do sistema, de quais réplicas eles dispõem, ou ainda quais clientes possuem uma *lease*. Essas informações são imprescindíveis para satisfazer requisições de leitura e escrita e manter a consistência dos arquivos. Para o Hot Standby Node adotou-se estratégias diferentes para cada subcomponente do estado *soft*. Descrevemo-nas no que se segue.

Informações sobre DataNodes e réplicas

É somente a partir das mensagens advindas dos DataNodes que o NameNode toma ciência de onde estão as réplicas de blocos no *cluster* e de quais DataNodes podem receber mais blocos. À primeira vista uma solução natural seria fazer o NameNode encaminhar as mensagens dos DataNodes para o Hot Standby Node por meio do *log* ou outros canais (Figura 5.4(a)). No entanto, com *clusters* compostos de dezenas a milhares de DataNodes e cada um enviando heartbeats a cada três segundos o tráfego gerado na rede poderia

ser muito intenso. A solução seria ter um enlace dedicado entre o NameNode e o Hot Standby Node. Todavia um enlace dedicado pode não ser factível em ambientes em que não se detém a posse da infraestrutura, tais como ambientes de computação em grade ou elástica. Outra alternativa é consolidar as mensagens para reduzir o fluxo. Contudo a consolidação desviará memória e capacidade de processamento que outrora seriam usadas para o tratamento de requisições de clientes, impactando o desempenho do NameNode. A trava global do HDFS teria de ser mantida por mais tempo, segurando outras operações. Se as mensagens fossem enviadas assincronamente, teriam de ser armazenadas em *buffers*, consumindo memória que seria usada para os metadados. De fato, o NameNode Cluster, que foi exposto no Capítulo 4 e que usa essa estratégia, relata uma diminuição de 85% no desempenho de operações que atuam estritamente sobre metadados. De qualquer forma, após uma falha do NameNode não há garantias de que Hot Standby Node terá as informações mais recentes sobre DataNodes. O estado dos DataNodes depende exclusivamente deles e pode ter-se alterado após a queda do NameNode.

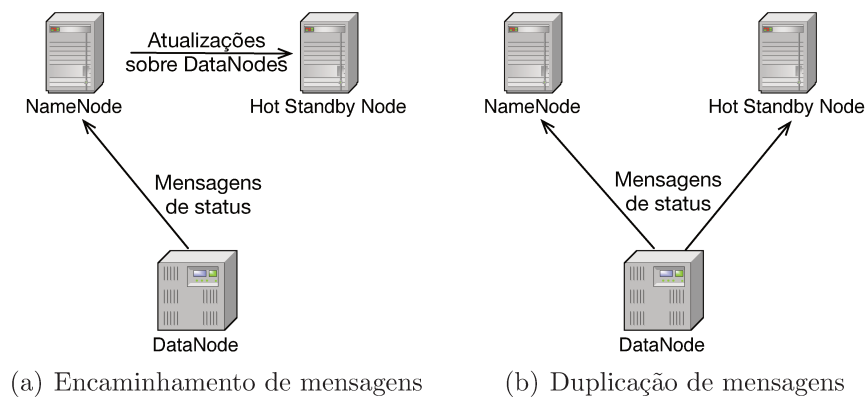


Figura 5.4: Possibilidades para replicar informações sobre blocos e DataNodes.

Uma alternativa é modificar os DataNodes para também enviarem suas mensagens ao Hot Standby Node (Figura 5.4(b)). Essa estratégia tem a vantagem de distribuir o tráfego extra e o trabalho de replicação de forma razoavelmente uniforme pelo *cluster*. Outra vantagem é de que como o NameNode não está envolvido na replicação o impacto sobre ele é mínimo. Em contraste com a solução anterior, a implementação é mais simples e mesmo em face de falha do NameNode o Hot Standby Node poderá contar com informações atualizadas sobre os DataNodes e réplicas de blocos.

Devido a esse conjunto de razões optou-se por essa estratégia de duplicar as mensagens de DataNodes. Esse foi também o caminho escolhido pelo Avatar Nodes do Facebook. Uma vez que o código da modificação dos DataNodes feito pelo AvatarNodes no HDFS 0.20 estava disponível e tinha sido amplamente testado em ambientes de produção

no Facebook, resolvemos portá-lo para a solução Hot Standby Node, a qual é baseada em HDFS 0.21.

O DataNode possui uma *thread* que é responsável por toda comunicação com o NameNode. Essa *thread* no início de sua execução faz o registro do DataNode com o NameNode. E depois segue então a executar o laço que constrói e envia as mensagens. A modificação do Avatar Nodes consiste em simplesmente duplicar essa *thread* de forma que uma de suas instâncias sirva ao Hot Standby Node. As duas *threads* executam de forma independente. Portanto não há garantias que as mensagens enviadas ao NameNode serão as mesmas que as enviadas ao Hot Standby Node. Logo suas visões sobre o *cluster* poderão divergir. Isso poderia ser evitado se as mensagens fossem disseminadas por meio de um protocolo de *broadcast* atômico, o que seria custoso. Porém tal consistência rígida não se faz necessária. Já é difícil manter o sincronismo entre os DataNodes e o NameNode conforme estes primeiros entram e deixam o *cluster*, falham e reiniciam. E em *clusters* grandes esses eventos são muito frequentes. Em um cluster da Yahoo com 4 mil nós [91] com a taxa de erro apresentada no Capítulo 4 isso significa pelo menos 6 eventos dessa natureza ocorrem todos os dias. O HDFS já é bastante robusto para lidar com tais situações. Se o DataNode falha em enviar uma mensagem, a própria camada de RPC retenta o envio. Se a comunicação com o NameNode é perdida, o DataNode se registrará novamente e enviará um novo block-report.

Por outro lado, enquanto o Hot Standby Node está desempenhando o papel de servidor passivo, ele é mantido em *safe mode*. Ou seja, ele não pode realizar nenhuma mudança no espaço de nomes ou enviar qualquer comando aos DataNodes. Deste modo sua visão particular do *cluster* não terá alguma influência sobre as operações do sistema de arquivos. Quando se tornar o servidor de metadados ativo, ele pode ter eventualmente perdido alguma mensagem de um DataNode (assim como o mesmo pode ocorrer com o NameNode). Isto poderá fazê-lo pensar que algum bloco está sub-replicado e ordenar que uma réplica extra seja criada quando não seria necessário. Note porém que dados não serão perdidos.

Outra particularidade da modificação nos DataNodes feita por Avatar Nodes é como ela lida com o fato de os blocos de um arquivo serem apenas registrados no *log* quando o arquivo é fechado. Isso significa que o Hot Standby Node apenas ficará ciente de que um certo bloco pertence a um certo arquivo quando este arquivo for fechado. Assim, conforme os blocos de um arquivo são escritos e os DataNodes enviam mensagens de block-received ao Hot Standby Node sobre esses novos blocos, o Hot Standby Node lhes dará como inválidos, pois não sabe dizer a que arquivos eles pertencem. A resposta foi fazer o *standby* devolver esses blocos nas repostas para mensagens de block-received para que os DataNodes possam retentá-los nas próximas mensagens até que o arquivo seja fechado.

As modificações nos DataNodes e a alteração para devolver os blocos não reconhecidos em mensagens de *block-received* reduziram os tamanhos das mensagens conforme pode ser apreciado na Tabela 5.1.

Mensagem	Tamanho (bytes)			
	HDFS 0.21		Hot Standby Node	
	Requisição	Resposta	Requisição	Resposta
block-received	457	95	425	51
heartbeat	324	148	289	148

Tabela 5.1: Exemplos de tamanhos de mensagem para o HDFS 0.21 e para a solução Hot Standby Node.

Os valores da Tabela 5.1 foram obtidos a partir de *traces* TCP de uma instalação de HDFS 0.21 e um do Hot Standby Node. O conteúdo dos *traces* foi útil para elucidar as diferenças de tamanho. Todo DataNode possui um identificador de armazenamento que é uma cadeia de caracteres. Esse identificador é enviado em toda mensagem de *block-received* e *heartbeat*. O formato desse identificador tornou-se mais compacto depois das alterações, resultando em uma economia de mais de 30 bytes para ambas as mensagens. Quanto a redução das repostas para as mensagens de *block-received* da Tabela 5.1, ela se deve a particularidades do protocolo de RPC do Hadoop. Antes das mudanças o valor de retorno das mensagens de *block-received* era do tipo *void* e agora é um arranjo de blocos. No caso específico da Tabela 5.1, o valor de retorno das mensagens de *block-received* foram todos arranjos vazios. Acontece que o protocolo RPC do Hadoop representa tipos usando o nome totalmente qualificado da classe do objeto a ser empacotado. O nome totalmente qualificado de uma classe inclui seu pacote, o que pode torná-lo longo. Consequentemente representar um valor de retorno de tipo *void* acaba usando 44 bytes a mais do que um arranjo vazio de blocos porque o nome do pacote da classe *Block* é mais curto do que da classe usada para o tipo *void*.

Leases

Optou-se por não replicar os dados sobre *leases*. Esta decisão se deve à uma peculiaridade da implementação do HDFS 0.21. O NameNode concede uma *lease* ao cliente quando um arquivo é aberto para escrita. Conforme o arquivo é escrito blocos adicionais são requisitados ao NameNode, porém essa informação fica apenas residente na memória do NameNode. Conforme já dito, quando então a escrita termina e o cliente fecha o arquivo, o NameNode cria uma entrada no *log* para registrar o fechamento do arquivo juntamente com seus blocos e tempo de modificação. Assim se o NameNode falhar enquanto um arquivo é escrito, os blocos adicionados a ele desde a última vez em que foi aberto serão

perdidos, pois não haverá nenhum registro permanente que associe os blocos ao arquivo. Consequentemente, o cliente terá de reiniciar a escrita, o que exigirá a requisição de uma nova *lease*. Portanto a *lease* concedida antes da falha não será reutilizada e logo não há vantagens em replicá-la.

Este comportamento é de certa forma tolerável pelas principais aplicações que utilizam o HDFS [17]. Hadoop MapReduce [10], o *framework* de processamento distribuído, retentará qualquer tarefa que falhe. HBase [45], o banco de dados versionado orientado a colunas do projeto Hadoop, não considerará uma transação efetivada até que seu conteúdo seja persistido para um arquivo do HDFS.

5.2.2 Mecanismo automático de *failover*

O Hot Standby Node deve ser capaz de reagir rapidamente à falha do NameNode de forma a garantir o menor tempo de indisponibilidade do sistema. Para isso criou-se um mecanismo automático de *failover* baseado em outro componente do projeto Hadoop: o ZooKeeper.

ZooKeeper [11, 52] é um serviço de coordenação distribuído altamente disponível. ZooKeeper mantém um banco de dados replicado entre um quórum de servidores (o *ensemble*) por meio de um protocolo de *broadcast* atômico próprio, o *Zab* [55, 86]. O banco de dados é estruturado em forma de uma árvore de *znodes* como poder ser visto na Figura 5.5. Znodes podem armazenar dados e toda operação sobre eles ou sobre a árvore é feita atomicamente. Znodes irmãos podem ser criados com seu nome sucedido por um contador incremental, o que permite estabelecer uma ordem total entre eles. Znodes folhas podem ser criados como efêmeros. Isto significa que esse znodes existem apenas enquanto a sessão do cliente que os criou estiver ativa. Um cliente do ZooKeeper pode se registrar para ser notificado de qualquer evento (criação, escrita, remoção) que venha ocorrer com determinado nó ou filhos dele. Todas essas características permitem criar vários protocolos distribuídos como eleições, travas, primitivas de grupo, filas, entre outros.

O cliente do ZooKeeper não faz nenhum tratamento de erro, ficando isso a carga da aplicação que o usa. Portanto implementou-se uma biblioteca para intermediar todas as interações com o ZooKeeper. A biblioteca implementa um protocolo de retentativas com recuo exponencial. Mesmo com esse protocolo há ainda o risco de falhas parciais, na quais o cliente enviou a mensagem porém não sabe se ela atingiu o *ensemble* ou ainda se foi aplicada. Assim, antes de retentar a operação deve-se verificar se ela foi efetivada. Para evitar esse problema todas as interações da solução são idempotentes, isto é, podem ser aplicadas repetidas vezes sem prejuízo.

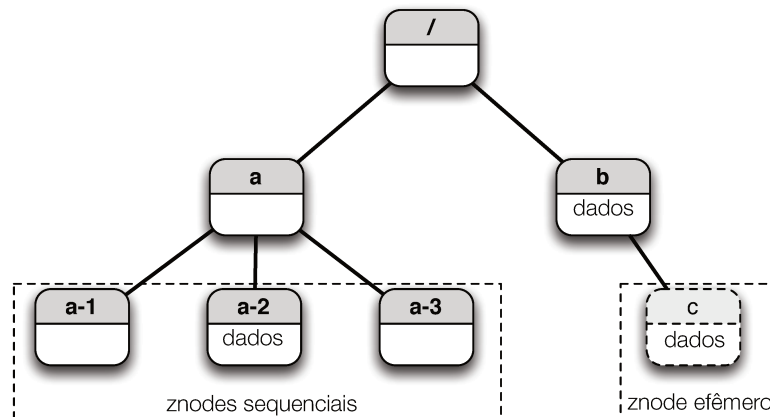


Figura 5.5: Exemplo de árvore do ZooKeeper.

Detecção de falhas

Para manter sua sessão com o ZooKeeper, o cliente deve mandar mensagem do tipo “*Estou vivo*” de forma periódica ao *ensemble* antes que o *timeout* de sua sessão estoure. Se o processo que executa o cliente falhar, ele obviamente não enviará mais mensagens e por fim sua sessão expirará. Com base nesse princípio podemos usar o ZooKeeper como um detector de falhas não confiável. Todo processo a ser monitorado deve criar um znode efêmero. Todo processo que deseja monitorar outro processo deve se registrar para ser notificado da remoção do znode. A ausência do znode efêmero ou a notificação de sua remoção caracterizam seu respectivo processo como *suspeito*. Essa é essência do Failover Manager.

O Failover Manager é um componente adicionado tanto ao processo do NameNode como ao do Hot Standby Node. Ele tem por responsabilidades manter a comunicação com o ZooKeeper, monitorar nós por falhas e iniciar o processo de transição. Quando o Failover Manager inicia, ele cria um znode interno se esse já não existir (Figura 5.6(a)). Esse znode é usado para representar o grupo de servidores de metadados e para armazenar o endereço de rede do nó atualmente no papel de servidor ativo. Iremos nos referir a esse znode como o “*znode do grupo*” a partir desse ponto. Pois bem, o passo seguinte do Failover Manager é criar um znode efêmero como filho do *znode do grupo* para representar o nó em que ele está a executar (Figura 5.6(b)). Desse ponto em diante o fluxo do Failover Manager segue caminhos distintos para o NameNode e para o Hot Standby Node. Para o NameNode o Failover Manager irá escrever seu endereço de rede no *znode do grupo* (Figura 5.6(c)). Para o Hot Standby Node o Failover Manager irá se registrar para ser notificado da remoção do znode do NameNode.

Quando o NameNode falha o *timeout* de sua sessão irá estourar (Figura 5.6(d)). Quando o estouro ocorre o ZooKeeper define sua sessão como expirada e remove seu znode. O ZooKeeper então avisa o Hot Standby Node do ocorrido (Figura 5.6(e)).

Transição para o estado ativo

Ao ser notificado da remoção do znode do NameNode, o Hot Standby Node inicia os procedimentos de *failover* que o transformarão no servidor de metadados ativo. Os procedimentos de *failover* consistem em:

1. **Parar a *thread* de limpeza de blocos inválidos**

Além de devolver os blocos não reconhecidos para os DataNodes, o Hot Standby Node adiciona esses blocos à uma fila de blocos inválidos. Isto ocorre porque é o comportamento original para blocos não reconhecidos. E de fato alguns desses blocos podem ser inválidos. Esse comportamento pode gerar dois problemas. O primeiro é que, dependendo da carga ao qual o sistema de arquivos é submetido, muito blocos podem se acumular na fila de inválidos preenchendo toda memória. Outro é que alguns blocos ainda podem permanecer na fila de inválidos após o *failover*. Nesse momento o Hot Standby Node irá pedir aos DataNodes que removam todos os blocos que constem na fila. Para evitar esse dois problemas a fila de blocos inválidos é limpa a cada minuto para evitar o primeiro problema e mais uma vez antes do término do *failover* para evitar o segundo. Os blocos que são realmente inválidos poderão ser novamente identificados como tais após o envio de novos block-reports.

2. **Restaurar as configurações de *lease***

Durante o *failover* o limite *hard* é restaurado para seu valor original, já que o Hot Standby Node passará a administrar as *leases*. Durante o processamento normal, o Backup Node (e por extensão o Hot Standby Node) define o limite *hard* das *leases* para infinito. Essa medida se deve ao fato do Backup Node criar uma nova *lease* quando recebe uma operação de abertura de arquivo, devido a um comportamento herdado do NameNode. Uma vez que o Backup Node não recebe outras informações sobre *leases*, essa *lease* expiraria e poderia criar problemas quando novas operações sobre o arquivo aberto fossem processadas. Essa *lease* será mantida até que o arquivo seja fechado.

3. **Fechar todos os arquivos abertos**

Todos os arquivos que se encontram abertos no momento do *failover* são fechados e suas *leases* associadas são removidas para permitir que os clientes reiniciem suas escritas.

4. Sair do *safe mode*

O Hot Standby Node é mantido forçosamente em *safe mode* para preveni-lo de enviar qualquer comando para os DataNodes, o que poderia causar perdas de dados. Quando o Hot Standby Node passar a ser o servidor de metadados ativo, ele deve poder comandar os DataNodes para realizar as atividades do sistema de arquivos.

5. Atualização do *znode* do grupo

Ao final do *failover* o Hot Standby Node escreverá seu endereço de rede no *znode do grupo* (Figura 5.6(f)) para sinalizar aos demais nós que é novo servidor de metadados.

Determinação do servidor ativo

Uma forma de prover uma transição transparente para os clientes do HDFS é atuar em nível de rede. Alterando o mapeamento DNS ou usando técnicas como IP Virtual, clientes podem usar um único endereço para o NameNode ou Hot Standby Node sem estar cientes de qual é o servidor de metadados ativo no momento. Mas há situações em que esta técnica não poderá ser utilizada, principalmente quando não se é dono da infraestrutura. Há certas exceções nesse caso, mas nem sempre são vantajosas. Amazon EC2 oferece *IPs elásticos*, IPs que podem ser mapeados dinamicamente para uma instância de máquina virtual qualquer. Porém tais IP são externos à rede da Amazon. Isso significa que os dados terão de seguir uma rota que é mais custosa e que pode não ser a mais eficiente [5, 98].

Outra alternativa é atuar no nível de software. Na solução Hot StandbyNode o NameNode escreve seu endereço de rede no *znode do grupo*. O Hot Standby Node faz o mesmo ao final do *failover*. Isto posto, a determinação do servidor de metadados atual resume-se a consultar o valor do *znode do grupo*. Para manter essa informação atualizada basta se registrar no ZooKeeper para ser notificado de qualquer alteração no *znode do grupo*.

A primeira alternativa não requer mudanças no cliente do HDFS. No protótipo não foram feitas mudanças no cliente do HDFS para apoiar a segunda alternativa porque o foco era medir o impacto da solução sobre os nós servidores. Além do mais, como escritas em progresso serão perdidas é melhor que o tratamento do *failover* seja no nível dos aplicativos, pois estes saberão tratar melhor eventuais falhas.

Discussões

Há ainda dois outros aspectos do mecanismo de *failover* que devem ser discutidos:

Definição do *timeout* do ZooKeeper A escolha do valor para o *timeout* da sessão do cliente do ZooKeeper deve ser feita criteriosamente de forma a minimizar suspeitas

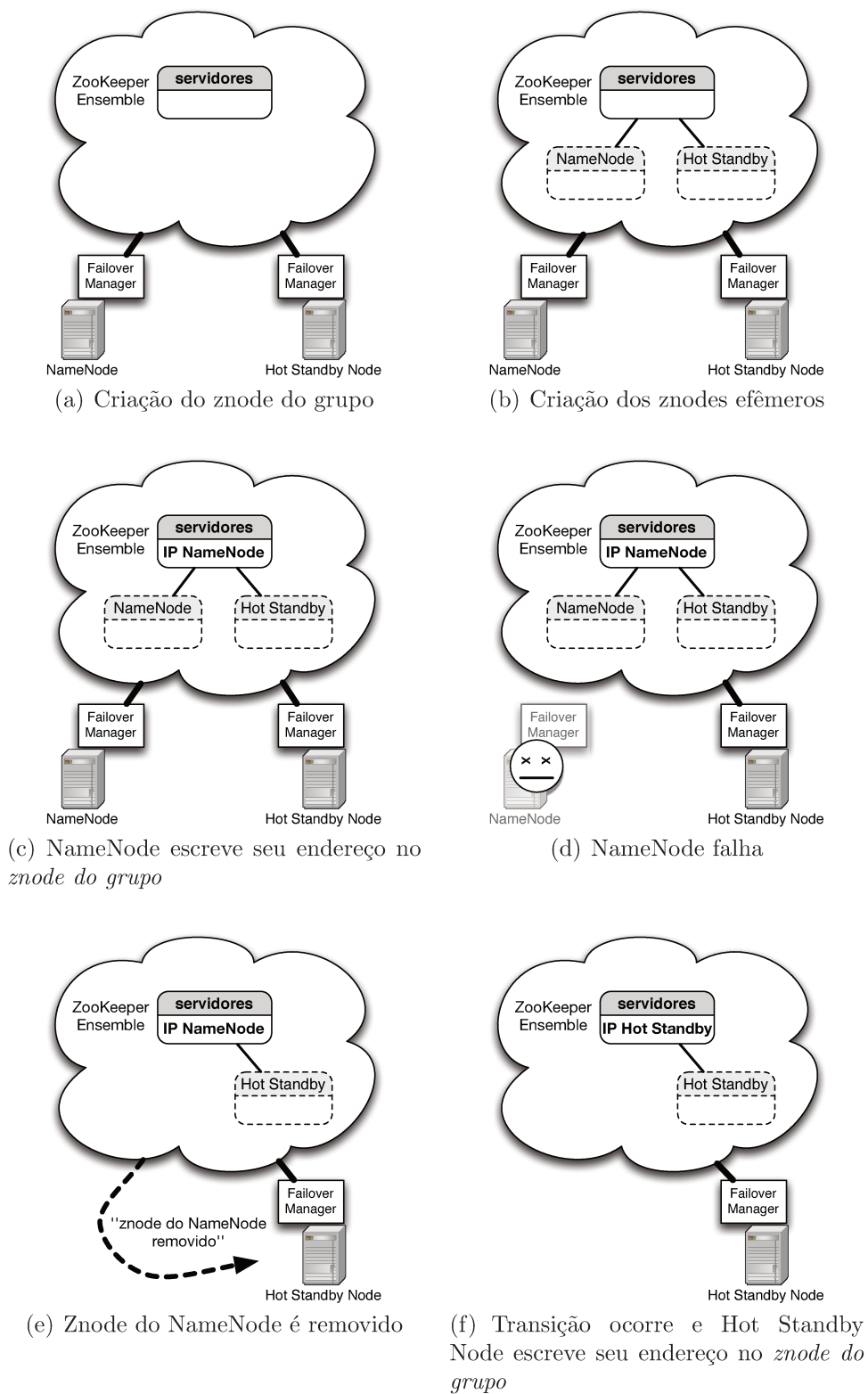


Figura 5.6: Ciclo de vida do Failover Manager.

incorretas de falhas. A latência da rede é o principal fator a ser considerado, pois o *timeout* não pode ser menor que o tempo máximo necessário para mensagem sair do cliente e chegar ao ZooKeeper. Do contrário obteremos um falso positivo. Entretanto é difícil na prática determinar a latência máxima. Trabalha-se então com algum valor que garanta que uma boa porcentagem das mensagens sejam entregues no tempo estipulado. Outro fator a ser considerado é o coletor de lixo do Java. Alguns algoritmos de coleta de lixo podem eventualmente parar a execução de todas as *threads* (*stop-the-world*), ocasionando atrasos no envio da mensagens. Outra condição que poder gerar atraso no envio das mensagens é a virtualização porque que uma máquina virtual pode não conseguir a fatia de processamento da máquina física necessária para o envio em tempo das mensagens.

Antecipando-se ao Capítulo 6, no ambiente de testes usado nos experimentos, Amazon EC2 [3], observou-se latência máxima de pouco mais de 500 milissegundos. Quanto a virtualização, o *stolen time*, ou seja, o tempo que em se precisaria usar a CPU física mas ela estava sendo usado por outra máquina virtual ou pelo hipervisor atingiu marcas de 5.6% do tempo total de execução. Isso significa que uma tarefa que em um ambiente normal tomaria 100 segundos, poderia durar 106 segundos no ambiente virtualizado. No entanto o fator mais impactante foi o coletor de lixo do Java, por razões que serão discutidas no Capítulo 6. As configurações tiveram de ser alteradas para fazer o coletor de lixo usar 8 *threads* e o algoritmo *Concurrent Mark-Sweep*. O algoritmo *Concurrent Mark-Sweep* não realiza *stop-the-world*, entretanto em situações mais drásticas de memória ele pode para alguma *thread* para realizar a coleta.

Particionamento da rede O particionamento de rede pode causar o problema de *split-brain*: ambos servidores de metadados pensam estar ativos, podendo gerar corrupção de dados. O particionamento dentro do *ensemble* já é tratado pelo próprio ZooKeeper. Em linhas gerais, o protocolo do ZooKeeper é baseado em quórum simples, logo apenas a partição que detiver a maioria do servidores sobreviverá. O NameNode e o Hot Standby Node podem inequivocamente determinar qual deles dever ser o servidor de metadados ativo consultando o ZooKeeper. Caso algum deles perca sua conectividade com o ZooKeeper e não consiga restabelecê-la antes de que sua sessão expire, o nó irá se desativar automaticamente. A razão para isso é que como a sessão expirou, o nó foi considerado suspeito de falha e conseqüentemente seu par assumiu a posição de ativo.

5.2.3 Mudanças para captura de métricas

O HDFS possui um sistema de coletas de métricas com vários parâmetros interessantes para análises de desempenho. Para os DataNodes é possível por exemplo dar-se a conhe-

cer o número de blocos lido e escritos, block-reports e heartbeats enviados, número de conexões com clientes, etc. Por padrão as métricas são produzidas a cada intervalo de 10 segundos. O fluxo gerado de métricas é voltado para ser exibido em tempo real em sistemas de monitoramento de *clusters* como o Ganglia [40]. O fluxo de métricas pode também ser redirecionado para arquivos. Entretanto a análise desses arquivos é um tanto complicada devido a ausência de marcas de tempo que permitam dizer em qual momento uma dada métrica foi colhida. Nesse sentido modificou-se o sistema de métricas do Hadoop para incluir marcas de tempo. Adicionou-se também uma nova métrica para revelar o tempo gasto com a transição do Hot Standby Node para o estado ativo.

5.3 Arquitetura geral e ciclo de vida

Com base nas decisões de projeto explicitadas na seção anterior produziu-se um protótipo de uma solução de alta disponibilidade para o Hadoop Distributed File System. O protótipo baseia-se no HDFS 0.21 e usa ZooKeeper 3.3.3. Foram modificadas 28 linhas, adicionadas 1155 e removidas 209. Ou seja, foram feitas alterações em apenas 0,18% da base de código original. A divisão do esforço de código está representada na Tabela 5.2.

	Modificadas	Adicionadas	Removidas
Alterações nos DataNodes	25	662	209
Transição e Métrica	3	493	0
Failover Manager	0	320	0
Total	28	1155	209

Tabela 5.2: Divisão do esforço de código em linhas.

A maior parte do esforço se concentrou no porte das mudanças feita pela solução AvatarNodes nos DataNodes. “Transição e Métrica” se referem às alterações no NameNode e no Backup Node para realizar a transição e às mudanças feitas no sistema de métricas para adicionar marcas de tempo e a métrica de *failover*. O Failover Manager compreende a camada de comunicação com o ZooKeeper e o código disparo da transição.

A arquitetura da solução Hot Standby Node pode ser apreciada na Figura 5.7. Trata-se de um par primário-*backup*, tendo inicialmente o NameNode como primário e o Hot Standby Node como *backup*. Ambos estão conectados ao *ensemble* de servidores ZooKeeper por meio do Failover Manager. Cada um dos DataNodes está também conectado tanto ao NameNode como ao Hot Standby Node.

O ciclo de vida da solução Hot Standby Node é apresentado no diagrama de sequência da Figura 5.8.

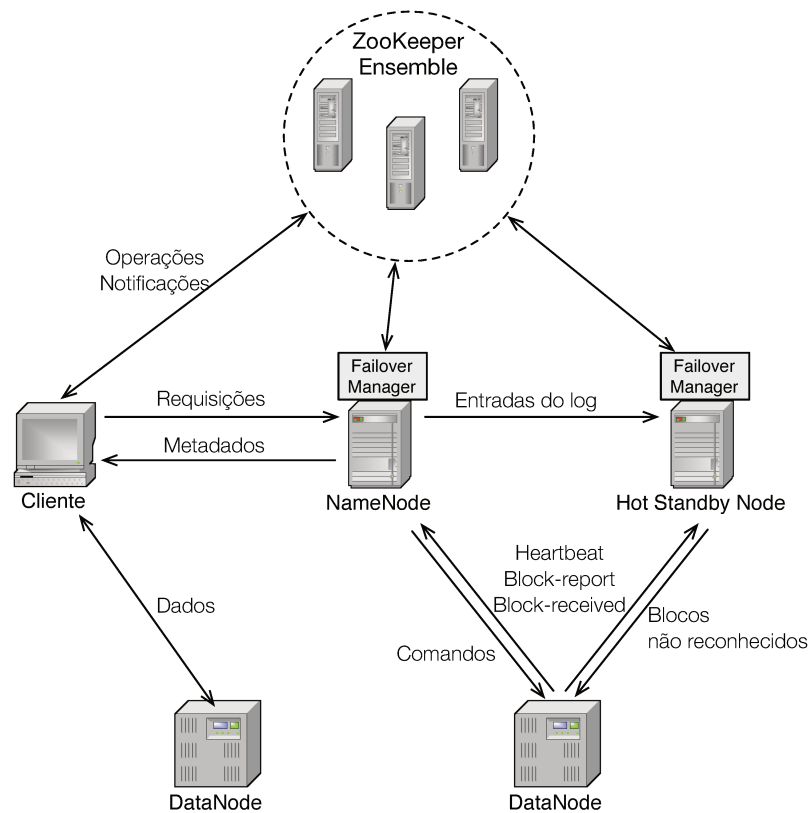


Figura 5.7: Arquitetura do Hot Standby Node.

O código do protótipo implementado encontra-se disponível sob licença livre Apache 2 no serviço de hospedagem de repositórios de código Github no seguintes endereços:

- <https://github.com/aoriani/hadoop-common> — código comum do Hadoop.
- <https://github.com/aoriani/hadoop-hdfs> — código específico do HDFS.

5.4 Limitações do protótipo construído

O protótipo construído possui as seguintes limitações:

1. Apenas uma falha é tolerada

A solução por ser baseada no Backup Node incorpora suas restrições. O NameNode aceita o registro de apenas um Backup Node, o que faz sentido pois sua função é produzir *checkpoints*. O NameNode poderia ser modificado para aceitar múltiplos

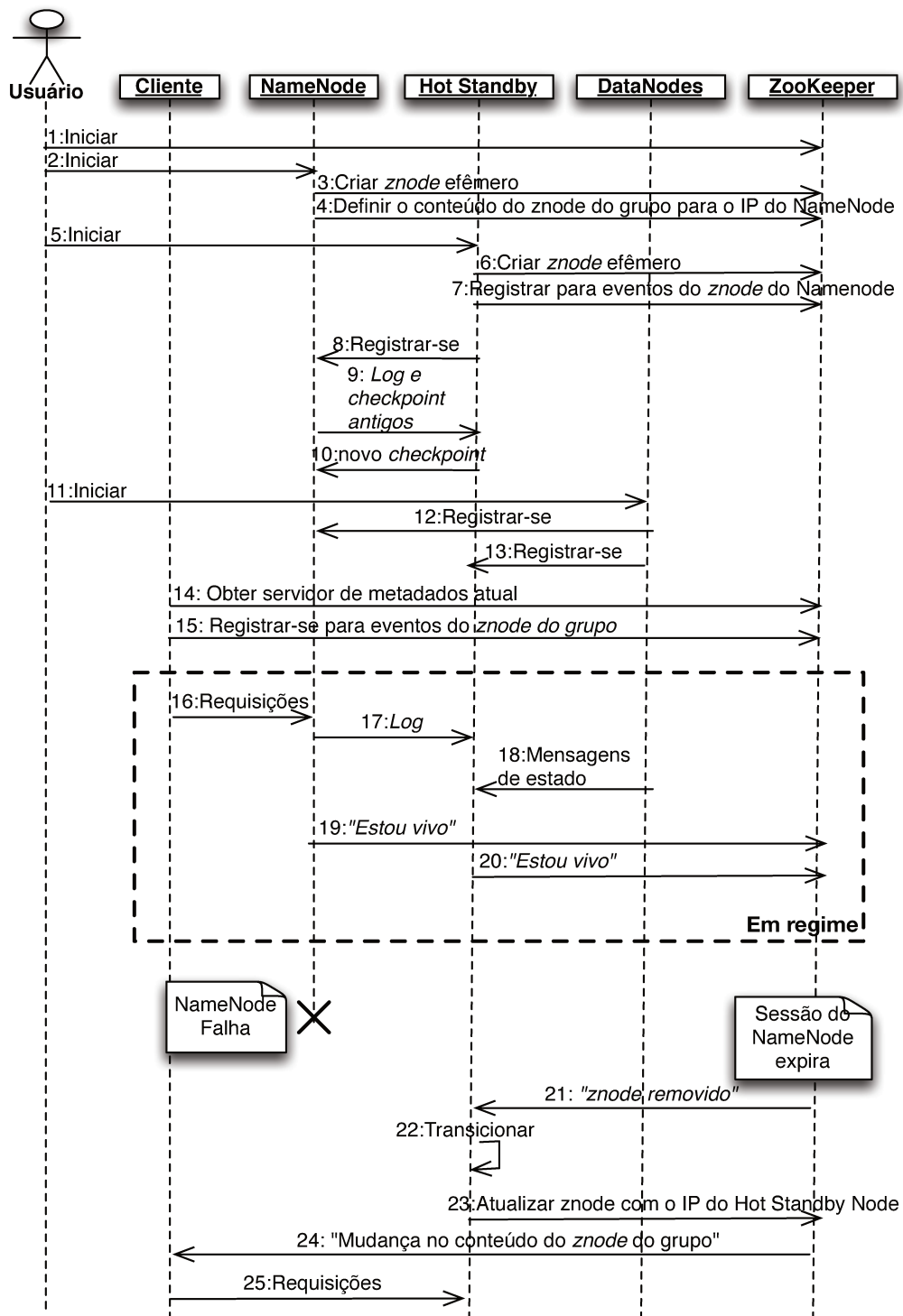


Figura 5.8: Ciclo de vida da solução Hot Standby Node.

Hot Standby Nodes e o ZooKeeper poderia auxiliar no gerenciamento das réplicas e na eleição durante o *failover*.

2. Não implementa reconfiguração

Outra característica herdada devido ao reuso do Backup Node é a assimetria entre o NameNode e o Hot Standby Node. O NameNode aceita conexão do Hot Standby Node mas não pode se conectar a ele. Por sua vez o Hot Standby Node se conecta ao NameNode mas não pode receber conexões. Essa falta de simetria impede o NameNode de se tornar um *hot standby* para o Hot Standby Node depois de se recuperar. A evolução natural será ter um único tipo de servidor de metadados que pode atuar tanto como primário ou como *backup*.

3. O Hot Standby Node não atua como *checkpoint helper*

Existe um defeito registrado no tíquete ainda aberto *HDFS-1989 - When checkpointing by backup node occurs parallely when a file is being closed by a client then Exception occurs saying no journal streams* [107] que pode ocasionalmente provocar a falha do Hot Standby Node durante um *checkpoint*. Como forma de contornar esse problema o Hot Standby Node faz apenas o *checkpoint* inicial durante a sincronização com o NameNode.

4. Escritas não podem ser continuadas após *failover*

O NameNode não registra no *log* os blocos de um arquivo conforme eles são alocados. Assim, em caso de falha do NameNode todos os blocos adicionados ao arquivo desde seu último fechamento serão perdidos. Há algumas sutilezas que dificultam a implementação do suporte à continuação de escritas após falhas do servidor de metadados. Blocos alocados não necessariamente significam efetivados. Devido a falhas blocos podem ser abandonados ou assumir novas identidades. E para ser registrado no *log*, o conteúdo do bloco deve estar no DataNode e não no *cache* do cliente. Isso exige uma operação de *flush*, o que pode impactar o desempenho de forma considerável [61, 85]. Uma solução seria implementar uma versão mais leve dessa chamada em que as entradas do *log* fossem propagadas para o Hot Standby Node, porém os dados não são persistidos nos discos dos DataNodes.

5.5 Comparação com outras soluções

Sendo um solução *hot standby* ela tem uma óbvia vantagem sobre soluções *cold* e *warm standby* como as desenvolvidas pela ContextWeb e pela IBM China. Como visto na Seção 5.2, o tempo para reconstruir o mapeamento entre blocos e DataNodes pode ser

longo em *clusters* grandes. Porém a solução da IBM China tem a vantagem de poder tolerar mais falhas.

Essa mesma vantagem sobre a solução Hot Standby Node está presente no NameNode Cluster. Porém no NameNode Cluster todo esforço de replicação se concentra no mestre. O mestre, que já é o único responsável por realizar as operações sobre metadados, acaba sendo também o único responsável pelas atividades de replicação. O excesso de funções pode tornar o mestre um gargalo para o sistema. O fato do mestre repassar as informações dos DataNodes para os escravos garante que mestre e escravos compartilhem a mesma visão sobre o *cluster*. Entretanto isso não garante que seja a visão mais atual, visto que os estados do DataNodes são independentes do mestre.

Com a escolha de fazer os próprios DataNodes mandarem suas mensagens de estado para o *standby*, a solução Hot Standby Node, Avatar Nodes e HDFS 2.0 conseguem distribuir o fardo da replicação por todo o *cluster* sem onerar o servidor de metadados. A versão mais atual do Avatar Nodes e o HDFS 2.0 vão ainda mais além. Elas permitem que escritas sejam continuadas, permitindo que *failover* sejam totalmente tratados pelo cliente HDFS, deixando-o mais transparente para as aplicações. Porém elas não contam com um mecanismo automático de *failover*. A grande diferença entre o Hot Standby Node e as duas outras soluções reside em como as alterações sobre metadados são propagadas. Avatar Nodes e HDFS 2.0 optam por réplicas do NameNode que se mantêm continuamente lendo o *log* a partir de um dispositivo de armazenamento compartilhado. Há então uma transferência do problema da disponibilidade do HDFS para esse dispositivo. O dispositivo tem de ser estável, confiável e altamente disponível, o que implica em um custo maior para as instalações de HDFS. Nós optamos por reutilizar o Backup Node, o que nos garantiu um mecanismo de replicação simples, síncrono, bem testado e que não carece de hardware especial. Além disso a latência de replicação é menor: apenas uma transferência de rede contra uma transferência para o dispositivo de armazenamento seguido de uma transferência de rede, como é no caso de Avatar Nodes.

O UpRight-HDFS consegue mascarar completamente falhas por empregar replicação ativa. Sua proteção trazida pela tolerância a falhas bizantinas poder ser nulificada pela falta de mecanismos de segurança na versão de HDFS em que foi implementado, embora hoje a questão da segurança no HDFS tenha melhorado. A necessidade de servidores deterministas restringe o uso de paralelismo, a principal estratégia para melhoria de desempenho de sistemas distribuídos.

Capítulo 6

Experimentos

Quando se avalia uma solução de alta disponibilidade dois questionamentos são inevitáveis:

1. Qual o custo adicional imposto pela solução sobre o desempenho do sistema em relação à sua versão original?
2. Qual é o tempo de reação do novo sistema? Em outras palavras, em quanto tempo após a falha clientes poderão fazer novas requisições?

Nesse sentido concebeu-se experimentos para responder a cada uma dessas perguntas. A fim de enriquecer a análise decidiu-se realizar os experimentos em dois cenários extremos de uso do HDFS:

- **Cenário Metadados** - Há um domínio de operações de metadados sobre operações de leitura e escrita no decorrer do tempo. Ou seja, há uma maior interação entre clientes e o NameNode e essa interação constitui a maior parte da carga do NameNode.
- **Cenário Entrada/Saída (E/S)** - Há uma predominância de operações de E/S sobre operações de metadados. Ou seja, a principal interação dos clientes se dará com os DataNodes para ler e escrever blocos. A carga do NameNode será composta basicamente do processamento das mensagens enviadas pelos DataNodes.

O ambiente escolhido para a realização dos experimentos foi a Amazon Elastic Computing Cloud (Amazon EC2) [3]. No tocante à análise, para que esta fosse possível assumiu-se a hipótese de que os relógios de todas as instâncias de máquinas virtuais utilizadas estão parcialmente sincronizadas. E que portanto quaisquer desvios entre os relógios podem ser negligenciados. Tal hipótese se demonstra plausível na medida em que as máquinas da Amazon EC2 atualizam seus relógios [51] por meio do Network Time Protocol (NTP) [72].

6.1 Ambiente experimental

A Amazon EC2 [3] é um serviço *Web* do tipo *IaaS-Infrastructure as a Service* (infraestrutura como serviço) que permite o aluguel de recursos computacionais em *data centers* da Amazon espalhados pelo mundo. Em termos práticos o usuário pode executar a quantidade desejada de instâncias de máquinas virtuais podendo definir vários parâmetros. Os parâmetros das máquinas virtuais podem ser que imagem de sistema operacional irão utilizar, que poder computacional terão, qual *data center* do mundo irá executar as instâncias, entre outros parâmetros. O consumo é cobrado por hora de cada instância utilizada.

O uso da Amazon EC2 criou a possibilidade de realizar experimentos com um número de nós mais próximo de um uso real do HDFS — instalações de produção normalmente variam de dezenas a milhares de nós [111]— a custos razoáveis. Dos tipos de instâncias possíveis optou-se pelo tipo *small*, por ser a instância de custo mais baixo que possui discos locais. Discos locais são vitais para que o desempenho de disco mantenha-se próxima do desempenho de instalações reais de HDFS. A Amazon define uma instância do tipo *small* como sendo [4]:

- 1,7 GB de memória RAM;
- 1 unidade computacional EC2 (1 núcleo virtual com uma unidade computacional EC2);
- 160 GB de armazenamento local;
- Plataforma de 32 ou 64 bits;
- Desempenho de E/S moderado.

Uma unidade computacional EC2 é definida como “*a capacidade de CPU equivalente de um processador 2007 Opteron ou 2007 Xeon de 1,0-1,2 GHz*”. Também por questões de custo decidimos por usar o *data center* da Virgínia, Estados Unidos, embora estivesse disponível o *data center* de São Paulo. Como a maioria do tráfego de dados se dá internamente à Amazon EC2 esta escolha não teve considerável impacto negativo. No tocante à imagem de sistema operacional usada pela instâncias, ela é uma imagem de Ubuntu 10.04.1 LTS 32 bits com núcleo Linux 2.6.32-311, Java 1.6.0_24-b07 e Apache ZooKeeper 3.3.3.

Entretanto um conjunto de instâncias de máquinas virtuais interligadas não se comportará exatamente como um *cluster* HDFS numa rede local composta de nó físicos dedicados. Embora a Amazon EC2 alegue ter por objetivo “*fornecer uma quantidade consistente de capacidade da CPU independentemente do hardware subjacente real*”, o estudo de Ou et

al. [78] aponta uma realidade diferente. O estudo mostra um crescimento da heterogeneidade de hardware para um mesmo tipo de instância entre 2011 e 2012. Verificou-se que entre instâncias do mesmo tipo e que usem o mesmo modelo de CPU o desempenho de processamento é similar. Porém entre instâncias com CPUs diferentes pode haver variações de até 60%. O desempenho de disco segue tendência semelhante porém com variação de até 25%. Já o estudo de Wang et al. [101] mostra que mesmo quando a rede está sob pouca carga, a vazão pode variar rapidamente entre 1 GB/s e zero e está de certa forma atrelada ao escalonamento da instância pela hipervisor. A forma encontrada para conter tais variações foi colher várias amostras.

6.2 Experimentos de desempenho

Nos experimentos de desempenho comparou-se o comportamento da solução Hot Standby Node com o HDFS 0.21 quando submetidos às mesmas cargas de trabalho. Não foi possível usar a versão 0.21 do HDFS sem ligeiras modificações. A razão para isso se deve ao mesmo defeito que impediu a solução desenvolvida de atuar como *checkpoint* *helper*, conforme foi explicado na Seção 5.4. A forma de contorná-lo foi a mesma que a usada para a solução: um único *checkpoint* no início das atividades com o fito de estabelecer a sincronização com o NameNode. De qualquer forma, o fato de ambos terem a mesma modificação torna a comparação mais justa. Outra mudança trazida do Hot Standby Node foi a mudança no sistema de coletas de métrica para incluir marcas temporais.

Deste modo, preparou-se uma imagem para ser usada por todas as instâncias de máquina virtual. A imagem continha uma instalação do HDFS 0.21 com as modificações já citadas, uma instalação do Hot Standby Node, ZooKeeper 3.3.3, os programas de geração de carga de trabalho e *scripts* para coleta de métricas. Um máximo de 43 instâncias foram usadas para os experimentos de desempenho:

- 20 instâncias para o papel de clientes executando o programa de geração de carga de trabalho;
- 20 instâncias para o papel de DataNodes;
- 1 instância para o papel de NameNode;
- 1 instância para o papel de Backup Node quando testando HDFS 0.21 ou de Hot Standby Node quando testando a solução;
- 1 instância extra para o papel de servidor ZooKeeper caso testando a solução. Usou-se um único servidor ao invés de um *ensemble* para simplificar a configuração.

O *timeout* da sessão do ZooKeeper foi definido para 2 minutos. A razão por trás de tal valor conservador será revelada quando discutimos *failover* na Seção 6.3. Para o tamanho de bloco e nível de replicação do HDFS foram utilizados os valores padrões, 64 MB e 3 respectivamente.

6.2.1 Programa de geração de carga de trabalho

Cada uma das instâncias que foram definidas para atuarem como clientes executa o programa de geração de carga de trabalho. O programa de geração de carga de trabalho tem por objetivos produzir a carga de trabalho demandada pelos experimentos e coletar medidas de vazão do ponto de vista do cliente. Trata-se de um programa configurável, no qual é possível definir o número de arquivos a serem criados bem como o número de blocos por arquivo. É por meio da manipulação desses dois parâmetros que são criados os cenários de Metadados e E/S. A Tabela 6.1 mostra os valores dos parâmetros para cada cenário.

Cenário	Arquivos	Blocos
Metadados	200	1
E/S	1	200

Tabela 6.1: Configurações do programa de geração de carga de trabalho para os cenários.

Assim, no cenário Metadados cada cliente cria 200 arquivos gerando um grande número de operações de metadados. Já no cenário E/S apenas um arquivo é criado por cliente. Consequentemente o foco das atividades do sistema fica nas operações de entrada e saída. Entretanto o número total de blocos é o mesmo para ambos os cenários: 4.000. Isso significa uma utilização de um quarto da capacidade total do *cluster* formado. Para cada arquivo o programa de geração de carga de trabalho realiza o seguinte ciclo:

1. O arquivo é escrito;
2. O conteúdo do diretório raiz é listado;
3. O conteúdo do arquivo recém-criado é lido.

O programa de geração de carga de trabalho registra em *log* próprio o início e fim de cada atividade para que seja possível o cômputo das vazões médias de leitura e escrita.

6.2.2 Procedimento experimental

A Tabela 6.2 lista em ordem os passos para os experimentos de desempenho. Um tique na coluna significa que o passo é execução na configuração correspondente (HDFS 0.21 ou a Hot Standby Node).

Passo	HDFS 0.21	Hot Standby
Iniciar manualmente a instância que roda o servidor ZooKeeper		✓
Iniciar manualmente a instância do Name-Node	✓	✓
Iniciar manualmente a instância do Backup Node	✓	
Iniciar manualmente a instância do Hot Standby Node		✓
Iniciar por meio de <i>script</i> as 20 instâncias dos DataNodes	✓	✓
Iniciar por meio de <i>script</i> as 20 instâncias dos Clientes	✓	✓

Tabela 6.2: Passos do experimento de desempenho.

A razão para iniciar manualmente o servidor ZooKeeper, o NameNode, o Backup Node e o Hot Standby Node é que os IPs desses nós devem ser conhecidos e passados para os demais nós. O experimento é considerado concluído quando todos os 4 mil blocos são escritos e lidos. Para cada permutação de configuração e cenário são obtidos 5 amostras. Portanto, para os experimentos de desempenho totalizaram-se 20 execuções: 2 configurações (HDFS 0.21 e Hot Standby Node) x 2 cenários (Metadados e E/S) x 5 amostras.

6.2.3 Coleta de dados

O sistema de coleta de métricas do HDFS já gerava muitos das medidas considerados interessantes para a análise de desempenho. Desta forma, ambas as instalações, HDFS 0.21 e o Hot Standby Node, foram configuradas para gerar suas saídas do sistema de coleta de métricas para arquivo a cada dez segundos. Graças à mudança para incluir marcas temporais nas saídas foi possível ter uma visão história do sistema, analisando a variação das métricas durante um certo intervalo de tempo.

Para complementar as métricas geradas pelo HDFS criou-se um *script* Python para monitorar o uso de CPU, uso da interface Ethernet, número de *threads*, bem como outros dados dos nós servidores. O *script* também coleta valores a cada dez segundos. Além disso o programa de geração de carga de trabalho registrou a vazão média para as leituras e escritas que executou.

6.2.4 Resultados e análise

A análise de desempenho visa responder a **Pergunta 1** do início deste capítulo. Para a análise resolvemos considerar apenas o intervalo de tempo em que o sistema esteve submetido à carga que foi definida para cada cenário experimental. Tal intervalo, ao qual chamaremos de *período de análise*, se inicia com o primeiro cliente executando o programa de geração de carga de trabalho (e portanto fazendo a primeira requisição ao NameNode) e termina com o último cliente encerrando suas atividades com o sistema de arquivo. As durações médias do período de análise para cada permutação podem ser vistas na Tabela 6.3. Os desvios após as médias se referem ao desvio padrão entre as amostras colhidas.

Cenário	Configuração	Intervalo
Metadados	HDFS 0.21	$(3.257 \pm 409)s$
	Hot Standby	$(3.774 \pm 568)s$
E/S	HDFS 0.21	$(3.586 \pm 253)s$
	Hot Standby	$(3.588 \pm 368)s$

Tabela 6.3: Duração média em segundos do período de análise.

À maneira que sistemas distribuídos são tradicionalmente avaliados, pautamos a análise de desempenho primordialmente nos parâmetros rede, processamento e memória. No gráficos que seguem as colunas descrevem os valores médios colhidos entre as amostras conforme será relatado em cada item. As barras de erro revelam os desvios padrões entre as médias das amostras. As colunas cinza-claro (■) se referem aos experimentos feitos com o HDFS 0.21. Ao passo que as colunas cinza-escuro (■) denotam resultados dos experimentos com a solução Hot Standby Node. Os gráficos sempre ocorrem aos pares, com o gráfico para o cenário Metadados à esquerda e o gráfico para o cenário E/S à direita.

Consumo de rede

Nesta subsubseção tentaremos identificar se houve aumento no tráfego de mensagens recebidas e enviadas pelos nós servidores devido à replicação de estado.

DataNodes O tráfego de dados de um DataNode é composto por mensagens de controle e por dados de arquivo durante operações de escrita e leitura. A fim de contabilizar por ambos os tráfegos analisamos os fluxos de dados na interface Ethernet dos DataNodes. Assumimos que o tráfego gerado por outros processos executando na mesma instância que o DataNode eram irrisórios. Em cada amostra coletamos a média de dados enviados e recebidos pela interface Ethernet das 20 instâncias que executavam o processo de

DataNode.

As colunas da Figura 6.1 foram obtidas da seguinte maneira:

1. Em cada amostra é conhecido o total de dados recebidos e enviados por cada DataNode durante o período de análise;
2. Obtém-se a média entre os valores dos DataNodes da amostra \Rightarrow média da amostra;
3. As colunas representam a média entre as médias das amostras.

Analisar os dados da Figura 6.1 é complicado devido ao fator tempo. Quanto maior o tempo maior o tráfego, já que periodicamente o DataNode enviará mensagens de heart-beat. Para remover a influência do tempo e permitir comparar melhor a versão original do HDFS 0.21 com o Hot Standby Node, os valores foram divididos pelos respectivos períodos de análise (considerando os desvios). Com isso produziu-se a Figura 6.2 como uma versão normalizada da Figura 6.1 em relação ao período de análise.

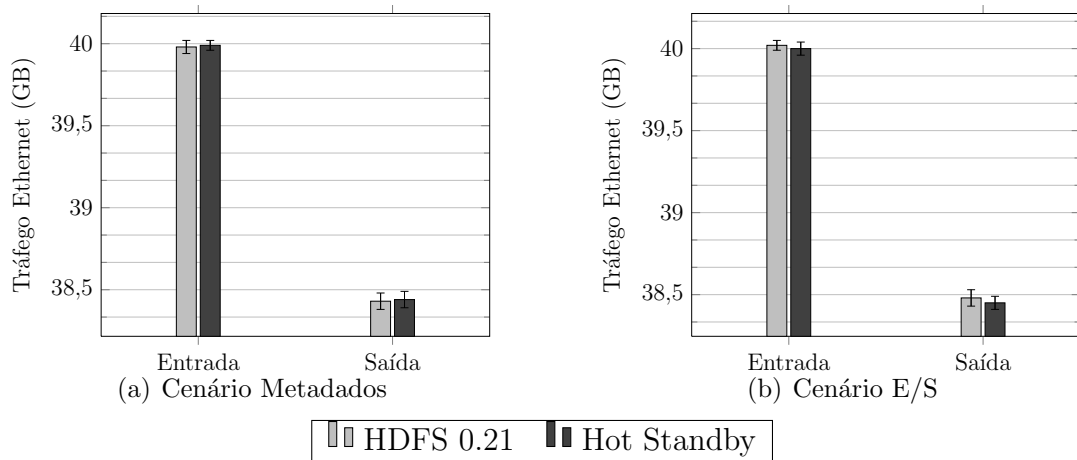


Figura 6.1: Tráfego médio na interface Ethernet dos DataNodes.

De maneira geral observa-se que o tráfego de entrada e saída são parecidos na Figura 6.1. Lembre-se que durante uma escrita os DataNodes são organizados em fila, com um DataNode repassando os dados para o outro, exceto pelo DataNode da cauda. Então há uma chance de $\frac{2}{3}$ de que um bloco recebido tenha de ser transferido para outro DataNode. Por outro lado cada uma das três réplicas é armazenada em um DataNode diferente. Assim há uma chance de $\frac{1}{3}$ de um réplica ser a escolhida para a leitura de um bloco. Portanto um DataNode irá receber e transmitir a mesma quantidade de blocos em média. A vantagem do tráfego de entrada sobre o de saída na Figura 6.1 se deve ao *TCP Segmentation Offload* [60]. Algumas placas de rede permitem que a divisão de um segmento TCP em segmentos de tamanho compatível com a MTU (*Maximum Transmission Unit* - Unidade Máxima de Transmissão) seja feita pela própria placa ao invés da

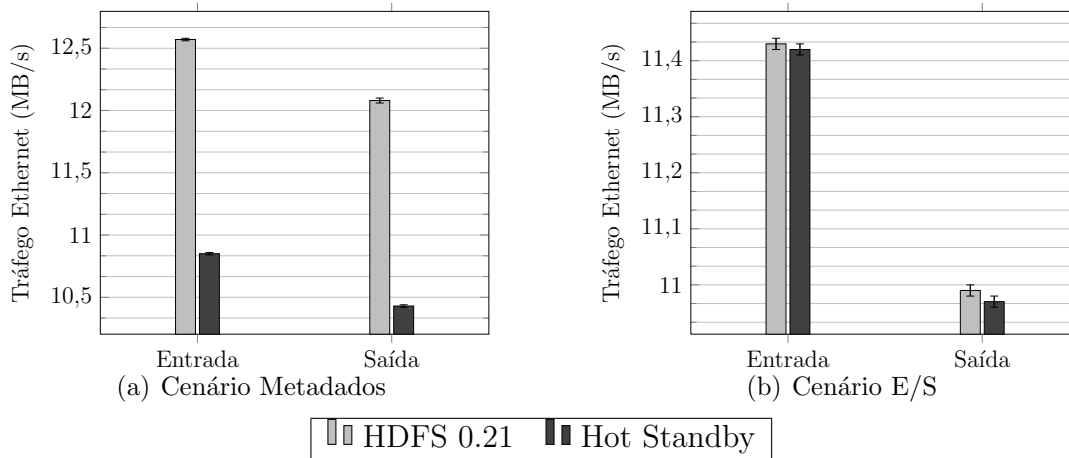


Figura 6.2: Tráfego médio na interface Ethernet dos DataNodes normalizado.

pilha TCP/IP do sistema operacional. Isso minimiza o uso de CPU e permite que maiores volumes de dados sejam enviados ao hardware de uma vez só, gerando ganhos de desempenho. Como a divisão de um segmento TCP em segmentos menores é feito na placa, o custo adicional gerado pelos cabeçalhos dos pacotes extras não pode ser contabilizado pelo sistema operacional. Assim é registrado um volume de dados enviados menor do que o realmente posto no cabos.

Em ambos os cenários os fluxos para o HDFS 0.21 e para a solução não são significativamente distintos na Figura 6.1. A expectativa seria um tráfego consideravelmente maior para a solução Hot Standby Node devido à duplicação das mensagens enviadas pelos DataNodes. Entretanto o volume gerado por essas mensagens extras é muito pequeno e acaba por se diluir no tráfego gerado pela leitura e escrita dos arquivos. Não temos dados sobre o tráfego de RPCs criado pelo DataNode, o qual contém essas mensagens extras. Todavia podemos fazer algumas extrapolações para se ter uma análise quantitativa. O maior volume de dados recebidos pelo Hot Standby Node via RPC registrado durante os experimentos foi de 122,12 MB. E a menor quantidade de dados enviados por um DataNode foi 7,39 GB. A razão entre esses valores é de 1,61%. Dado que o Hot Standby Node também recebe dados vindos do NameNode através de RPCs e de outros DataNodes, é seguro afirmar que no pior caso as mensagens extras são responsáveis por menos de 1,61% dos dados inseridos na rede por um DataNode.

Na Subseção 5.2.1 mencionamos reduções nas mensagens trocadas entre DataNodes e os outros nós servidores. Essas reduções se tornam perceptíveis na Figura 6.2. A redução do cenário E/S (Figura 6.2(b)) é menor devido às mensagens de block-received trocadas entre o DataNode e o Hot Standby. Essas mensagens são maiores por conterem os blocos não reconhecidos.

NameNode A comunicação do NameNode com os demais nós se dá quase que exclusivamente por RPCs. Uma notável exceção é a transferência de *checkpoints* entre NameNode e Backup/Hot Standby Node que usa HTTP. Porém como ela ocorre fora do período de análise, tal fluxo não será considerado. Portanto a análise limita-se ao tráfego RPC.

As colunas da Figura 6.3 foram obtidas da seguinte forma:

1. Para cada amostra sabe-se tráfego RPC enviado e recebido pelo NameNode durante o período de análise \Rightarrow valor da amostra;
2. As colunas referem-se à média dos valores das amostras.

A Figura 6.4 é uma versão normalizada da Figura 6.3 com relação do período de análise. Ou seja, os valores foram divididos pelos respectivos períodos de análise (considerando os desvios) de forma a remover a influência de sua duração sobre os resultados.

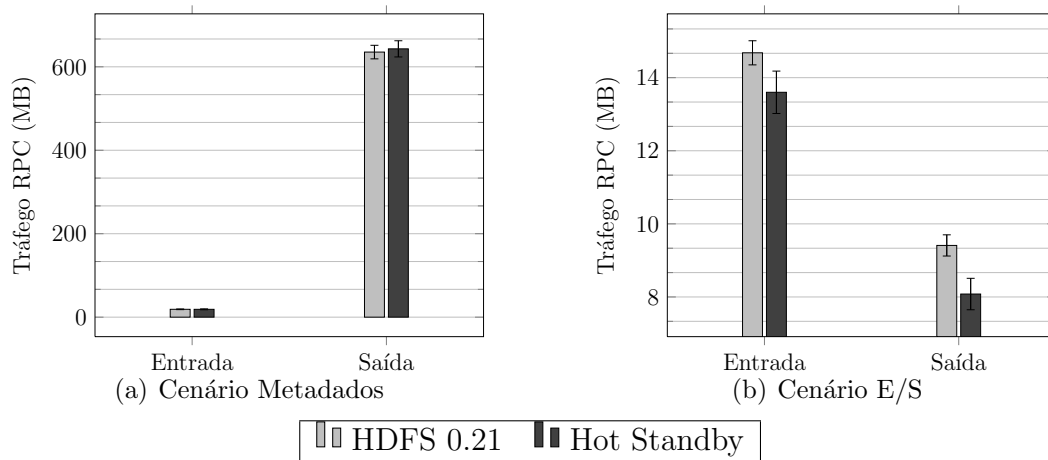


Figura 6.3: Tráfego RPC médio para o NameNode.

De maneira geral observa-se que no cenário Metadados (Figura 6.3(a)) o fluxo de saída é maior do que os outros fluxos. Isto se deve a um maior envio de respostas aos clientes e de entradas do *log* para o Backup/Hot Standby Node. Este último fluxo será refletido nas colunas de dados recebidos da Figura 6.5(a). Já no cenário E/S (Figura 6.3(b)), como são reduzidas as interações com clientes e Backup/Hot Standby Node, o NameNode passa a ter as mensagens vindas dos DataNodes como fluxo preponderante.

O tráfego RPC do NameNode é composto de uma parte fixa e uma parte variável. A parte fixa corresponde ao tratamento das requisições dos clientes, à transmissão das entradas do *log* para o Backup/Hot Standby Node, ao registro dos DataNodes e ao tratamento de mensagens de block-received. Essa parte é fixa no sentido que apresentará variações desprezíveis se a carga aplicada aos nós for a mesma. Já na parte variável se encontram as mensagens de block-report e heartbeat que por serem mensagens periódicas tem seu peso

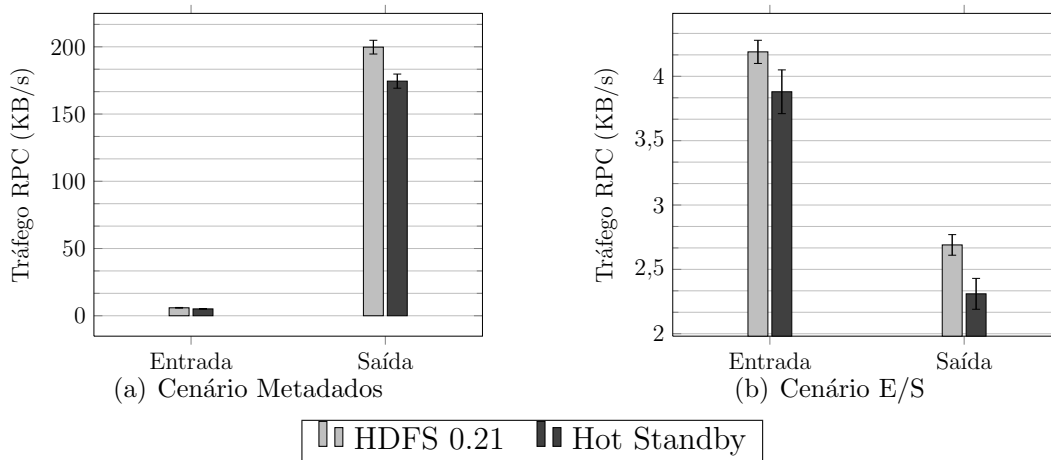


Figura 6.4: Tráfego RPC médio para o NameNode normalizado.

no tráfego determinado pela duração do experimento. Dado o número de 20 DataNodes, o NameNode recebe mensagens de heartbeat à uma taxa constante de 6,6 mensagens por segundo. Os block-reports são mais raros devido a sua menor periodicidade, sendo o recorde de 7 mensagens desse tipo durante o período de análise.

Essa caracterização do tráfego RPC do NameNode gera expectativas sobre os resultados que deveriam ser obtidos. Entretanto essas expectativas são contrariadas pelas reduções nos tamanhos das mensagens trocadas com os DataNodes conforme foi apresentado na Subseção 5.2.1 e é evidenciado pela Figura 6.4. No cenário Metadados (Figura 6.3(a)), o período de análise para a solução Hot Standby Node é maior que para o HDFS 0.21. Desta forma a expectativa era que a parte variável do tráfego RPC do NameNode contribuísse com um aumento considerável. No entanto os valores para ambas as configurações são os mesmos. No cenário E/S (Figura 6.3(b)), os períodos de análise são quase os mesmos. Logo a expectativa era por fluxos de dados parecidos. Mas a redução tanto na requisição como na resposta para mensagens de block-received promoveu valores menores para a solução.

Backup Node e Hot Standby Node As razões para se analisar apenas o tráfego RPC do Backup/Hot Standby Node e a maneira pelas quais as colunas da Figura 6.5 foram geradas são os mesmos que no caso do NameNode. Não convém portanto repeti-las aqui. De maneira similar, a Figura 6.6 é uma versão normalizada da Figura 6.5 com relação ao período de análise.

Em todos os casos o tráfego de RPC do Hot Standby Node é maior que do Backup Node (embora nem sempre significativamente). É um resultado óbvio pois, adicionalmente em relação ao Backup Node, o Hot Standby Node tem de processar o *log* do sistema de

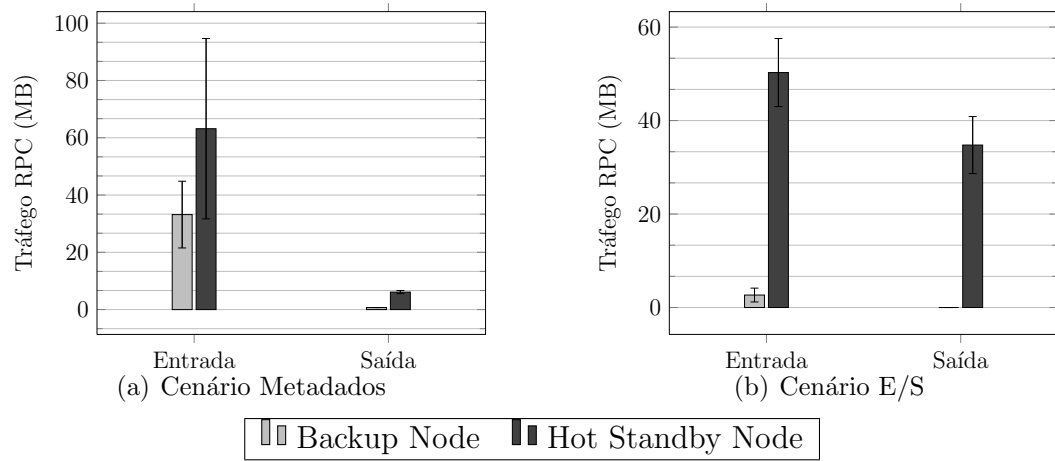


Figura 6.5: Tráfego RPC médio para o Backup Node e o Hot Standby Node.

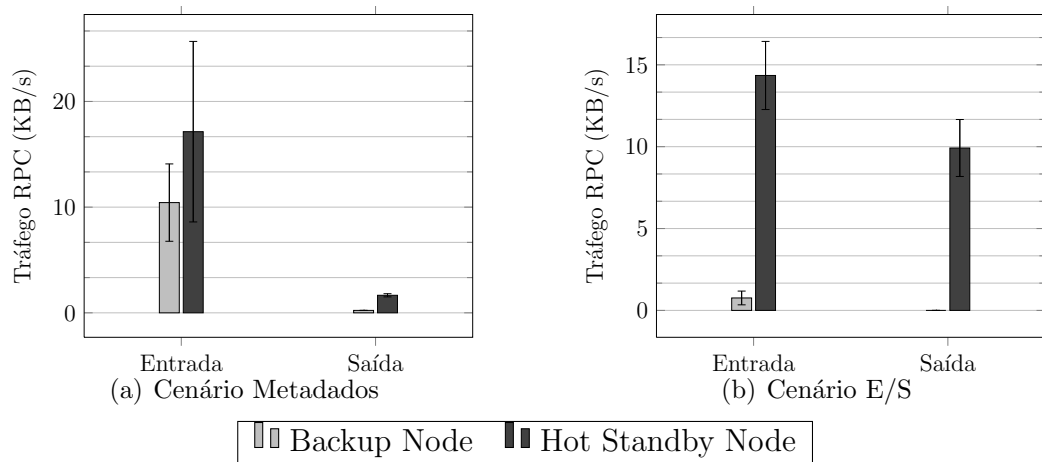


Figura 6.6: Tráfego RPC médio para o Backup Node e o Hot Standby Node normalizado.

arquivo e interagir com os DataNodes.

No cenário Metadados (Figura 6.5(a)), o volume de dados recebidos pelo Backup Node e pelo Hot Standby Node é maior que em outras situações devido principalmente ao envio das entradas do *log* pelo NameNode.

No cenário E/S (Figura 6.5(b)) o volume de dados enviados pelo Hot Standby Node é maior que no cenário Metadados mesmo embora o nó tenha de trocar mensagens com os DataNodes pelo mesmo número de blocos. No mesmo cenário a quantidade de dados recebidos pelo Hot Standby Node alcança níveis próximos ao do cenário Metadados, muito embora o tráfego vindo do NameNode seja menor nesta situação. A razão para tal conduta reside nas mensagens de *block-received*. Lembre-se que o Hot Standby Node devolve para o DataNode na resposta de uma mensagem de *block-received* os blocos que estavam presente na mensagem mas que não foram reconhecidos pelo Hot Standby Node. Desta forma, o DataNode pode retentar enviar esses blocos não-reconhecidos na próxima mensagem de *block-received*. No cenário Metadados os arquivos são compostos por um bloco apenas. Portanto quando a mensagem de *block-received* chega ao Hot Standby Node é muito provável que o arquivo já esteja fechado. Consequentemente o Hot Standby Node reconhece quase todos os blocos da mensagem de *block-received* como pertencentes a um arquivo em sua primeira tentativa. Desta forma as mensagens de *block-received* são mantidas curtas, com tamanho médio de 465 bytes. Também são mantidas curtas as respostas do Hot Standby Node para estas mensagens: tamanho médio de 165 bytes.

Todavia o mesmo não ocorre para o cenário E/S. Neste cenário os arquivos são compostos por 200 blocos. Isto significa que apenas após a escrita do ducentésimo bloco é que algum dos blocos componentes do arquivo serão reconhecidos pelo Hot Standby Node, pois é nesse momento que se dará o fechamento do arquivo e o nó tomará ciência de quais blocos o compõem. Assim, conforme os arquivos vão sendo escritos porém não são fechados, as mensagens de *block-received* bem como a resposta à elas manter-se-ão crescendo em tamanho. De fato, a mensagem nesse caso possui tamanho médio de 7.830 bytes e resposta de 5.703 bytes. Ou seja um fator de aumento de 16,83 e 34,56 vezes respectivamente, ocasionado as grandes colunas vistas na Figura 6.5(b) para o Hot Standby Node.

ZooKeeper Durante o período de análise toda comunicação entre nós e ZooKeeper se limitará única e exclusivamente à manutenção da sessão. Para tanto, um cliente ZooKeeper envia uma mensagem de “*ping*” para o ZooKeeper a cada $\frac{1}{3}$ do *timeout* definido. No caso dos experimentos isso significa uma mensagem de *ping* a cada 40 segundos. De acordo com *traces* TCP obtidos durante versões reduzidas do experimento, as mensagens de *ping* carregam 144 bytes e a resposta do ZooKeeper a essas mensagens carrega outros 86 bytes.

No protótipo os únicos nós servidores que executam um cliente ZooKeeper são o NameNode (NN) e o Hot Standby Node (HN), visto que executam o Failover Manager. A Tabela 6.4 compara os tráfegos médios gerados por RPC e pelo ZooKeeper nesses nós. A terceira linha da tabela mostra quanto o tráfego gerado pelo ZooKeeper representa em termos percentuais do tráfego gerado por RPCs para o NameNode e o Hot Standby Node.

	Metadados				E/S			
	Entrada		Saída		Entrada		Saída	
	NN	HN	NN	HN	NN	HN	NN	HN
Tráfego RPC (KB)	19.237	64.675	658.507	6.262	13.925	51.478	8276	35.594
Tráfego ZooKeeper (KB)	7,89		13,22		7,47		12,52	
Comparação. (%)	0,041	0,012	0,002	0,211	0,054	0,015	0,151	0,035

Tabela 6.4: Comparação entre o tráfego médio gerando por RPC e pelo ZooKeeper.

Como pode ser visto na Tabela 6.4 o fluxo de dados inserido por cada nó que interage com o servidor ZooKeeper é pequeno, não atingindo 1%. Mesmo considerando todos os 22 nós (20 clientes e 2 servidores) o tráfego total na rede adicionado pelo ZooKeeper é de menos de 300 KB.

Processamento

Nesta subseção avaliaremos o processamento extra imposto pela solução Hot Standby Node aos nós servidores. Nesse contexto entende-se por consumo de CPU como sendo a soma dos tempos de usuário e de núcleo gastos pelo processo servidor durante o período de análise. Esse valor é medido em segundos e obtidos a partir do *proc file system* [83] do Linux, o qual fornece várias informações sobre o sistema e processos. Os tempos de usuário e núcleo podem ser obtidos do pseudo-arquivo `/proc/<pid>/stat`, no qual `pid` se refere ao identificador do processo servidor em questão.

DataNodes O consumo de CPU pelos DataNodes é basicamente composto pela transferência de dados para clientes e o envio de mensagens de estado para o NameNode. O trabalho de transferência de dados para clientes é o mesmo para os *clusters* das duas configurações (HDFS 0.21 e Hot Standby Node).

As colunas da Figura 6.7 foram obtidas da seguinte forma:

1. Em cada amostra o consumo de CPU durante o período de análise é conhecido para cada DataNode;
2. Obtém-se a média entre os consumos de CPU dos DataNodes da amostra \Rightarrow média da amostra;
3. As colunas refletem a média entre as médias das amostras.

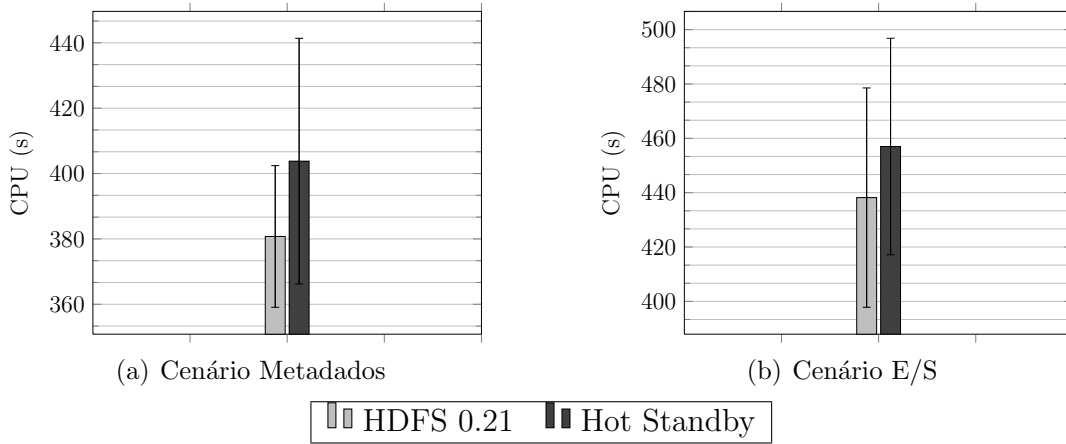


Figura 6.7: Consumo médio de CPU pelos DataNodes.

Observa-se que para ambos os cenários o fato de termos uma *thread* adicional para enviar as mensagens para o Hot Standby Node causou um aumento no consumo médio de CPU, mas que considerando os desvios não é significativo.

NameNode As colunas da Figura 6.8 foram obtidas da seguinte forma:

1. Para cada amostra sabe-se o consumo de CPU pelo NameNode durante o período de análise \Rightarrow valor da amostra;
2. As colunas se referem à média dos valores das amostras.

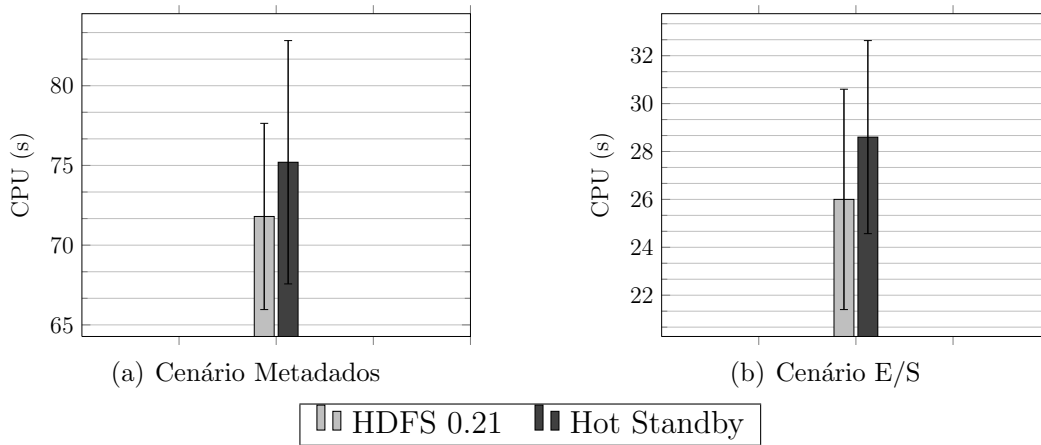


Figura 6.8: Consumo médio de CPU pelo NameNode.

A Figura 6.8(a) em comparação com a Figura 6.8(b) revela um consumo maior de CPU no cenário Metadados do que no cenário E/S. Isso significa que os cenários funcionaram

como projetados. O cenário de Metadados deve focar-se na manipulação de metadados. Em consequência ele exige mais do NameNode. Na solução Hot Standby Node o processo do NameNode também abriga o Failover Manager, o que causa um aumento no consumo médio de CPU em comparação com o HDFS 0.21, mas que considerando os desvios não chega a ser significativo.

Backup Node e Hot Standby Node As colunas da Figura 6.9 foram obtidas da seguinte forma:

1. Para cada amostra sabe-se o consumo de CPU pelo Backup Node ou Hot Standby Node durante o período de análise \Rightarrow valor da amostra;
2. As colunas referem-se à média dos valores das amostras.

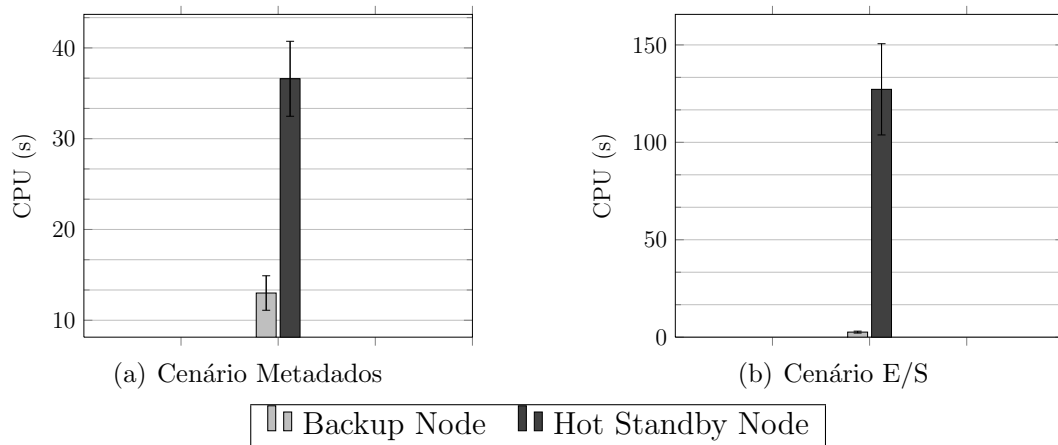


Figura 6.9: Consumo médio de CPU pelo Backup Node e Hot Standby Node.

É previsto que o Backup Node gaste mais CPU no cenário Metadados (Figura 6.9(a)) do que no cenário E/S (Figura 6.9(b)). A razão para isso reside no fato de o segundo cenário ter menos operações sobre metadados, o que resulta em menos entradas de *log* para processar. Também é previsto que o consumo de CPU do Hot Standby Node seja maior que do Backup Node nos mesmos cenários, visto que além de realizar as atividades do Backup Node ele tem de receber e processar as mensagens dos DataNodes e hospedar uma instância do Failover Manager. Mas um fato que chama a atenção é de que na Figura 6.8(b) o consumo de CPU do Hot Standby Node é quase 4,5 vezes consumo do NameNode no mesmo cenário e configuração (Figura 6.8(b)) para desempenhar atividades semelhantes. Isso também é decorrente do problema com as mensagens de *block-received*. Deixaremos a discussão desse fato para a subsubseção seguinte a fim de correlacioná-la com o consumo de memória, de forma que ela seja então mais completa.

Memória

Todos os processos envolvidos nos experimentos são programas Java. Uma máquina virtual Java é responsável por alocar e gerenciar a memória utilizada pelo programa que ela executa. O sistema de memória da máquina virtual Java gerencia os seguintes tipos de memória [65]:

Heap É o espaço de memória que em tempo de execução é usado para alocar todas as instâncias de objetos e arranjos. É também o espaço de memória sujeito à coleta de lixo.

Non-heap É o espaço de memória compartilhado entre todas as *threads* e armazena dados de classes como informações sobre campos e métodos, código para métodos e construtores e *pools* como *strings* internalizadas. Também pode ser usado para processamento interno e otimização da máquina virtual.

Dado que Java adota essa divisão, nós também o faremos nessa análise. A quantidade de memória usada pelos programas varia no tempo. O sistema de coleta de métricas do HDFS registra o consumo de memória a cada dez segundos. Como resumo do que ocorreu durante o período de análise usaremos a média desse valores.

DataNodes As colunas da Figura 6.10 e da Figura 6.11 foram obtidas da seguinte forma:

1. O DataNode registra por meio do sistema de coleta métrica o seu uso de memória *heap* e *non-heap* a cada dez segundos;
2. Para cada DataNode calculamos a média entre as métricas colhidas durante o período de análise \Rightarrow uso médio de memória do DataNode;
3. Para cada amostra calculamos a média entre os usos médios de memória dos DataNodes \Rightarrow média da amostra;
4. As colunas refletem a média entre as médias das amostras.

Embora a escala dos gráficos da Figura 6.11 exagerem a diferença, há uma leve aumento na memória *non-heap* devido ao código que foi adicionado ao DataNode. Na Figura 6.10(a) vê-se que o consumo de memória *heap* mantém-se o mesmo apesar da *thread* adicionada pela solução. Isso ocorre porque no cenário Metadados, embora as mensagens enviadas pelos DataNodes são duplicadas, elas são muito pequenas não ocupando muito espaço extra. Em contraste no cenário E/S (Figura 6.10(b)), o tamanho das mensagens de block-received promove um aumento no uso de memória *heap*.

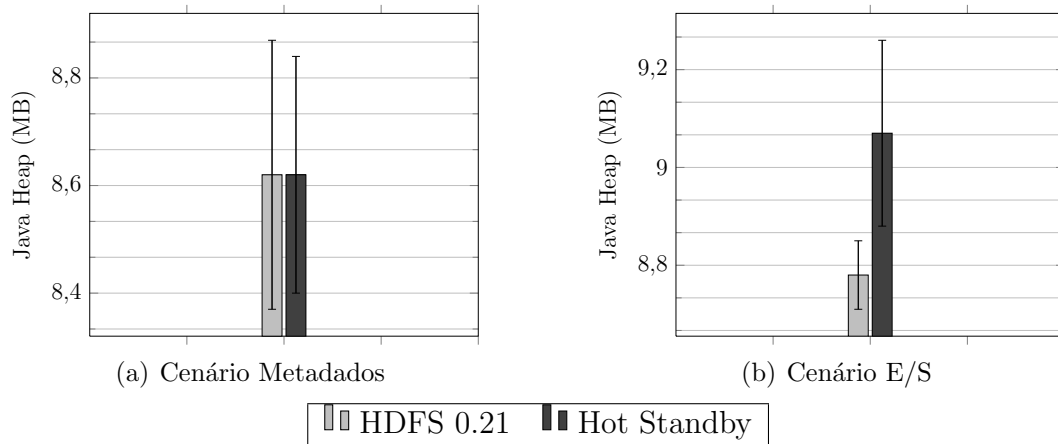


Figura 6.10: Uso médio de memória *heap* pelos DataNodes.

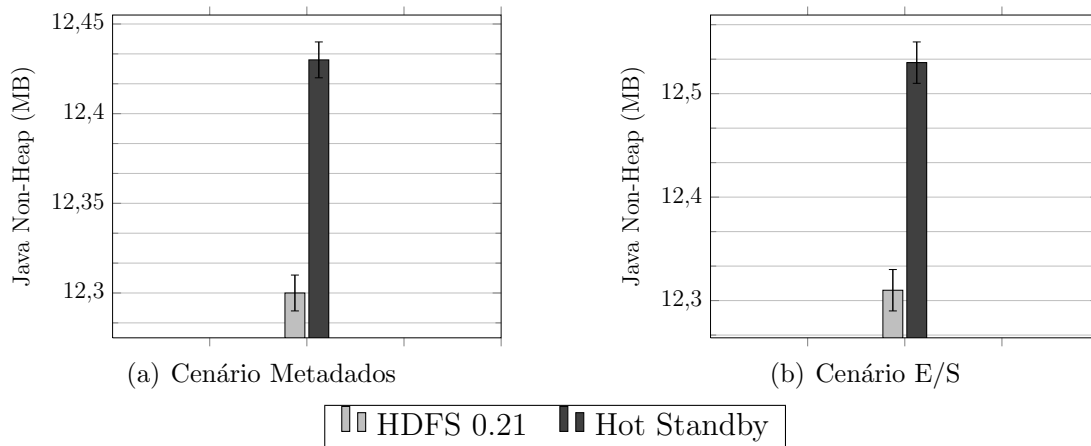


Figura 6.11: Uso médio de memória *non-heap* pelos DataNodes.

NameNode As colunas da Figura 6.12 e da Figura 6.13 foram obtidas da seguinte forma:

1. O NameNode registra por meio do sistema de coleta métrica o seu uso de memória *heap* e *non-heap* a cada dez segundos;
2. Em cada amostra calculamos a média entre as métricas colhidas para o NameNode durante o período de análise \Rightarrow média da amostra;
3. As colunas refletem a média entre as médias das amostras.

Em todos os casos há aumento no uso de memória entre o NameNode do HDFS 0.21 e o da solução Hot Standby Node, aumento esse decorrente da execução do Failover Manager.

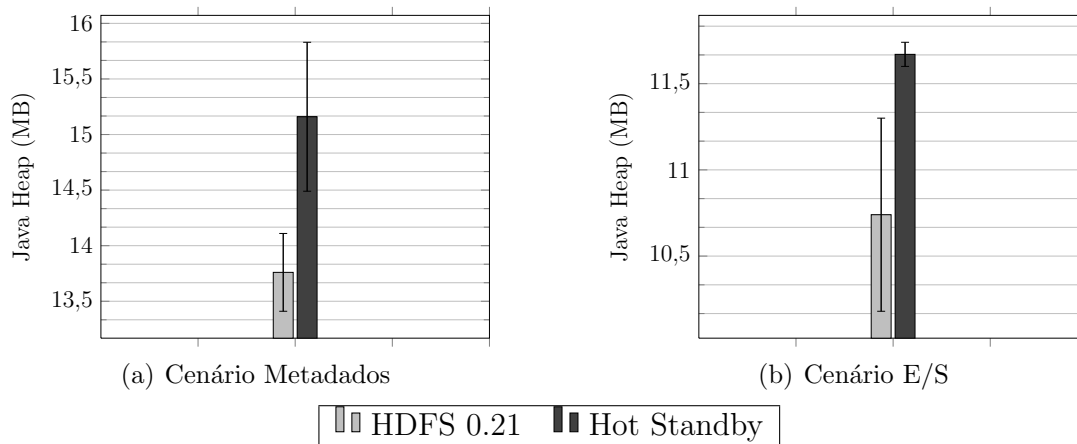


Figura 6.12: Uso médio de memória *heap* pelo NameNode.

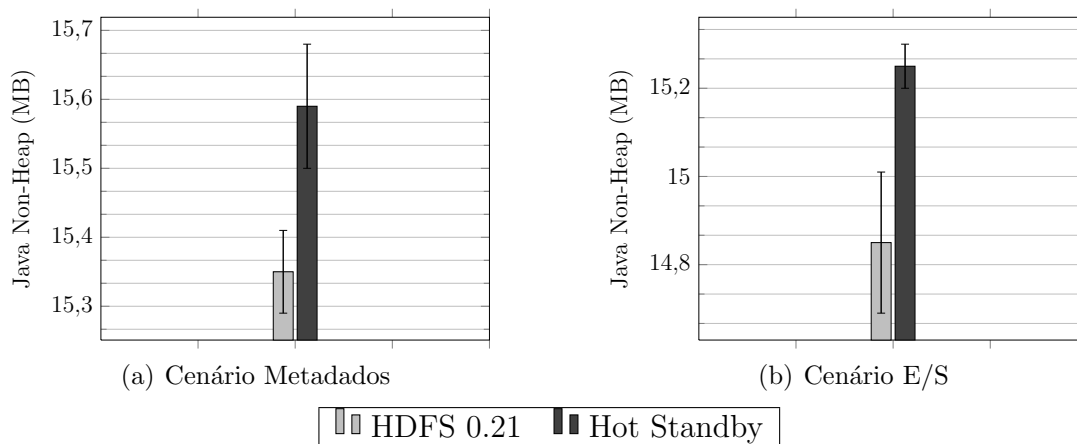


Figura 6.13: Uso médio de memória *non-heap* pelo NameNode.

Backup Node e Hot Standby Node Nesse parágrafo continuamos a análise que havia sido pausada quando discutimos o consumo de CPU do Hot Standby Node. Os valores para os gráficos da Figura 6.14 e da Figura 6.15 foram colhidos de maneira semelhante ao que foi feito para o NameNode.

Assim como para CPU há também aumento do uso de memória em comparação com o Backup Node devido ao Failover Manager e ao processamento das mensagens advindas dos DataNodes. Também aqui o cenário E/S se revela como sendo o pior para o Hot Standby Node. Seu consumo de memória *heap* é pouco mais que 1,5 vezes a do NameNode para a mesma situação. Recapitulando o que ocorre no cenário E/S:

- Os arquivos possuem 200 blocos. Portanto um arquivo somente será fechado após seu ducentésimo bloco ter sido escrito. É somente após o fechamento que o Hot Standby Node, por intermédio do *log*, tornar-se-á ciente de quais blocos compõem

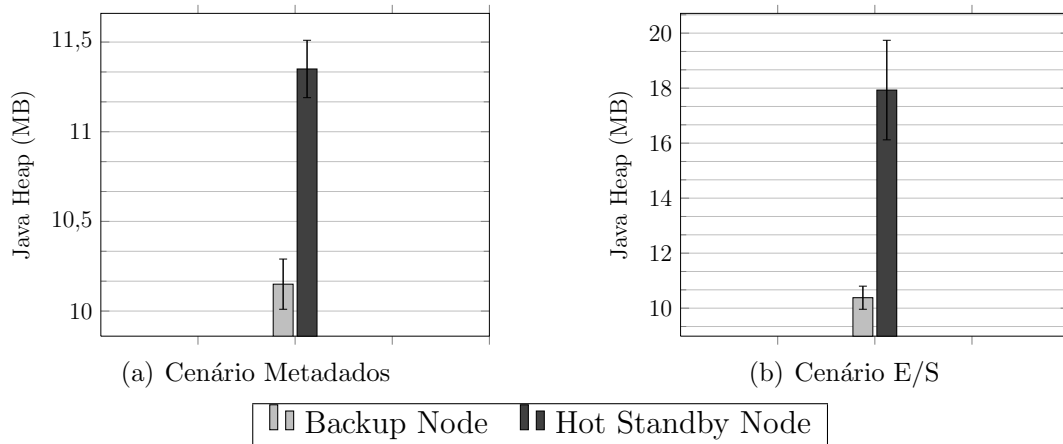


Figura 6.14: Uso médio de memória *heap* pelo Backup Node e Hot Standby Node.

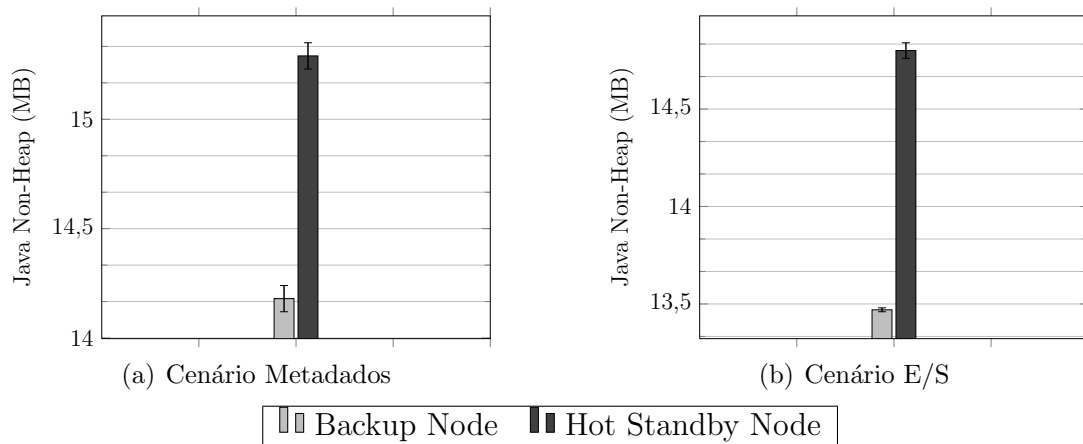


Figura 6.15: Uso médio de memória *non-heap* pelo Backup Node e Hot Standby Node.

o arquivo. E deste modo poderá então reconhecer seus blocos em mensagens de *block-received*.

- Os DataNodes enviam mensagens de *block-received* tanto para o NameNode como para o Hot Standby Node conforme os blocos dos arquivos são escritos.
- O Hot Standby Node ao receber uma mensagem de *block-received* reconhecerá poucos blocos. Por conseguinte, sua resposta à mensagem incluirá muitos dos blocos recebidos.
- O DataNode enfileirá os blocos que estão na reposta do Hot Standby para serem retentados na próxima mensagem. Como ainda os arquivos não foram fechados, os blocos novamente não serão reconhecidos. Assim ele passarão a acumular e as

mensagens de block-received e suas respectivas repostas irão gradualmente crescendo de tamanho.

Quando essas mensagens grandes chegam ao Hot Standby Node o nó tem de desempacotá-las, criando inúmeros objetos para representar os blocos, o que consome muita memória *heap*. Todos esses blocos têm de ser processados. Como são muitos o processamento é custoso. Ao final do processamento poucos objetos representando os blocos serão aproveitados, pois poucos serão os blocos reconhecidos. Ou seja, teremos vários objetos sendo criados e descartados muito rapidamente. Isto forçará mais execuções do coletor de lixo, conforme pode ser visto na Tabela 6.5 que lista o número médio de execuções do coletor de lixo para cada permutação.

Nó	Cenário	
	Metadados	E/S
Backup Node	57,60	4,80
Hot Standby Node	281,80	2.613,00

Tabela 6.5: Número médio de execuções do coletor de lixo.

O coletor de lixo do Hot Standby node trabalha nove vezes mais no cenário E/S que no Metadados. Esse grande número de execuções do coletor de lixo contribui ainda mais para o consumo de CPU do Hot Standby Node.

Vazão

Até então analisamos o impacto da solução Hot Standby Node sobre a rede, sobre o processamento e sobre a memória dos nós servidores. Esse custo adicional poderá apenas ser justificado se o efeito final sobre os clientes for mínimo. Nesse sentido consideramos a vazão de dados como a métrica primordial para tal avaliação, pois ela é uma medida da quantidade de trabalho que o sistema de arquivos pode fazer em favor dos clientes.

Cada cliente mede a vazão de leitura ou escrita dividindo o tamanho do arquivo pelo tempo gasto para lê-lo ou escrevê-lo. A Figura 6.16 mostra a vazão média percebida pelos clientes para todas as permutações. Suas colunas foram obtidas da seguinte forma:

1. O cliente registra a vazão para cada escrita e leitura que faz;
2. Para cada amostra calculamos a média entre as vazões medidas pelos clientes \Rightarrow média da amostra;
3. As colunas refletem a média entre as médias das amostras.

De maneira geral vê-se que a vazão de escrita é menor do que a de leitura, consequência da replicação de blocos. A boa notícia é que, apesar do aumento de consumo de recursos

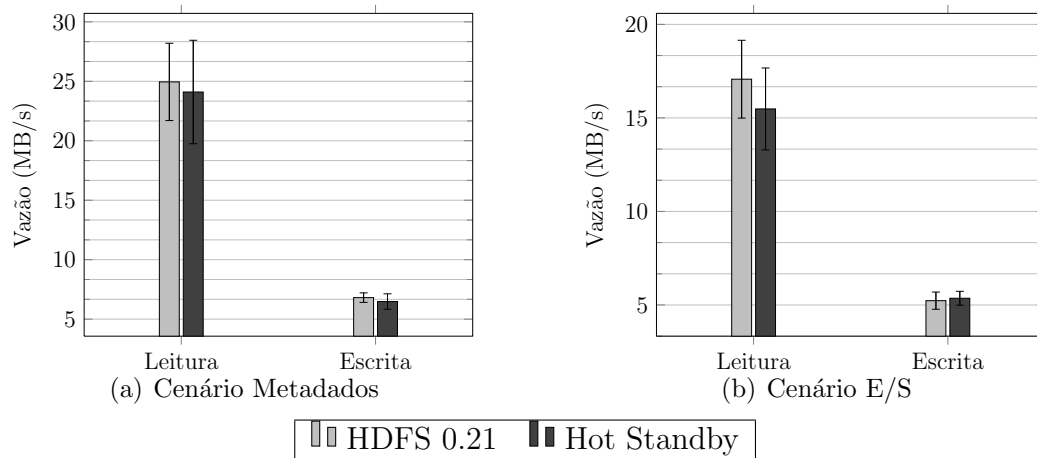


Figura 6.16: Vazão média de leitura e escrita para clientes.

com a solução Hot Standby Node, a vazão manteve-se a mesma em comparação com o sistema original considerando os desvios padrões. A vazão média atingida pela solução Hot Standby Node não foi menor que 90% da vazão obtida com o HDFS 0.21. No caso de escrita essa diferença tende a ser menor em razão da amortização causada pela replicação.

6.3 Experimentos de *failover*

Nos experimentos de *failover* são observados os tempos de reação do Hot Standby e do programa de geração de carga de trabalho frente à falha do NameNode. Utilizou-se a mesma imagem, os mesmos cenários com os mesmos números de arquivos e blocos a serem criados, o mesmo número de instâncias que os utilizados para os experimentos de desempenho, e o mesmos valores para o tamanho de bloco e nível de replicação. As únicas diferenças com relação aos experimentos de desempenho são o programa de geração de carga de trabalho e o fato do experimento ser interrompido no meio em comparação com os experimentos de desempenho. Obviamente os experimentos foram executados apenas na solução desenvolvida. Também aqui adotou-se o valor de 2 minutos para o *timeout* da sessão do ZooKeeper. O racional para esta decisão será abordado quando discutirmos os resultados.

6.3.1 Programa de geração de carga de trabalho

O programa de geração de carga de trabalho usado nos experimentos de *failover* também permite definir o número de arquivos, assim como o número de blocos por arquivo. Porém, ao contrário do programa usado para os experimentos de desempenho, este aplicativo

apenas cria e escreve os arquivos.

Para o protótipo da solução Hot Standby Node não foram feitas alterações no cliente do HDFS. Portanto é responsabilidade das aplicações implementarem o tratamento do *failover*. Consequentemente, o programa de geração de carga de trabalho se conecta ao servidor ZooKeeper e se registra para ser notificado sobre mudanças de servidor de metadados. O programa interrompe a escrita atual e redireciona suas requisições para este servidor ao receber a notícia de que o Hot Standby Node é o novo servidor de metadados.

Todos os eventos tais como início e fim da escrita, falhas e mudança do servidor de metadados são registrados em *log* próprio do aplicativo, sendo sempre precedidos de marcas temporais para a determinação de intervalos de tempo entre os eventos.

6.3.2 Procedimento experimental

O passos para os experimentos de *failover* são listados a seguir:

1. Iniciar manualmente a instância que roda o servidor ZooKeeper;
2. Iniciar manualmente a instância do NameNode;
3. Iniciar manualmente a instância do Hot Standby Node;
4. Iniciar com *script* as 20 instâncias dos DataNodes;
5. Iniciar manualmente uma instância que atuará como cliente de controle e executará o programa de geração de carga de trabalho;
6. Iniciar com *script* as 19 instâncias restantes dos clientes;
7. Quando o sistema de arquivo possuir mais 2.000 blocos matar o processo do NameNode;
8. Esperar que todo o processo de *failover* tenha terminado e que tenham sido escritos pelo menos mais 100 blocos.

Aqui, diferentemente dos experimentos de desempenho, usaremos apenas um cliente para análise dos dados ao invés de todos os vinte que participaram dos experimentos. A razão para isso é que o cliente de controle foi iniciado manualmente para que fosse monitorado mais de perto, permitindo verificar se todo o processo de *failover* transcorria de forma correta. Para cada um dos dois cenários, Metadados e E/S, são colhidas 5 amostras totalizando 10 execuções.

6.3.3 Coleta de dados

Os dados dos experimentos são obtidos a partir dos *logs* de execução de cada nó. Cada entrada de *log* é precedida de uma marca temporal, sendo assim possível determinar quando ela foi criada. Todos os eventos para a análise do experimento são registrados nos *logs* de execução dos nós. Tais eventos serão enumerados quando da apresentação dos resultados e sua análise.

6.3.4 Resultados e análise

Nesta subseção responderemos a **Pergunta 2** do início desse capítulo, ou seja, em quanto tempo o Hot Standby Node estará pronto para atender os clientes após a falha do NameNode. Para a análise consideraremos os seguintes eventos:

- I NameNode comunica-se com o ZooKeeper pela última vez;
- II NameNode falha;
- III Comunicação do cliente de controle com o NameNode falha;
- IV Sessão do NameNode com o ZooKeeper expira;
- V Hot Standby Node inicia a transição;
- VI Hot Standby Node termina a transição;
- VII Cliente de controle é notificado sobre a mudança de servidor de metadados;
- VIII Cliente de controle reinicia sua requisição de escrita;
- IX Hot Standby Node processa sua primeira requisição.

Com base nos eventos citados definimos os seguinte intervalos de tempo explicitados na Tabela 6.6 e mostrados de forma gráfica na Figura 6.17. Os tempos médios para esses intervalos, juntamente com o desvio padrão das amostras é exibido na Tabela 6.7.

O intervalo **C** é o único intervalo fixo. Trata-se do *timeout* para as sessões do ZooKeeper. Adotou-se um valor conservador de 2 minutos para evitar falsa detecção de falha do NameNode. Esse valor foi adotado com base nas recomendações do Hadoop para Amazon EC2 [82] e nas observações feitas quando se discutiu como decidir uma valor de timeout na Subsubseção 5.2.2. Como dito anteriormente um cliente ZooKeeper envia

Intervalo	Nome	Evento Inicial	Evento Final
A	Tempo Total de <i>Failover</i>	II	IX
B	Tempo de Detecção de Falha	II	IV
C	<i>Timeout</i> do ZooKeeper	I	IV
D	Tempo de Reação do Hot Standby	IV	V
E	Tempo de Percepção de falha pelo cliente	II	III
F	Tempo de Notificação do cliente	VI	VII
G	Tempo de Reação do cliente	VII	VIII
H	Tempo de Transição do Hot Standby	V	VI
I	Tempo para Primeira Requisição	VI	IX

Tabela 6.6: Intervalos para a análise do tempo de *failover*.

Intervalo	Metadados	E/S
A	(1,62 ± 0,20)min	(2,31 ± 0,40)min
B	(1,61 ± 0,20)min	(1,81 ± 0,15)min
C	2min	2min
D	(18,60 ± 59,20)ms	(8,60 ± 24,79)ms
E	(15,81 ± 10,07)s	(8,84 ± 3,53)s
F	(18,20 ± 118,10)ms	(390,40 ± 721,08)ms
G	(0,40 ± 0,49)ms	(0,80 ± 0,75)ms
H	(21,80 ± 7,14)ms	(3.173,20 ± 2.654,81)ms
I	(207,20 ± 75,29)ms	(27.311,20 ± 26.452,58)ms

Tabela 6.7: Duração dos intervalos considerados para a análise do tempo de *failover*.

pings a cada $\frac{1}{3}$ do *timeout* para manter a sessão. Estatisticamente uma falha do NameNode ocorrerá entre dois *pings*. Logo o tempo de detecção (intervalo **B**) será menor que o *timeout* do ZooKeeper.

O tempo de reação do Hot Standby Node (intervalo **D**), ou seja o tempo entre o ZooKeeper declarar a sessão do NameNode como expirada, notificar o Hot Standby Node e iniciar o processo de transição; é pequeno: alguns milissegundos. Isso mostra que o Hot Standby Node pode prontamente iniciar o processo de transição.

A partir do início da transição do Hot Standby Node temos tempos bem distintos para o cenário Metadados e o E/S. No cenário Metadados quando o NameNode é terminado, tem-se no máximo 20 arquivos abertos e 20 blocos não reconhecidos. O trabalho de processamento de blocos é pequeno para poder impactar o trabalho de transição. Portanto o tempo de transição do Hot Standby Node (intervalo **H**) é pequeno, já que o nó pode ocupar-se apenas da transição. E se compararmos o tempo que o ZooKeeper levou para notificar os clientes (intervalo **F**) e o tempo para o processamento da primeira requisição

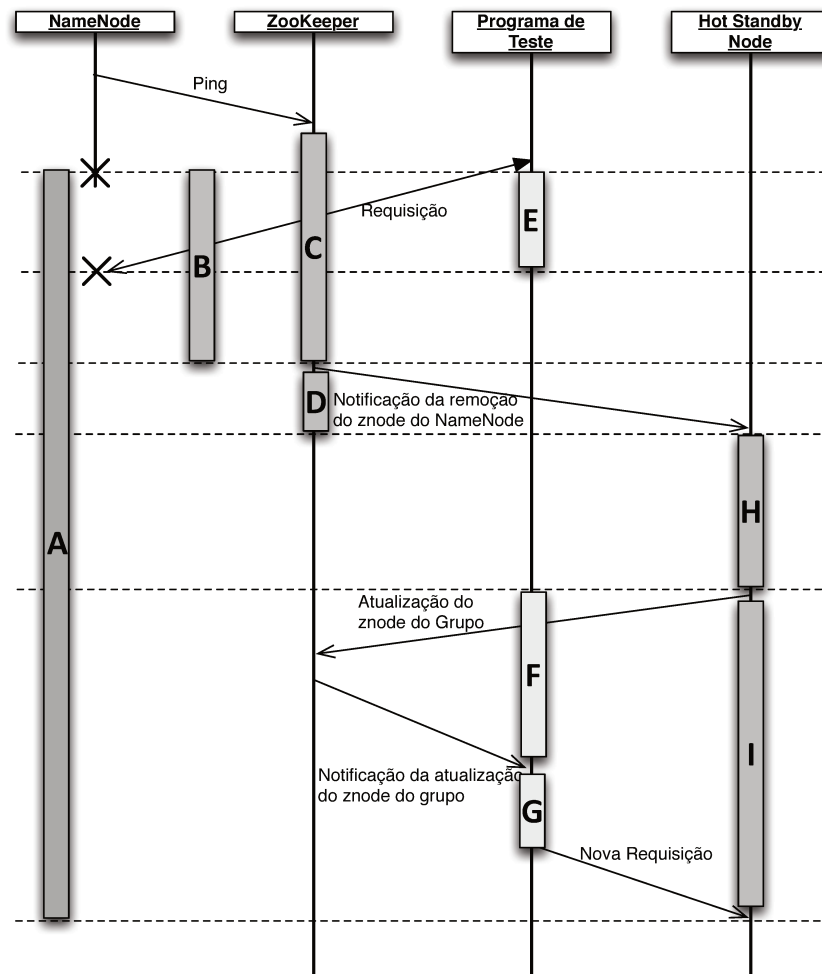


Figura 6.17: Intervalos para a análise do tempo de *failover*.

(intervalo **I**), considerando os desvios padrões, vê-se que realmente o Hot Standby Node é capaz de atender clientes logo após ter findo essa transição.

Já a situação do Hot Standby Node no cenário E/S é bem diferente. São também 20 arquivos abertos, porém são mais de 2.000 blocos não reconhecidos. Enquanto o Hot Standby Node faz a transição, ele também terá de processar mensagens de block-received para esses mais de 2.000 blocos. Como foi visto na seção anterior, esse processamento é custoso. O processamento de blocos domina sobre a transição, fazendo-a tomar muito mais tempo que no cenário Metadados para fazer o mesmo trabalho. Findo o processo de transição, o Hot Standby Node se tornará a autoridade sobre os DataNodes, passando a gerenciar os blocos. Como os dois mil blocos são desconhecidos ao Hot Standby Node, ele ordenará os DataNodes a removê-los. Em suma, findo o processo de transição, as

atividades relacionadas com gerenciamento de blocos iram predominar ainda mais sobre as outras atividades do Hot Standby Node. Isso explica o fato que embora os clientes possam reagir rapidamente à mudança do servidor de metadados (intervalo **G**), o Hot Standby Node somente consegue processar suas requisições muito tempo depois: de dois segundos para a melhor amostra a 64 segundos para a pior amostra.

O intervalo entre o início da transição do Hot Standby Node e o processamento da primeira requisição (soma dos intervalos **H** e **I**), que é o intervalo que é diretamente impactado pela implementação da solução Hot Standby Node, corresponde em média a 0,24% do tempo total de *failover* para o cenário Metadados e 22% para o cenário E/S. Este intervalo contém atividades que são dependentes do número de blocos e arquivos no sistema. Portanto é interessante verificar como o intervalo se comporta com a variação desses dois fatores. Para isso repetimos os experimentos de *failover* com 1000 e 500 blocos, mas nestes casos usamos apenas 3 amostras. Os resultados podem ser vistos na Figura 6.18.

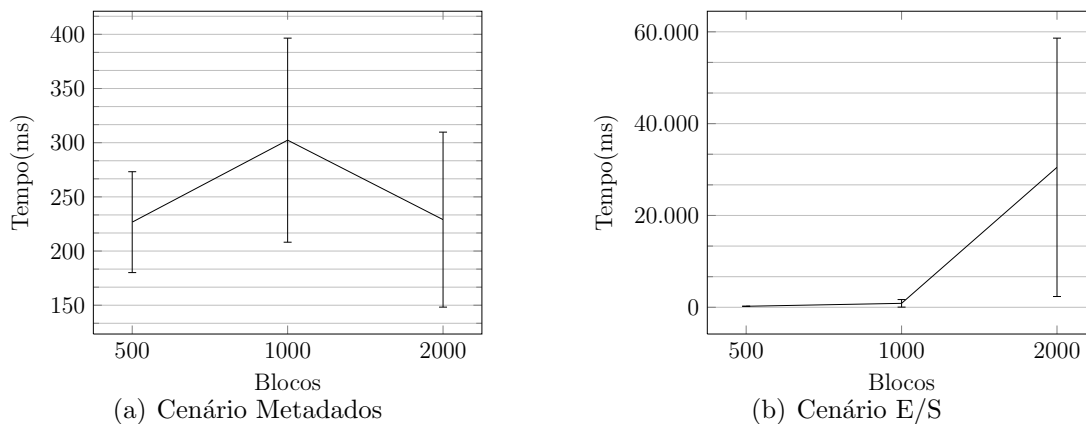


Figura 6.18: Tempo gasto para transição em relação ao número de blocos.

O fato dos dois gráficos não serem uma curva monotônica crescente mostra que o número de blocos não tem impacto considerável no processo de *failover* em si. O fato de apenas a Figura 6.18(b) seguir essa tendência e de que o desvio cresça com o número de blocos corrobora ainda mais para a tese de que o processamento de mensagens de block-received impacta indiretamente na transição. Quanto ao número de arquivos também não parece haver contribuição notável. Na Figura 6.18(a) o número de arquivos é equivalente ao de blocos e na Figura 6.18(b) é fixo em 20. As figuras não apontam um progressão conforme o número de arquivos aumenta.

Embora não foram feitas alterações no cliente do HDFS, os intervalos **F** e **G** indicam que é possível em nível de aplicação reagir rapidamente ao *failover* e que aplicação de geração de carga de trabalho teve impacto mínimo no tempo total de *failover*.

6.4 Considerações finais

Os resultados mostram que atingimos o objetivo de criar uma solução de alta disponibilidade com pouco impacto sobre os nós. No cenário Metadados, considerando os valores médios, houve um aumento de 6,0% no consumo de CPU para os DataNodes enquanto o uso de memória *heap* e de rede mantiveram-se os mesmos. Considerando-se os desvios não houve diferenças significativas entre os DataNodes do HDFS 0.21 e da Solução Hot Standby Node. O NameNode sofreu aumento de 4,7% no uso médio de CPU, aumento de 10,2% no uso médio de *heap*, redução de 0,2% no fluxo médio de RPC de entrada e aumento de 1,2% no de saída. Considerando-se os desvios, não há variação significativa no consumo de CPU e rede. Comparado com o NameNode da solução, o Hot Standby Node consome metade de CPU e $\frac{3}{4}$ de memória *heap*. Apesar de levar 1,16 vezes mais tempo que um *cluster* HDFS 0.21 para completar o mesmo trabalho, a solução Hot Standby Node consegue entregar 97% da vazão média de leitura e 95% da vazão média de escrita para os clientes em relação ao HDFS 0.21. Considerando-se os desvios, a vazão de dados manteve-se inalterada. E uma vez detectada a falha, o Hot Standby Node pode assumir o posto de servidor de metadados em menos de 0,4 segundos.

No cenário E/S os DataNodes usam 4,3% a mais de CPU, mantêm o mesmo uso de rede, mas passaram agora a usar 3,3% a mais de memória *heap* que os DataNodes do HDFS 0.21. Considerando-se os desvios, a variação no consumo de CPU deixa de ser significativa. O NameNode consumiu em média 10,7% a mais de CPU, 8,7% a mais de memória *heap*, mas reduziu seu tráfego RPC de entrada em 7,4% e o de saída em 14,1%. Entretanto, considerando os desvios, a diferença no consumo de CPU deixa de ser significativa. Já o Hot Standby Node apresentou uma considerável elevação no uso de recursos. Comparado com o NameNode da solução ele usa 4,4 vezes mais CPU e 1,5 vezes mais memória *heap*. E uma vez detectada a falha o Hot Standby Node pode levar até 60 segundos para atender uma requisição de cliente. Contudo, a solução Hot Standby Node gastou o mesmo tempo que o HDFS 0.21 para a mesma tarefa, entregando 91% da vazão média de leitura e a mesma vazão média de escrita.

Os resultados também provam que a técnica do Avatar Nodes de fato distribui os custos de replicação pelo *cluster*, não impondo grande ônus ao NameNode. Os resultados do cenário E/S mostram um potencial para melhoria se o problema do Hot Standby Node apenas conhecer os blocos quando o arquivo é fechado for abordado de outra forma.

Capítulo 7

Conclusão e trabalhos futuros

Conforme as chamadas tecnologias de *Big Data* comumente empregadas em sistemas em lote estão cada vez mais ocupando espaço em sistemas de tempo real e missão crítica, a questão da alta disponibilidade vem se tornando cada vez mais crucial. E sistemas de arquivos distribuídos baseados em *cluster* — um dos pilares de sustentação dessas tecnologias — certamente não estão imunes à essa pauta. Diante desse cenário tomamos por tópico de pesquisa a alta disponibilidade de sistemas de arquivos distribuídos baseados em *cluster*. Os esforços se concentraram em um dos mais largamente utilizados exemplares de tais sistemas: o Hadoop Distributed File System.

7.1 Iterações

Se por um lado a escolha pelo HDFS aumentou a relevância do projeto de pesquisa e garantiu uma comunidade considerável de usuários e desenvolvedores que poderiam auxiliar durante o desenvolvimento, por outro lado ela gerou algumas dificuldades. A base de código do Hadoop é grande, complexa e pouco documentada. Isso demandou um longo estudo dos detalhes de implementação do HDFS e várias trocas de emails na lista de desenvolvimento para esclarecer pontos mais obscuros.

Essa complexidade impôs barreiras para implementar algumas dos projetos concebidos durante o desenvolvimento da pesquisa. A primeira dessas ideias foi usar o ZooKeeper para armazenar informações sobre os DataNodes e as entradas do *log*. Cada DataNode criaria um znode efêmero onde manteria suas informações como espaço livre, carga, blocos, entre outras informações. Por ser um znode efêmero e conter todas as informações sobre o DataNodes, isso dispensaria a necessidade do envio de mensagens de estado (heartbeat, block-report e block-received) ou ainda de se registrar com o NameNode. Para cada entrada no *log* seria criado um znode sequencial para registrá-la. Isso garantiria a ordem entre as entradas do *log* e seu armazenamento estável. Com essa arquitetura bastaria

apenas para qualquer réplica do NameNode consultar o ZooKeeper para obter as entradas do *log* que lhe faltassem e recuperar informações sobre DataNodes e blocos. As réplicas registrar-se-iam para serem notificadas de eventos nos znodes, de forma que pudessem garantir que estavam sempre atualizadas. Além do conjunto grande de mudanças que essa solução impõem — pois mudou-se o local de onde as informações são obtidas — havia outras dificuldade técnicas. As respostas para as mensagens de heartbeat são usadas para enviar comandos para os DataNodes. Com o fim dos heartbeats, os comandos teriam de ser enviados de outra forma. Outra dificuldade foi também que não havia uma interface do ZooKeeper que permitisse recuperar um intervalo de znodes sequencias, já que réplicas estariam apenas interessadas nas entrada faltantes e não em todas. Houve então um estudo do funcionamento interno do ZooKeeper e tentativa de implementar essa nova funcionalidade.

A complexidade e as dificuldades técnicas somadas às incertezas sobre o desempenho final devido ao uso massivo do ZooKeeper levaram-nos a considerar outra solução. Usar replicação ativa seria mais interessante para alta disponibilidade devido à ausência de *fail-over* e a maior transparência de falhas para os clientes. No entanto realizar as mudanças para garantir o determinismo do NameNode e verificá-las seria um trabalho demasiadamente extenso. Também haveria o risco de a solução ser vetada pela comunidade, já que o requisito de determinismo do NameNode imporia o restrições ao uso de paralelismo, principal estratégia para desempenho em sistemas distribuídos.

Para contornar o problema causado pela imposição do determinismo cogitou-se então em usar replicação semi-passiva [33]. De maneira bem simples, a replicação semi-passiva é similar a replicação passiva. Porém todas as réplicas recebem a requisição do cliente, o que permite que elas continuem a requisição em caso de falha da réplica primária. Assim clientes observam um atraso ou invés de uma falha, como ocorreria na replicação passiva. Na replicação semi-passiva a réplica primária é responsável por executar a requisição e definir os parâmetros não deterministas a serem usados pelas outras réplicas por meio de sua mensagem de atualização de estado. Para do protocolo de replicação semi-passiva decidiu-se utilizar o Zab [55], o que rendeu uma implementação parcial do protocolo de *broadcast* atômico do ZooKeeper.

7.2 Projeto final e resultados

Porém mesmo com a solução empregando replicação semi-passiva seriam necessárias mudanças complexas e amplas no código de componentes do Hadoop que demandariam um tempo considerável de execução. Além do mais mudanças amplas no código poderiam gerar efeitos colaterais diversos, os quais são difíceis de serem depurados em ambientes distribuídos, o que poderia significar atrasos no projeto. Deste modo optamos por desen-

volver uma solução que fosse mais auto-contida, simples e que gerasse pouco impacto para outras partes de código. Com isso seria possível desenvolver uma solução com qualidade em uma janela de tempo compatível com um mestrado.

O resultado foi uma solução que reutiliza a replicação de metadados do Backup Node e a de informações sobre réplicas e DataNodes do Avatar Nodes. Além da replicação, a solução implementa um mecanismo automático de *failover* baseado no ZooKeeper. A solução gerou alterações em apenas 0,18% do código do HDFS, o que a torna simples para ser estudada e estendida por outros pesquisadores. E ela não usa componentes externos ao projeto Hadoop, o que a torna de fácil instalação e configuração. Embora simples, o Hot Standby Node representa um avanço perante outros sistemas de arquivos distribuídos baseados em *cluster* usados em *Big Data*. Isto porque esses sistemas normalmente usam *cold standbys* ou réplicas vivas que não podem exercer todas as funções de um servidor de metadados.

Em relação a outras propostas de alta disponibilidade para o HDFS tem-se vantagens e desvantagens. A solução desenvolvida não permite continuar escritas em progresso e não tolera falhas bizantinas ou mais de uma falha. Em compensação o Hot Standby Node pode reagir automaticamente a falhas em um curto espaço de tempo, de menos de 0,4 a 60 segundos. Ele não transfere o problema de disponibilidade para outros sistemas, nem impõem latência extra na replicação ou determinismo ao NameNode. E os resultados mostram outra vantagem da solução Hot Standby Node: baixo impacto sobre os nós do HDFS. Em média, o NameNode e os DataNodes não observam aumento superiores a 11% em seu consumo de CPU, memória ou na comunicação com outros nós. Considerando-se os desvios houve apenas aumento significativo no consumo de memória pelo NameNode e DataNode, porém esse aumento é no máximo de 8,7% e apenas no cenário E/S. Os clientes sentiram aumento de até 16% no tempo para realizarem suas tarefas. Em todos os casos houve sobreposição entre os desvios obtidos para vazão de dados da solução Hot Standby Node e do sistema original.

7.3 Trabalhos futuros

Apesar de bons, os resultados denunciam pontos de melhoria. Fazer o Hot Standby Node devolver blocos não reconhecidos provou-se não ser a melhor estratégia. O impacto sobre o consumo de recursos e o tempo de *failover* foi grande apesar do resultado geral ser bom. Outro ponto a ser trabalhado são as limitações do protótipo. Como trabalhos futuros podemos citar o suporte à continuação de escritas, à reconfiguração, a múltiplas falhas e o tratamento do *failover* pelo próprio cliente do HDFS.

Apesar de HDFS ter integrado sua própria solução enquanto a parte final dessa pesquisa transcorria, os resultados apresentados aqui ratificam como apropriada a ma-

neira pela qual dados sobre réplicas e DataNodes são replicados. Em vista de um replicação passiva tradicional em que o primário coordena toda a replicação, verificou-se que tal estratégia desonera o NameNode liberando-o para concentrar-se em suas atividades primárias. Além disso ela distribui o custo de replicação pelo *cluster*, promovendo um impacto reduzido no desempenho do sistema de arquivos. Mesmo com uma solução integrada, a alta disponibilidade do HDFS continua um assunto em aberto. Há ainda os esforços para tornar o sistema tolerante a várias falhas e para ter seu próprio mecanismo de armazenamento estável do *log*.

Referências Bibliográficas

- [1] PVFS2 High-Availability Clustering using Heartbeat 2.0. <http://www.pvfs.org/cvs/pvfs-2-7-branch.build/doc/pvfs2-ha-heartbeat-v2/pvfs2-ha-heartbeat-v2.php>. Último acesso em 18 de abril de 2013.
- [2] Message from general chairs. Em *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, páginas xii–xii, outubro.
- [3] Amazon EC2. <http://aws.amazon.com/ec2/>. Último acesso em 18 de abril de 2013.
- [4] Amazon EC2 Instance Types. <http://aws.amazon.com/ec2/instance-types/>. Último acesso em 18 de abril de 2013.
- [5] Amazon EC2 Pricing - Transferência de dados Pública e Elastic IP e Elastic Load Balancing. <http://aws.amazon.com/pt/ec2/pricing/>. Último acesso em 18 de abril de 2013.
- [6] Amazon S3 Growth for 2011 - Now 762 Billion Objects. <http://aws.typepad.com/aws/2012/01/amazon-s3-growth-for-2011-now-762-billion-objects.html>. Último acesso em 18 de abril de 2013.
- [7] Amazon Simple Storage Service. <http://aws.amazon.com/pt/s3/>. Último acesso em 18 de abril de 2013.
- [8] An Ethernet Address Resolution Protocol. <http://tools.ietf.org/html/rfc826>. Último acesso em 18 de abril de 2013.
- [9] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, e R. Y. Wang. Serverless network file systems. Em *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, páginas 109–126, Nova Iorque, NY, EUA, 1995. ACM.
- [10] Apache Hadoop. <http://hadoop.apache.org/>. Último acesso em 18 de abril de 2013.

- [11] Apache Zookeeper. <http://hadoop.apache.org/zookeeper/>. Último acesso em 18 de abril de 2013.
- [12] G. Attebury, A. Baranovski, K. Bloom, B. Bockelman, D. Kcira, J. Letts, T. Levshina, C. Lundestedt, T. Martin, W. Maier, Haifeng Pi, A. Rana, I. Sfiligoi, A. Sim, M. Thomas, e F. Wuerthwein. Hadoop distributed file system for the Grid. Em *Nuclear Science Symposium Conference Record (NSS/MIC)*, 2009 IEEE, páginas 1056 –1061, novembro de 2009.
- [13] Automatic failover support for NN HA. <https://issues.apache.org/jira/browse/HDFS-3042>. Último acesso em 18 de abril de 2013.
- [14] Big data. <http://www.symmetrymagazine.org/article/august-2012/big-data>. 1 de agosto de 2012 .Último acesso em 18 de abril de 2013.
- [15] Christophe Bisciglia. Hadoop HA Configuration. <http://www.cloudera.com/blog/2009/07/hadoop-ha-configuration/>. Último acesso em 18 de abril de 2013.
- [16] BookKeeper overview. <http://hadoop.apache.org/zookeeper/docs/r3.3.1/bookkeeperOverview.html>. Último acesso em 18 de abril de 2013.
- [17] Dhruba Borthakur. Hadoop AvatarNode High Availability. <http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html>. Último acesso em 18 de abril de 2013.
- [18] Dhruba Borthakur. HDFS High Availability. <http://hadoopblog.blogspot.com/2009/11/hdfs-high-availability.html>. Último acesso em 18 de abril de 2013.
- [19] Dhruba Borthakur. The High Availability story for HDFS so far. http://www.borthakur.com/ftp/hdfs_high_availability.pdf. Apresentado no ApacheCon em 5 de novembro de 2009 em Oakland, CA, EUA. Último acesso em 18 de abril de 2013.
- [20] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, e Amitanand Aiyer. Apache Hadoop Goes Realtime at Facebook. Em *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, páginas 1071–1080, Nova Iorque, NY, EUA, 2011. ACM.
- [21] Kenneth. Brayer. *Evaluation of 32 Degree Polynomials in Error Detection on the SATIN IV AUTOVON Error Patterns*. Defense Technical Information Center, Ft. Belvoir, 1975.

- [22] Amy Brown e Greg Wilson. *The Architecture Of Open Source Applications*. lulu.com, junho de 2011.
- [23] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, e Sam Toueg. Distributed systems (2nd ed.). capítulo The primary-backup approach, páginas 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.
- [24] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, e Rajeev Thakur. PVFS: a Parallel File System for Linux Clusters. Em *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, páginas 28–28, Berkeley, CA, EUA, 2000. USENIX Association.
- [25] Ceph. <http://ceph.com>. Último acesso em 18 de abril de 2013.
- [26] 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 Trans. Comput. Syst.*, 26(2):1–26, 2008.
- [27] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, e Taylor Riche. UpRight Cluster Services. Em *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, páginas 277–290, Nova Iorque, NY, EUA, 2009. ACM.
- [28] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, e Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. Em *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, páginas 153–168, Berkeley, CA, EUA, 2009. USENIX Association.
- [29] Comunicado 004/2012 - Atualização da Classificação de Conferências - Ciência da Computação - Qualis Capes . http://www.capes.gov.br/images/stories/download/avaliacao/Comunicado_004_2012_Ciencia_da_Computacao.pdf. Último acesso em 18 de abril de 2013.
- [30] George Coulouris, Jean Dollimore, e Tim Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, EUA, 2005.
- [31] Data Integrity and Availability in Apache Hadoop HDFS. <http://hortonworks.com/blog/data-integrity-and-availability-in-apache-hadoop-hdfs/>. Último acesso em 18 de abril de 2013.

- [32] Jeffrey Dean e Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, janeiro de 2008.
- [33] X. Defago, A. Schiper, e N. Sargent. Semi-passive replication. Em *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, páginas 43–50, 1998.
- [34] Alex Dorman e Paul George. Production deep dive with high availability. <http://www.scribd.com/doc/20971412/Hadoop-World-Production-Deep-Dive-with-High-Availability>. Apresentado no Hadoop World em 2 de outubro de 2009 em Nova Iorque, NY, EUA. Último acesso em 18 de abril de 2013.
- [35] DRBD. <http://www.drbd.org>. Último acesso em 18 de abril de 2013.
- [36] Exiting safemode takes a long time when there are lots of blocks in the HDFS. <https://issues.apache.org/jira/browse/HDFS-1391>. Último acesso em 18 de abril de 2013.
- [37] Facebook has the world’s largest Hadoop cluster! <http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html>. Último acesso em 18 de abril de 2013.
- [38] AvatarNodes implementation at Facebook’s code repository at Github. <https://github.com/facebook/hadoop-20/tree/master/src/contrib/highavailability>. Último acesso em 18 de abril de 2013.
- [39] Matt Foley. High Availability HDFS. Em *28th IEEE Conference on Massive Data Storage, MSST’12*, 2012. Apresentação técnica feita em 17 de abril.
- [40] Ganglia Monitoring System. <http://ganglia.info/>. Último acesso em 18 de abril de 2013.
- [41] Sanjay Ghemawat, Howard Gobioff, e Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.
- [42] C. Gray e D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.*, 23(5):202–210, novembro de 1989.
- [43] Rachid Guerraoui e André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.
- [44] Hadoop HDFS - ASF JIRA. <https://issues.apache.org/jira/browse/HDFS>. Último acesso em 18 de abril de 2013.

- [45] HBase. <http://hbase.apache.org/>. Último acesso em 18 de abril de 2013.
- [46] HDFS Architecture. http://hadoop.apache.org/common/docs/current/hdfs_design.html. Último acesso em 18 de abril de 2013.
- [47] Heartbeat. <http://www.linux-ha.org/wiki/Heartbeat>. Último acesso em 18 de abril de 2013.
- [48] High Availability for the Hadoop Distributed File System (HDFS). <http://www.cloudera.com/blog/2012/03/high-availability-for-the-hadoop-distributed-file-system-hdfs/>. Último acesso em 18 de abril de 2013.
- [49] High Availability Framework for HDFS NN. <https://issues.apache.org/jira/browse/HDFS-1623>. Último acesso em 18 de abril de 2013.
- [50] Hot Standby for NameNode. <http://issues.apache.org/jira/browse/HDFS-976>. Último acesso em 18 de abril de 2013.
- [51] How accurate are EC2 date/time clocks? <https://forums.aws.amazon.com/thread.jspa?messageID=50437>. Último acesso em 18 de abril de 2013.
- [52] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, e Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. Em *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, páginas 11–11, Berkeley, CA, EUA, 2010. USENIX Association.
- [53] Improve namenode restart times by short-circuiting the first block reports from datanodes. <https://issues.apache.org/jira/browse/HDFS-1295>. Último acesso em 18 de abril de 2013.
- [54] The Java Programming Language. [http://en.wikipedia.org/wiki/Java_\(programming_language\)](http://en.wikipedia.org/wiki/Java_(programming_language)). Último acesso em 18 de abril de 2013.
- [55] Flavio Junqueira, Benjamin Reed, e Marco Serafini. Dissecting zab. Relatório Técnico YL-2010-0007, Yahoo! Labs, Sunnyvale, Califórnia, outubro de 2010.
- [56] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, e Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, janeiro de 2010.
- [57] John Kubiawicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, e Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, novembro de 2000.

- [58] Link aggregation. http://en.wikipedia.org/wiki/Link_aggregation. Último acesso em 18 de abril de 2013.
- [59] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
- [60] Large segment offload. http://en.wikipedia.org/wiki/Large_segment_offload. Último acesso em 18 de abril de 2013.
- [61] Log newly allocated blocks. <https://issues.apache.org/jira/browse/HDFS-1108>. Último acesso em 18 de abril de 2013.
- [62] Lustre: A scalable, high-performance file system. <http://www.cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf>, novembro de 2002. Último acesso em 18 de abril de 2013.
- [63] Lustre 2.x Filesystem Operations Manual. <http://build.whamcloud.com/job/lustre-manual/lastSuccessfulBuild/artifact/lustre-manual.pdf>. Último acesso em 18 de abril de 2013.
- [64] Carlos Maltzahn, Esteban Molina-Estolano, Amandeep Khurana, Alex Nelson, Scott Brandt, e Sage Weil. Ceph as a scalable alternative to the Hadoop Distributed File System. *login: The Usenix Magazine*, 35(4):38–49, agosto de 2010.
- [65] MemoryMXBean- Java™ Platform Standard Ed. 6. <http://docs.oracle.com/javase/6/docs/api/java/lang/management/MemoryMXBean.html>. Último acesso em 18 de abril de 2013.
- [66] Moving an Elephant: Large Scale Hadoop Data Migration at Facebook. <http://www.facebook.com/notes/facebook-engineering/moving-an-elephant-large-scale-hadoop-data-migration-at-facebook/10150246275318920>. Último acesso em 18 de abril de 2013.
- [67] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, e Benjie Chen. Ivy: a read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, dezembro de 2002.
- [68] David Nagle, Denis Serenyi, e Abbie Matthews. The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage. Em *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, páginas 53–, Washington, DC, EUA, 2004. IEEE Computer Society.

- [69] Namenode cluster(multi namenode working in active-active mode) to keep high availability and expansivity. <https://issues.apache.org/jira/browse/HDFS-243>. Último acesso em 18 de abril de 2013.
- [70] NN Availability - umbrella Jira. <http://issues.apache.org/jira/browse/HDFS-1064>. Último acesso em 2 de dezembro de 2012.
- [71] NameNode cluster's code repository at Github. <https://github.com/gnawux/hadoop-cmri/>. Último acesso em 18 de abril de 2013.
- [72] Network Time Protocol v4. <http://tools.ietf.org/html/rfc5905>. Último acesso em 18 de abril de 2013.
- [73] Object Storage Device. http://en.wikipedia.org/wiki/Object_storage_device. Último acesso em 18 de abril de 2013.
- [74] Open Scalable File Systems Inc. <http://www.opensfs.org/>. Último acesso em 18 de abril de 2013.
- [75] André Oriani e Islene Calciolari Garcia. A High Availability Solution for The Hadoop Distributed File System. Em *Anais do VII Workshop de Teses, Dissertações e Trabalhos de Iniciação Científica em Andamento do IC-UNICAMP*, páginas 6–7. Instituto de Computação, Universidade Estadual de Campinas, junho de 2012.
- [76] André Oriani e Islene Calciolari Garcia. From Backup to Hot Standby: High Availability for HDFS. Em *Proceedings of 31st International Symposium on Reliable Distributed Systems, SRDS'12*, páginas 131–140. IEEE Computer Society, outubro de 2012.
- [77] André Oriani, Islene Calciolari Garcia, e Rodrigo Schmidt. The Search for a Highly-Available Hadoop Distributed Filesystem. Relatório Técnico IC-10-24, Instituto de Computação, Universidade Estadual de Campinas, agosto de 2010. Em Inglês, 27 páginas.
- [78] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, e Pan Hui. Exploiting hardware heterogeneity within the same instance type of Amazon EC2. Em *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing, HotCloud'12*, páginas 4–4, Berkeley, CA, EUA, 2012. USENIX Association.
- [79] Parallel Virtual File System, Version 2. <http://www.orangefs.org/documentation/releases/current/doc/pvfs2-guide/pvfs2-guide.php>. Último acesso em 18 de abril de 2013.

- [80] David A. Patterson, Garth Gibson, e Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). Em *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, páginas 109–116, Nova Iorque, NY, EUA, 1988. ACM.
- [81] The Open Group Base Specifications issue 7. <http://pubs.opengroup.org/onlinepubs/9699919799/>. Último acesso em 18 de abril de 2013.
- [82] Problem: ZooKeeper SessionExpired events – Hadoop Wiki. <http://wiki.apache.org/hadoop/Hbase/Troubleshooting#A8>. Último acesso em 18 de abril de 2013.
- [83] Proc - Process information pseudo-file system. <http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html>. Último acesso em 18 de abril de 2013.
- [84] psutil - A cross-platform process and system utilities module for Python. <http://code.google.com/p/psutil/>. Último acesso em 18 de abril de 2013.
- [85] Record every new block allocation of a file into the transaction log. <https://issues.apache.org/jira/browse/HDFS-978>. Último acesso em 18 de abril de 2013.
- [86] Benjamin Reed e Flavio P. Junqueira. A simple totally ordered broadcast protocol. Em *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, páginas 2:1–2:6, Nova Iorque, NY, EUA, 2008. ACM.
- [87] Philipp Reisner e Lars Ellenberg. Drbd v8 - Replicated storage with shared disk semantics. Em *Proceedings of the 12th International Linux System Technology Conference (Linux-Kongress) 2005*, Hamburgo, Alemanha, outubro de 2005.
- [88] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, e B. Lyon. Innovations in internetworking. capítulo Design and implementation of the Sun network filesystem, páginas 379–390. Artech House, Inc., Norwood, MA, EUA, 1988.
- [89] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: a Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.
- [90] Jame E. Short, Roger E. Bohn, e Chaitanya Baru. How much information? 2010 report on enterprise server information. http://hmi.ucsd.edu/pdf/HMI_2010_EnterpriseReport_Jan_2011.pdf. Global Information Industry Center at the School of International Relations and Pacific Studies, UC San Diego. Último acesso em 18 de abril de 2013.
- [91] Konstantin Shvachko. HDFS Scability: The Limits to Growth. *login: The Usenix Magazine*, 35(2):6–16, abril de 2010.

- [92] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, e Robert Chansler. The Hadoop Distributed File System. Em *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, páginas 1–10, jul de 2010.
- [93] Abraham Silberschatz, Peter Baer Galvin, e Greg Gagne. *Operating System Concepts - 8th edition*. Wiley Publishing, 2008.
- [94] Speedup NameNode image loading and saving by storing local file names. <https://issues.apache.org/jira/browse/HDFS-1070>. Último acesso em 18 de abril de 2013.
- [95] STONITH. <http://linux-ha.org/wiki/STONITH>. Último acesso em 18 de abril de 2013.
- [96] Sun Microsystems, Inc., Santa Clara, CA, EUA. Lustre file system - High-performance storage architecture and scalable cluster file system. http://www.raidinc.com/assets/documents/lustrefilesystem_wp.pdf, 2007. Último acesso em 18 de abril de 2013.
- [97] Chandramohan A. Thekkath, Timothy Mann, e Edward K. Lee. Frangipani: a scalable distributed file system. Em *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, páginas 224–237, Nova Iorque, NY, EUA, 1997. ACM.
- [98] Using Amazon EC2 public IP address inside EC2 network. <http://www.cloudiquity.com/2009/02/using-amazon-ec2-public-ip-address-inside-ec2-network/>. Último acesso em 18 de abril de 2013.
- [99] Virtual IP - Wikipedia. http://en.wikipedia.org/wiki/Virtual_IP_address. Último acesso em 18 de abril de 2013.
- [100] Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, e Ying Li. Hadoop High Availability through Metadata Replication. Em *CloudDB '09: Proceeding of the first international workshop on Cloud data management*, páginas 37–44, Nova Iorque, NY, EUA, 2009. ACM.
- [101] Guohui Wang e T. S. Eugene Ng. The impact of virtualization on network performance of amazon EC2 data center. Em *Proceedings of the 29th conference on Information communications*, INFOCOM'10, páginas 1163–1171, Piscataway, NJ, EUA, 2010. IEEE Press.
- [102] Xu Wang. Pratices of NameNode Cluster for HDFS HA. <http://gnawux.info/hadoop/2010/01/practice-of-namenode-cluster-for-hdfs-ha/>. Último acesso em 18 de abril de 2013.

- [103] Sage A. Weil. *Ceph: reliable, scalable, and high-performance distributed storage*. PhD thesis, Santa Cruz, CA, EUA, 2007. AAI3288383.
- [104] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, e Carlos Maltzahn. Ceph: a Scalable, High-performance Distributed File System. Em *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, páginas 307–320, Berkeley, CA, EUA, 2006. USENIX Association.
- [105] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, e Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. Em *Proceedings of the 2004 ACM/IEEE conference on Supercomputing, SC '04*, páginas 4–, Washington, DC, EUA, 2004. IEEE Computer Society.
- [106] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, e Bin Zhou. Scalable Performance of the Panasas Parallel File System. Em *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, páginas 1–17, Berkeley, CA, EUA, 2008. USENIX Association.
- [107] When checkpointing by backup node occurs paralely when a file is being closed by a client then Exception occurs saying no journal streams. <http://issues.apache.org/jira/browse/HDFS-1989>. Último acesso em 18 de abril de 2013.
- [108] Tom White. *Hadoop: The Definitive Guide - 3rd Edition*. O'Reilly Media Inc, Sebastopol, CA, maio de 2012.
- [109] Worldwide LHC Computing Grid. <http://public.web.cern.ch/public/en/lhc/Computing-en.html>. Último acesso em 18 de abril de 2013.
- [110] Write - The Open Group Base Specifications issue 7. <http://pubs.opengroup.org/onlinepubs/9699919799/functions/write.html>. Último acesso em 18 de abril de 2013.
- [111] Yahoo! Launches World's Largest Hadoop Production Application. <http://developer.yahoo.com/blogs/hadoop/posts/2008/02/yahoo-worlds-largest-production-hadoop/>. Último acesso em 18 de abril de 2013.
- [112] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, e Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. Em *Proceedings of the nineteenth ACM symposium on Operating systems principles, SOSP '03*, páginas 253–267, Nova Iorque, NY, EUA, 2003. ACM.
- [113] Youtube Facts and Figures With Infographic. <http://www.speakymagazine.com/24-youtube-facts-and-figures-with-infographic/>. SpeakyMagazine de 6 de abril de 2012. Último acesso em 18 de abril de 2013.