

Dissertação de Mestrado

Estudo e Implementação de Mecanismos de Codificação
por Apagamento no Hadoop *File System*

Celina d'Ávila Samogin
Instituto de Computação — UNICAMP

5 de julho de 2011

Orientadora: Profa. Dra. Islene Calciolari Garcia

Resumo

Os dados em um sistema distribuído confiável devem estar disponíveis quando for necessário. A codificação por apagamento (*erasure codes*) tem sido utilizada por sistemas para alcançar requisitos de confiabilidade e de redução do custo de armazenamento de dados. O Hadoop é um *framework* para execução de aplicações em armazenamento distribuído de grande volume de dados e que pode ser construído com *commodity hardware*, que é facilmente acessível e disponível. Esta proposta apresentará uma análise da viabilidade da implementação prática de técnicas de codificação por apagamento no Hadoop *Distributed File System* (HDFS), as alterações no Hadoop e a eficácia dessas alterações. Esta proposta é uma contribuição para *software* livre em sistemas distribuídos.

Abstract

The data in a reliable distributed system should be available when needed. Erasure codes have been used by systems to meet reliability requirements and reduce the cost of data storage. The Hadoop is a framework for running applications on distributed storage of large volumes of data and it can be built with commodity hardware, which is easily accessible and available. This proposal will examine the feasibility of practical implementation of erasure coding techniques in Hadoop Distributed File System (HDFS), changes in Hadoop and effectiveness of those changes. This proposal is a contribution to free software in distributed systems.

Sumário

1	Introdução	7
2	Álgebra Abstrata	8
2.1	Congruência	8
2.2	Classe Residual	9
2.3	Corpo	9
2.4	Corpo de Galois	10
2.4.1	Ordem do Corpo	10
2.4.2	Ordem do Elemento	11
2.4.3	Aritmética de Corpo Binário	11
2.4.4	Propriedades dos Polinômios e suas Raízes	11
2.5	Construção de um Código Corretor de Erros	11
2.5.1	Métrica de Hamming	11
3	Codificação por Apagamento	12
3.1	Shannon: conceitos e teoremas fundamentais	13
3.2	Canais: modelos e erros	14
3.3	Transmissão e armazenamento de informação	16
3.4	Técnicas para detecção e correção de erros	18
3.4.1	<i>Automatic Repeat reQuest</i> - ARQ	18
3.4.2	<i>Forward Correction Code</i> - FEC	18
3.4.3	Como FEC funcionam	19
3.5	Códigos Convolucionais	22
3.6	Características de Códigos de Blocos	23
3.7	Códigos de Blocos Lineares	24
3.7.1	Códigos Reed-Solomon	24

3.7.2	Códigos RAID	25
3.7.3	Códigos Low-Density Parity-Check e Tornado	27
4	Discussão sobre esquemas adequados de redundância de dados	28
4.1	Esquemas de redundância de dados para sistema de armazenamento	29
4.1.1	Replicação	29
4.1.2	Codificação por Apagamento	32
4.2	Caracterização da replicação e da codificação	32
4.2.1	Replicação	34
4.2.2	Codificação por Apagamento	35
4.3	Sobrecarga de armazenamento	36
4.4	Disponibilidade dos peers	39
4.5	Leitura ou Atualização dos dados redundantes	39
5	Hadoop	39
5.1	MapReduce	40
5.2	Arquitetura do Hadoop <i>Distributed File System</i>	41
5.3	Codificação por apagamento	44

Lista de Símbolos e Abreviaturas

bit BInary digiT

BSD Berkeley Software Distribution

CD Compact Disk

CRC Cyclic Redundancy Check

DVD Digital Versatile Disc

ECC Error Correcting Code

GF Galois Field

GPL GNU General Public License

GPL2 GPL versão 2

IEEE Institute of Electrical and Electronics Engineers

LGPL GNU Lesser General Public License

MAID Massive Arrays of Idle Disks

NASA National Aeronautics and Space Administration

PNG Portable Network Graphics

POSIX Portable Operating System Interface

RAID Redundant Arrays of Independent Disks

RS Reed-Solomon

WMAN Wireless Metropolitan Area Network

WiMAX Worldwide Interoperability for Microwave Access

XOR Exclusive OR

1 Introdução

A codificação por apagamento (*erasures codes*) introduz redundância em um sistema de transmissão ou armazenamento de dados de maneira a permitir a detecção e correção de erros. A codificação por apagamento é, desde os anos 70, utilizada pela *NASA's Deep Space Network* para receber sinais e dados de telemetria (*downlinks*) vindos de veículos espaciais (*very distant spacecrafts*) e para enviar telecomandos (*uplinks*) para veículos espaciais [28, 42, 59].

A técnica de codificação por apagamento pode ser combinada com a distribuição de dados entre vários dispositivos de armazenamento, o que permite o aumento da largura de banda e a correção de erros [33, 49]. Requisitos de confiabilidade e de redução do tamanho do armazenamento podem ser observados em sistemas que tratam de: *Delay and Disruption Tolerant Networks*, redes de sensores e redes *peer-to-peer* [29, 37, 5, 4, 6, 52] e armazenamento de grande volume de dados [8, 3, 46, 50, 21, 26, 27], como também o sistema de arquivos distribuído do Hadoop (HDFS) [16].

O HDFS, por padrão, implementa alta disponibilidade dos dados via replicação simples dos blocos de dados. Esta abordagem acarreta um alto custo de armazenamento para garantir que os dados estarão sempre disponíveis. O objetivo do uso da codificação por apagamento no HDFS é permitir que o espaço de armazenamento possa ser reduzido sem prejudicar a disponibilidade dos dados. Esforços iniciais nessa linha foram feitos utilizando técnicas de RAID [16] e mais recentemente do algoritmo Reed-Solomon [18].

Este trabalho pretende avançar esta linha de pesquisa a partir dos seguintes passos:

- avaliação de desempenho, ganhos, e custos de diferentes estratégias de codificação por apagamento;
- implementação de otimizações ou extensões para o código que atualmente implementa Reed-Solomon, tentando melhorar, principalmente, a parte de distribuição de blocos;

- implementação de novos algoritmos (e.g., Tornado codes) e extensão da interface atual para aceitá-los;
- integração do código atual com o HDFS.

O texto a seguir está organizado da seguinte maneira: a Seção 2 introduz os conceitos básicos da codificação por apagamento, a Seção 3 comenta o *framework* Hadoop e seu sistema de arquivos, a Seção 4 apresenta os objetivos deste trabalho e a seção 5 cita as atividades propostas e o cronograma de execução.

2 Álgebra Abstrata

Essa seção tem por objetivo apresentar conceitos matemáticos fundamentais dentro do escopo de álgebra abstrata para entendimento de Códigos de Blocos. Alguns textos foram estudados para que ela pudesse ser escrita: [51].

Códigos RS são projetados através da aritmética de corpos finitos. Corpos finitos também são chamados corpos de Galois, em homenagem ao matemático francês Évariste Galois.

2.1 Congruência

Definição 1. Seja $m \in \mathbb{N}$. Dois inteiros a e b dizem-se *congruentes modulo m* se tiverem o mesmo resto na divisão por m . A notação é $a \equiv b(mod\ m)$. Daí temos que $a = mk + b$, para um $k \in \mathbb{N}$. Exemplos:

- $32 \equiv 2(mod\ 3)$
- $27 \equiv 5(mod\ 11)$
- $63 \equiv 7(mod\ 8)$

2.2 Classe Residual

Definição 2. Seja $m \in \mathbb{N}$ e $m > 1$. A classe residual módulo m do elemento $a \in \mathbb{Z}$ pode ser assim definida:

$$\bar{a} = \{x \in \mathbb{Z} \mid x \equiv a \pmod{m}\}$$

Exemplos:

- se $m = 2$, então temos 2 classes residuais: $\bar{0} = \{\dots - 4, -2, 0, 2, 4, \dots\}$ e $\bar{1} = \{\dots - 3, -1, 1, 3, \dots\}$
- se $m = 3$, então temos 3 classes residuais: $\bar{0}, \bar{1}, \bar{2}$
- se $m = 5$, então temos 4 classes residuais: $\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}$

Definição 3. $\mathbf{Z}_m = \{\bar{0}, \bar{1}, \dots, \overline{m-1}\}$ é o conjunto das classes residuais dos inteiros módulo m .

Notemos que \mathbf{Z}_m tem exatamente m elementos, portanto \mathbf{Z}_m é finito.

Definição 4. Se $a \equiv b \pmod{m}$, então $\bar{a} = \bar{b}$

2.3 Corpo

Um corpo é um conjunto F que resume-se um espaço fechado com operações binárias, como ”.” e ”+”, entre dois dos seus elementos, designados por operandos. O resultado da aplicação de uma operação resulta em um terceiro elemento também pertencente a F . As propriedades dessas operações são: associativa e comutativa e a operação ”.” é distributiva sobre a operação ”+”: $a.(b + c) = a.b + a.c, \forall a, \forall b, \forall c \in F$. Os elementos de F apresentam essas propriedades: existência de elemento identidade (neutro) em F para a operação ”.” e para a operação ”+” e existência de elemento um inverso da operação ”+” para cada elemento de F .

Os conjuntos como F podem ter ordem (por exemplo, cardinalidade) infinita. Como exemplos de corpos infinitos, temos os conjuntos dos números racionais \mathbb{Q} , reais \mathbb{R} , complexos \mathbb{C} e os inteiros $\text{mod } p$ (p é primo) sob as operações de adição e multiplicação usuais.

2.4 Corpo de Galois

Um Corpo de Galois é um corpo de ordem finita ou seja, sua cardinalidade é conhecida.

Exemplo: Seja o conjunto $G = \{0, 1\}$ e a operação binária \oplus em G : $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$.

A operação \oplus é chamada adição módulo 2. Portanto G é fechado em \oplus e \oplus é comutativa. Também é possível demonstrar que \oplus é associativa. O elemento 0 é o elemento identidade. Os inversos de cada um dos elementos de G também pertencem a G .

Corpo

2.4.1 Ordem do Corpo

O número de elementos de um corpo finito G é denominado ordem de G . Um corpo de Galois de ordem q é representado por $GF(q)$. Um corpo finito G tem ordem p^n , onde p é a característica do corpo G e $n = [K : Z_p]$. Todo o corpo finito tem p^n elementos para algum primo p e algum natural n . Para cada primo p e cada natural n , existe um corpo com p^n elementos. Qualquer corpo com p^n elementos é isomorfo à extensão de decomposição de $x^q - x$, $q = p^n$ sobre $GF(q)$. Exemplo:

- Corpos de Galois de ordem 2: $GF(2) = \{0, 1\}$
- Corpos de Galois de ordem 3: $GF(3) = \{0, 1, 2\}$
- Corpos de Galois de ordem 4: $GF(2^2) = GF(4) = \{0, 1, 2, 3\}$
- Corpos de Galois de ordem 5: $GF(5) = \{0, 1, 2, 3, 4\}$
- Corpos de Galois de ordem 7: $GF(7) = \{0, 1, 2, 3, 4, 5, 6\}$
- Corpos de Galois de ordem 8: $GF(2^3) = GF(8) = \{0, 1, 2, 3, 4, 5, 6, 7\}$

2.4.2 Ordem do Elemento

Seja $a_i \in GF(q)$. A ordem de a_i , representada por $ord(a_i)$, é o menor inteiro positivo o tal que $a_i^o = 1$. Se elevarmos ambos os lados a potência $q - 1$, temos que $(a_i^o)^{q-1} = 1$. Multiplicando ambos os lados por a_i^o , temos que $(a_i^o)^q = a_i^o$ e substituindo a_i^o por x , temos $x^q = x$. Portanto, todos os $a_i \in GF(p)$ satisfazem a equação: $x^q - x = 0, q = p^n$.

2.4.3 Aritmética de Corpo Binário

Códigos são construídos com elementos de um corpo $GF(q)$ onde q ou é primo p ou uma potência de p . Os dados de sistemas de transmissão e armazenamento de dados podem ser facilmente codificados em códigos com símbolos gerados a partir de um corpo $GF(2)$ ou $GF(2^m)$ [23].

2.4.4 Propriedades dos Polinômios e suas Raízes

$GF[q](x)$ é um corpo de Galois que tem ordem q , sobre o qual são aplicados polinômios de grau n com coeficientes entre 0 e $q - 1$. Um exemplo de um polinômio $p(x)$ pode ser apresentado na equação: $p(x) = a_7.x^7 + a_5.x^5 + a_2.x^2 + ax + 1$. Os coeficientes de um polinômio a_i pertencem ao $GF(p)$ e o grau do polinômio n , pode ter um valor qualquer.

2.5 Construção de um Código Corretor de Erros

2.5.1 Métrica de Hamming

página 4 - livro Hefez página 7 - Parâmetros fundamentais $[n, M, d]$

O objetivo da codificação de canal é aumentar a resistência do sistema de comunicações digital face aos efeitos do ruído de canal. No caso particular dos códigos de bloco, a cada bloco de k bits da sequência binária gerada pela fonte faz-se corresponder um bloco de n bits (palavra de código) com $n \geq k$. Este processo de codificação deve ser concebido de modo que a decodificação tenha solução única. Note-se que do universo de 2^n blocos binários de

comprimento n apenas 2^k são palavras de código (as que correspondem numa relação de um para um aos blocos binários de comprimento k gerados pela fonte). Este esquema está representado simbolicamente na Figura 8.4. No caso de a transmissão se efetivar sem erros, o processo de decodificação conduz ao bloco de comprimento k que havia sido gerado pela fonte. Quando ocorrem erros de transmissão, a palavra de comprimento n recebida pode não ser uma palavra de código e o erro é detectado e/ou corrigido. Figura 8.4: Codificação de canal

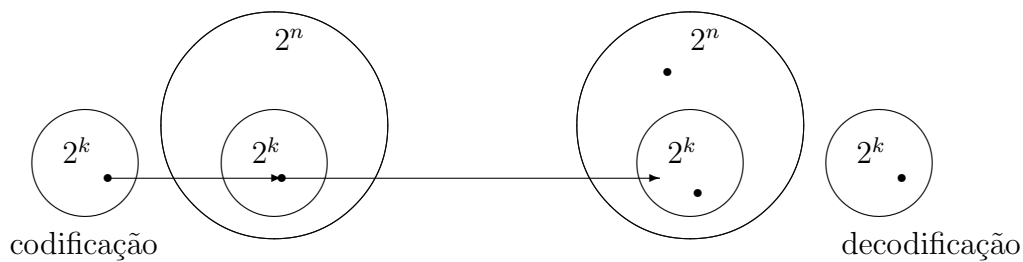


Figura 1: Figura 1 - Codificação de canal

Teoria da Informação: Capacidade do Canal de Transmissão

Codificação de Canal: Códigos de Bloco Lineares

3 Codificação por Apagamento

A transmissão e o armazenamento de informação tem muito em comum. Ambos transferem dados da fonte para o destino. Para garantir confiabilidade nessas operações, é utilizada codificação por apagamento ou códigos corretores de erros.

Em um trabalho publicado em 1948, o matemático Claude E. Shannon demonstrou que a eficiência da codificação de mensagens do emissor antes da transmissão e da decodificação das mensagens (possivelmente danificadas) que chegam no receptor, possibilita reparar, para um nível aceitável, os efeitos de um canal físico com ruídos [14] sem sobrecarregar a taxa de transmissão de informação ou o *overhead* de armazenamento [23]. A Teoria dos Códigos

tem sido estudada há décadas: por matemáticos nas décadas de 50 e 60 e a partir da década de 70 por engenheiros [51].

Com a popularização dos computadores e as pesquisas espaciais, os códigos corretores tornaram-se parte comum de comunicações por satélite, de redes de computadores, de armazenamento em discos óticos e outros meios magnéticos. A presença dos códigos corretores de erros é frequente em nosso cotidiano: quando se assiste a um programa de televisão, quando se ouve música a partir de um CD, quando se faz um telefonema, quando se assiste um filme gravado em DVD, quando se navega pela internet.

3.1 Shannon: conceitos e teoremas fundamentais

Shannon introduziu dois conceitos fundamentais sobre informação que é transmitida em um sistema de comunicação [60]:

a incerteza da informação se o dado que nos interessa é determinístico, então ele não tem valor algum. Por exemplo, a transmissão contínua de uma imagem em um sistema de televisão é supérflua. Desta forma, a fonte de informação é modelada por uma variável ou processo aleatório e uma probabilidade é utilizada para se desenvolver a teoria da informação.

a informação transmitida é digital o dado que nos interessa deve ser convertido em *bits* e ser entregue no destino corretamente, sem referência ao seu significado inicial. Esse trabalho do Shannon parece ser o primeiro trabalho publicado que usa o termo *bit* (*binary digit*).

Nesse mesmo trabalho, Shannon demonstrou dois importantes teoremas que são fundamentais na comunicação ponto-a-ponto:

source coding theorem introduz a entropia como medida da informação que, nesse caso, é caracterizada por a taxa mínima de código que representa uma informação livre de erros. Este teorema é a base teórica para compressão de dados.

channel coding theorem fala da capacidade de um canal com ruídos, na qual a informação é transmitida de forma confiável, desde que o ritmo de transferência de dados seja menor que a capacidade do canal.

A probabilidade de erro no canal determina a capacidade de transferência de informação no canal.

3.2 Canais: modelos e erros

O código da fonte é o conjunto de elementos que definem como a informação será transmitida, por exemplo, em termos de *bits* e o código do canal inclui o código da fonte e a redundância (por exemplo, em termos de *bits*) introduzida para garantir a correção de erros.

Uma dificuldade encontrada por quem estuda códigos corretores de erros é que não existe uma nomenclatura unificada [62]. Também segundo [24], existem poucos pesquisadores que são programadores de sistemas e que fazem propostas neste tema.

A idéia básica do código ótimo é que o objeto original possa ser reconstruído a partir de quaisquer m únicos fragmentos que são aproximadamente do mesmo tamanho do objeto original [12]. Para uma codificação quase ótima, são necessários $\approx (1 + e)m$ fragmentos, onde $e \geq 1$ [39].

Corrigir erros é uma tarefa mais complexa que detectá-los. Detectar erros tem a mesma complexidade que a operação de codificar, que pode ser linear no tamanho das palavras código. A operação de decodificar para uma correção de erros ótima é um problema NP-difícil e são conhecidos apenas algoritmos eficientes para algumas classes de códigos. Novas classes de códigos com eficientes decodificadores e novos algoritmos para decodificação para códigos conhecidos são promissoras pesquisas [53].

Um dos principais parâmetros de um código para detecção de erros é a probabilidade de detecção de erro. Os modelos estudados por pesquisadores envolvem canais simétricos, assimétricos e outros, com ou sem memória.

Em canais binários simétricos, assume-se que ambos os erros $0 \rightarrow 1$ e $1 \rightarrow 0$ ocorrem com igual probabilidade [32].

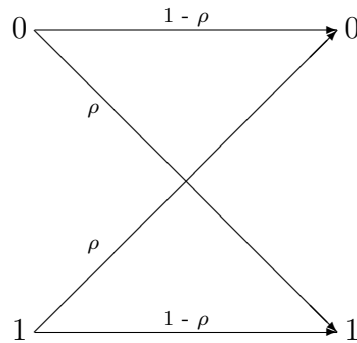


Figura 2: Figura 2 - Canal binário simétrico [32]

ρ = probabilidade da ocorrência de um erro

Em algumas aplicações como comunicações óticas, os erros tem uma natureza assimétrica. Canais, onde esse tipo de erro ocorrem, podem ser modelados para canais binários assimétricos ou canal-Z, onde apenas erros com 1's ocorrem. Um exemplo disso são sistemas que utilizam *photons* para transmitir informação [32].

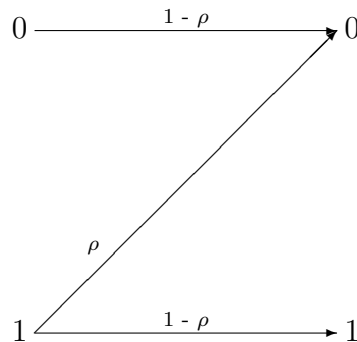


Figura 3: Figura 3 - Canal binário assimétrico [32]

ρ = probabilidade da ocorrência de um erro

3.3 Transmissão e armazenamento de informação

A transmissão e o armazenamento de informação tem muito em comum. Ambos transferem dados da fonte para o destino.

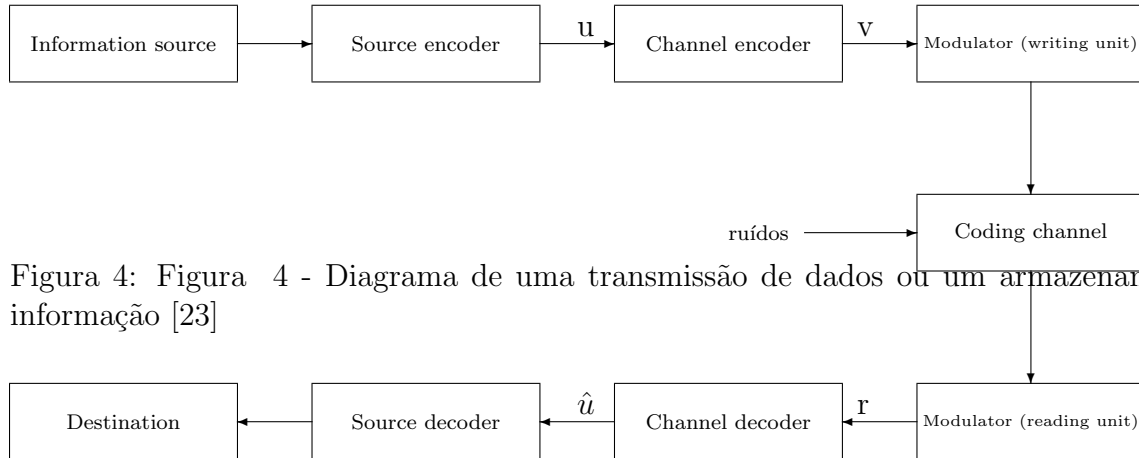


Figura 4: Figura 4 - Diagrama de uma transmissão de dados ou um armazenamento de informação [23]

Na figura 4, o *source encoder* transforma a saída da fonte em uma sequência de bits u . Esse codificador é projetado para que o número de bits que represente a saída da fonte seja mínimo e para que a saída da fonte possa ser reconstruída sem ambiguidade.

O *channel encoder* transforma a sequência u em uma sequência codificada discreta v , também chamada *codeword*. As instâncias de v também são sequências binárias e permitem códigos não-binários.

O *modulator* transforma cada símbolo da sequência v em uma onda de duração de T segundos que pode ser transmitida ou gravada. Esta onda entra em um canal ou mídia

de armazenamento e é corrompida pelo ruído. Os canais de transmissão podem ser linhas telefônicas, *links* de rádio, de telemetria, de satélite. Mídias de armazenamento podem ser memórias, fitas magnéticas, discos magnéticos, unidades de memória ótica. Cada um desses canais ou mídias podem ser expostos a vários ruídos.

O *demodulator* recebe a onda de duração de T segundos e produz uma saída que pode ser discreta ou contínua. A sequência de saída do *demodulator* corresponde a sequência codificada v e é chamada de sequência recebida r .

O *channel decoder* transforma a sequência recebida r em uma sequência binária \hat{u} chamada de sequência estimada. Idealmente a sequência \hat{u} será uma réplica da sequência u original.

O *source decoder* transforma a sequência estimada \hat{u} em uma sequência o mais próxima possível da saída da fonte e a entrega para o sumidouro.

Em um diagrama mais simplificado (figura 5), estão enfatizadas as operações de codificação e decodificação. A *information source* e o *source encoder* são combinados em uma fonte digital com saída u . O *modulator*, o *channel encoder* e o *demodulator* são combinados em um *coding channel* com entrada v e saída r . O *source decoder* e o *destination* são combinados em um sumidouro digital.

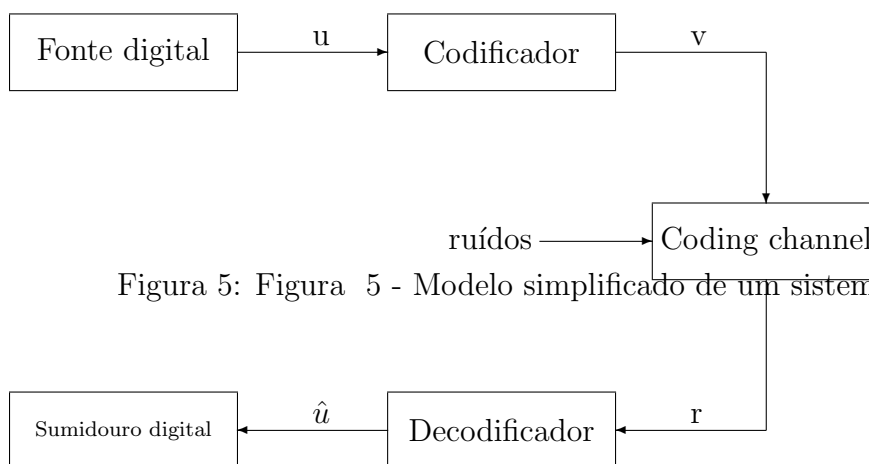


Figura 5: Figura 5 - Modelo simplificado de um sistema de codificação [23]

Em [57] são apresentados o processo de codificação e decodificação de uma mensagem

utilizando o teorema de Shannon.

3.4 Técnicas para detecção e correção de erros

Existem dois métodos básicos para tratar erros em comunicação e ambos envolvem a codificação de mensagens. A diferença está em como esses códigos são utilizados. Em um *repeat request system*, os códigos são utilizados para detectar erros e se existirem erros, é feito um pedido de retransmissão. Com *forward errors correction*, os códigos são usados para detectar e corrigir erros.

Os erros de um canal com ruídos podem ser corrigidos por um das seguintes estratégias: *Automatic Repeat reQuest* e *Forward corrEction Code* [31].

3.4.1 *Automatic Repeat reQuest* - ARQ

ARQ utiliza redundância para detectar erros em mensagens e, após a detecção, o destinatário solicita uma repetição da transmissão. Um caminho de retorno é necessário. São sistemas de *two-way transmission*. Exemplos de sistemas que utilizam ARQ são linhas telefônicas e alguns sistemas de satélite [23].

Se existe um grande atraso de propagação, uma grande distância entre emissor e destinatário, este método pode ser muito ineficiente. Podem existir casos em que a retransmissão não é possível, quando não existe *backup*.

Técnicas ARQ incluem repetição seletiva, *Go-back N* e sistemas de reconhecimento positivo ou negativo [58].

3.4.2 *Forward Correction Code* - FEC

O diagrama da figura 4 representa um *one-way system*. A transmissão ou gravação é restrita a uma direção: da fonte para o sumidouro (destino). Sistemas de armazenamento de fita magnética e sistemas da *NASA's Deep Space Network* utilizam técnicas FEC [23].

O destinatário corrige a mensagem recebida através de *Error correcting-codes*. Este procedimento geralmente é chamado de correção de erros de repasse e pode ser implementado em *hardware* de propósito especial.

FEC utiliza redundância, assim o decodificador pode corrigir os erros de mensagens no destinatário. Não é necessário um caminho de retorno.

Uma analogia pode ser feita com uma pessoa que fala devagar e repetidamente, em uma linha telefônica com ruídos, acrescentando mais redundância para o ouvinte entender a mensagem correta.

3.4.3 Como FEC funcionam

Os tipos de códigos de canal podem ser agrupados em duas grandes categorias: códigos de bloco e códigos convolucionais.

Técnicas FEC incluem códigos convolucionais, que são códigos com memória e códigos de blocos (*block codes*, que são códigos sem memória [36]. O alfabeto dos códigos explicados aqui consistem de dois elementos: 0 e 1. O conjunto $\{0, 1\}$ é denominado K .

Códigos de Blocos

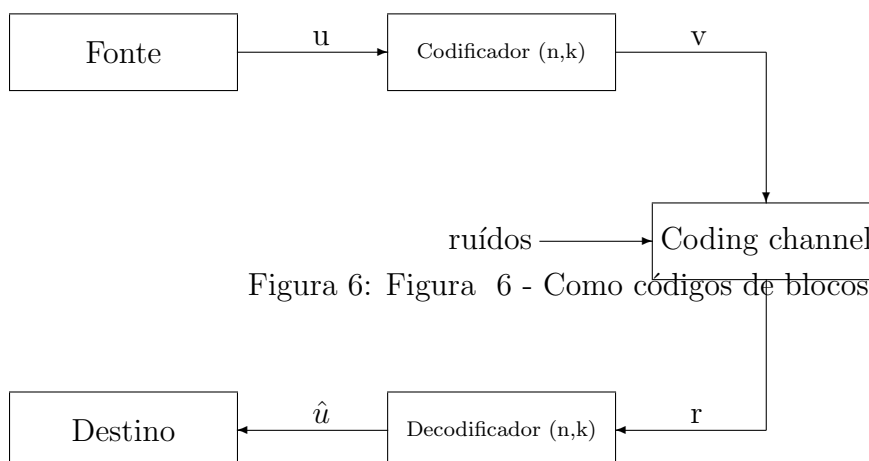


Figura 6: Figura 6 - Como códigos de blocos funcionam

Na figura 6, vemos um sistema que utiliza código de blocos. A fonte envia uma sequência de dados para o codificador. O codificador divide a sequência de dados em m blocos de

k bits cada chamados mensagens. Uma mensagem é representada por uma k -tupla binária $u = u_1, u_2, \dots, u_k$. Existem 2^k diferentes mensagens. O codificador insere bits redundantes (ou de paridade) para cada mensagem u , gerando uma sequência de saída de n bits chamada *code-word* ou palavra código representada por uma n -tupla de símbolos discretos $v = v_1, v_2, \dots, v_n$. Fazendo uma correspondência entre as 2^k mensagens, existem 2^k diferentes palavras código para a saída do codificador. Esse conjunto de 2^k palavras código de tamanho n é chamado de códigos de blocos (n, k) .

Códigos de blocos são identificados pela notação (n, k) , de acordo com o número de bits de saída n e o número de bits k de cada um dos blocos de entrada. Os $n - k$ bits são os bits redundantes que provem à codificação a capacidade de tratar os ruídos do canal.

Todas as palavras código de um código de blocos tem tamanho fixo, que é um certo número de blocos de k bits.

A geração de uma palavra código depende apenas de um cálculo algébrico entre os k bits, portanto, um codificador pode ser implementado como um circuito lógico combinacional.

O codificador executa o mapeamento: $T : U \rightarrow V$ onde U é um conjunto de palavras de dados de tamanho k e V é um conjunto de palavras código de tamanho n onde $n > k$. Cada uma das 2^k palavras de dados é mapeada para uma única palavra código.

A taxa de codificação e o *overhead* de armazenamento são calculados a partir de m blocos originais [4] [15]. São gerados n símbolos pelo algoritmo de codificação. $R = \frac{k}{n}$ é a taxa de codificação que pode ser interpretada como o número de bits de informação por palavra código transmitida e $O = \frac{1}{R}$ é o *overhead* de armazenamento.

Se $k \leq n$, mais bits redundantes podem ser adicionados, com aumento do tamanho da palavra código, mantendo $R = \frac{k}{n}$ constante. Como escolher este número $n - k$ de bits redundantes para obter transmissão confiável em cima de um canal com ruídos é o problema principal do projeto do codificador.

No destino, o decodificador extrai a sequência original de dados. O algoritmo de decodificação é mais complexo que o da codificação.

Outra métrica utilizada é a redundância que pode ser definida por $\frac{(n-k)}{n}$. A codificação que introduz uma alta redundância, isto é, $(n-k) \gg$ ou $(\frac{k}{n}) \ll$, transmite menos informação por palavra codificada. A codificação que introduz menos redundância transmite mais informação por palavra codificada.

A alta redundância é vantajosa porque reduz a possibilidade de todos os dados serem enviados em uma única transmissão. A desvantagem da redundância é que a adição de *bits* pode exigir uma largura de banda transmissão maior ou aumentar o atraso das mensagens (ou ambos).

Se o sistema não exige uma transferência de dados em tempo-real, então o atraso das mensagens é um usual *trade-off*.

A figura 7 apresenta uma classificação para códigos de blocos.

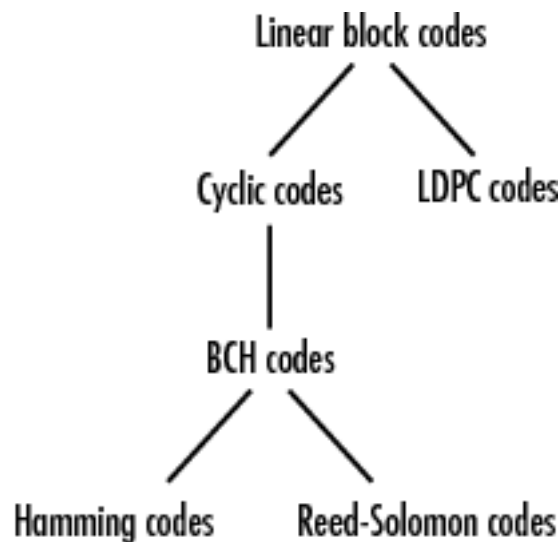


Figura 7: Uma classificação para códigos de blocos [34]

Um código C é linear se v e w são palavras código distintas de um código C , então $v+w$ é também uma palavra código de C . Um código linear contém a palavra código zero, pois $v + v = 0$. Operação simples de decodificação, pouca memória e métodos simples para

determinação de padrões de erros são algumas das vantagens de códigos lineares.

Um código C é chamado de cíclico se o deslocamento cíclico (*shift*) de qualquer palavra código gera uma nova palavra código. Por exemplo, $C1 = \{000, 110, 101, 011\}$ é um código cíclico. $C2 = \{000, 100, 011, 111\}$ não é um código cíclico.

3.5 Códigos Convolucionais

P. Elias introduziu códigos convolucionais em 1955. Um código convolucional é um dispositivo com memória. Apesar de aceitar uma mensagem de entrada de tamanho fixo e produzir uma saída codificada, seus cálculos não dependem somente da entrada atual, mas também das entradas e saídas anteriores.

Um codificador para um código convolucional também aceita blocos de k bits da sequência de dados u e gera uma sequência de saída v de n bits chamada *codeword* ou palavra código. Cada bloco da palavra código não depende apenas dos k bits do bloco da sequência de dados correspondente, mas também de M blocos anteriores. Dizemos que o codificador tem memória de ordem M , onde M é o número de registradores de memória. Esse conjunto de blocos de k bits, o codificador das palavras código de tamanho n e de memória de ordem M é chamado de (n, k, M) código convolucional. $R = \frac{k}{n}$ é a taxa de codificação. Como o codificador tem memória, ele pode ser implementado como circuito lógico sequencial [23].

Em um código convolucional, os $n-k$ bits redundantes, que provêm à codificação a capacidade de tratar os ruídos do canal, podem ser adicionados quando $k < n$. Para uma mesma taxa de codificação R , pode-se adicionar redundância, aumentando a ordem da memória M do codificador. Como usar a memória para obter uma transmissão confiável sob um canal com ruído é o principal problema do projeto do codificador.

Códigos convolucionais podem ser usados para melhorar o desempenho da comunicação por rádio e satélites. Códigos convolucionais são utilizados nas tecnologias CDMA (*Code division multiple access*) e GSM (*Global System for Mobile Communications*) para telefones celulares, *dial-up modems*, satélites, *NASA's Deep Space Network* deep-space communica-

tions e na rede WLAN Wi-Fi IEEE 802.11.

Existem vários algoritmos de codificação por apagamento [1, 3]. Alguns dos mais utilizados são códigos Reed-Solomon (RS), códigos Low-Density Parity-Check (LDPC) e códigos Tornado [30] [4].

Segundo [33], para sistemas de armazenamento, a codificação por apagamento baseada em operações simples, tais como XOR RAID e códigos Tornado, são preferíveis. Apesar de que um mecanismo externo deva ser utilizado para detectar erros, as operações de XOR podem ser realizadas rapidamente e resultar em alto *throughput* das operações de codificação e decodificação.

3.6 Características de Códigos de Blocos

A distância de Hamming (dH) entre duas palavras código v_i e v_j é o número de *bits* que são diferentes nessas duas palavras.

Segundo [22], as principais características de códigos de blocos são:

taxa de codificação $R = \frac{k}{n}$ é a medida da eficiência do código, pois é o quociente do número de *bits* da palavra de dados sobre o número de *bits* total da palavra transmitida

distância mínima (d_{min}) é a menor distância de Hamming entre duas quaisquer palavras do código; ela depende do número de *bits* redundantes $q = n - k$, tal que ($d_{min} \leq q + 1$)

capacidade de detecção detecta até l erros, onde $l \leq d_{min} - 1$

capacidade de correção corrige os erros até t erros, onde $t \leq \lfloor \frac{d_{min}-1}{2} \rfloor$

capacidade de detecção e correção detecta até l erros e corrige os erros até t erros, onde

$$d_{min} \geq l + t + 1 \text{ e } l > t$$

Um código com $d_{min} = 1$ não tem capacidade de detectar erros.

3.7 Códigos de Blocos Lineares

3.7.1 Códigos Reed-Solomon

São códigos de blocos, lineares e cíclicos.

Definition. A Reed-Solomon (or RS) code over $GF(q)$ is a BCH (BOSE-CHAUDHURI-HOCQUENGHEM) code of length $N = q - 1$. Of course q is never 2. Thus the length is the number of nonzero elements in the ground field. We shall use N , K and D to denote the length, dimension, and minimum distance (using capital letters to distinguish them from the parameters of the binary codes which will be constructed later). RS codes are important in concatenated codes and burst correction.

Em [49], o autor apresenta uma especificação completa do problema e do algoritmo da codificação e detalhes de sua implementação. O modelo estudado é formado por n dispositivos de armazenamento D_1, D_2, \dots, D_n (*data devices*) e outros m dispositivos de armazenamento C_1, C_2, \dots, C_m (*checksum devices*). O conteúdo de cada um dos m *checksum devices* é calculado a partir do conteúdo dos n *data devices*. O objetivo do cálculo dos C_i para $1 \leq i \leq m$ é tal que para quaisquer m dispositivos que falhem dos $D_1, D_2, \dots, D_n, C_1, C_2, \dots, C_m$, o conteúdo dos dispositivos que falharam possa ser reconstituído a partir dos dispositivos que não falharam.

Segundo [28], códigos RS são particularmente úteis para correção de erros em rajada (seqüência símbolos consecutivos, nenhum desses recebidos corretamente, chamados *burst errors*). Também podem ser usados eficientemente em canais onde o conjunto de símbolos de entrada é consideravelmente grande.

Uma implementação de biblioteca em C/C++ para o algoritmo RS foi apresentada em [10].

O sistema de armazenamento OceanStore [3] e o protocolo BitTorrent (aplicação da camada de rede da internet) usam uma codificação RS.

3.7.2 Códigos RAID

Redundant Arrays of Inexpensive [Independent] Disks (RAID) é uma classe de códigos RS. RAID é um método para prover tolerância a falhas ou alto desempenho em sistemas de armazenagem utilizando para isso uma codificação de correção de erros ou paridade. RAID foi introduzido por D. A. Patterson na Universidade da Califórnia, Berkeley (UC Berkeley) em 1988 [20].

São conceitos básicos [11]:

data striping é uma técnica para segmentar dados sequenciais, como um arquivo, de maneira que o acesso a segmentos sequenciais seja feito por diferentes dispositivos de armazenamento. Esta técnica é útil quando se quer processar mais rapidamente os pedidos de acesso a dados que os dispositivos de armazenamento permitem. Diferentes segmentos de dados são mantidos em diferentes dispositivos de armazenamento. A falha de um dos dispositivos torna toda a sequência de dados indisponível. Essa desvantagem é superada pelo armazenamento de informações redundantes (custo de armazenamento extra), como a paridade, com o objetivo de correção de erros. As configurações de RAID que utilizam paridade são RAID-2, RAID-3, RAID-4, RAID-5 e RAID-6 [38].

stripe são segmentos consecutivos ou faixas que são escritos sequencialmente através de cada um dos discos de um *array* ou conjunto. Cada segmento tem um tamanho definido em blocos.

RAID 2

Esta configuração divide os dados a nível de *bit* e usa códigos Hamming para correção de erros. Por exemplo, o código Hamming(7,4) (quatro *bits* de dados e tres *bits* de paridade) permite usar 7 discos em RAID 2, sendo 4 usados para armazenar dados e 3 usados

para correção de erros. Esta codificação tornou-se padrão para *hard drives* e tornou-se desnecessária, assim deixou de ser vantajosa.

RAID 3

Esta configuração divide os dados a nível de *byte* com um disco apenas para paridade. Isto requer que todos os discos operem em *lockstep* (rotação de todos os discos em sincronismo). Assim como RAID-2, tornou-se obsoleta.

RAID 4

Esta configuração divide os dados a nível de bloco com um disco apenas para paridade. Se o controlador de disco permitir, um conjunto RAID 4 pode atender várias solicitações de leitura ao mesmo tempo. Todos os *bits* de paridade estão em um único disco, o que pode se tornar um gargalo. RAID-5 substituiu esta configuração.

RAID 5

Esta configuração divide os dados a nível de bloco com um único bloco de paridade por *stripe* e os blocos de paridade ficam distribuídos por todos os discos. Esta configuração privilegia a leitura. Uma síndrome é computada para permitir a perda de uma unidade. Essa síndrome P pode ser um simples XOR de dados pelos *stripes*.

RAID 6

Esta configuração divide os dados a nível de bloco com dois blocos de paridade por *stripe* e os blocos de paridade distribuídos por todos os discos. Duas síndromes diferentes precisam ser computadas para permitir a perda de quaisquer duas unidades. Uma delas, P pode ser

um simples XOR de dados pelos *stripes*, como em RAID 5. A outra, Q pode ser um XOR de um *linear feedback shift register* de cada *stripe*.

Passo 0: nó 0 cria palavra código em blocos $b_1, b_2, b_3, \dots, b_k$

Passo 1: nó 0 cria b_1 e o envia para os nó 1 e nó 2

Passo 2: nó 0 cria b_2 e o envia para os nó 1 e nó 3

Passo 3: nó 0 cria b_3 e o envia para os nó 1 e nó 4

Passo 4: nó 0 e nó 1 codificam b_1, b_2 e b_3 e criam o bloco de paridade $p_1 = f_1(b_1, b_2, b_3)$ e $p_2 = f_2(b_1, b_2, b_3)$

3.7.3 Códigos Low-Density Parity-Check e Tornado

São códigos de blocos, lineares e acíclicos.

Código Low-Density Parity-Check (LDPC) é uma codificação baseada em grafos regulares [44]. Códigos LDPC são conhecidos também como códigos Gallager [40]. Uma aplicação dessa codificação é a rede *wireless* WMAN WiMAX (IEEE 802.16e *standard for microwave communications*) para internet móvel [41].

Códigos Tornado são uma classe de códigos LDPC [33]. Segundo [3], são mais rápidos para codificar e decodificar e necessitam de um pouco mais de m fragmentos para reconstruir a informação.

Em [9], os autores apresentam códigos Tornado baseados em grafos irregulares. Segundo [24], as implicações práticas desses códigos ainda não foram bem estudadas.

4 Discussão sobre esquemas adequados de redundância de dados

Pode-se encontrar na literatura um número significativo de projetos que propõem sistemas de armazenamento distribuído, sistemas de arquivos distribuídos, ou sistemas de backup. Apesar da disso, nenhum esquema de redundância de dados tem sido amplamente aceito para esses sistemas e nenhuma regra fácil foi criada para se encontrar um esquema adequado de redundância de dados.

Os autores em [2] estudaram redundância de dados em sistemas *peer-to-peer* para *backup* e propuseram um esquema híbrido para implementar redundância (replicação e codificação) de dados. Em [26], foi proposto um sistema de armazenamento de dados baseado em discos que usa dois níveis de codificação. Nesses textos, encontramos uma comparação entre alguns sistemas de armazenamento, avaliando-se algumas de suas características. Podemos observar que os tipos de esquema de redundância de dados são replicação, codificação por apagamento e híbrido, sendo a replicação, a estratégia mais utilizada nos sistemas comparados por [2] e a codificação por apagamento, para os sistemas comparados por [26].

Redundância de dados é necessária para prevenir perda de dados, mas não é suficiente. A avaliação de esquemas de redundância é muitas vezes baseada na suposição de que as réplicas falham de forma independente. Na prática, as falhas não são tão independentes, segundo [25, 35]. Esse trabalho não tratará a independência das réplicas.

Cada esquema de redundância estabelece (i) como criar os dados redundantes e (ii) como reconstruir os dados quando houver falha. Essas duas operações geram custos que diferem de um esquema para outro. Esse trabalho comentará os mais amplamente usados esquemas de redundância: replicação e codificação por apagamento, que chamaremos de codificação.

4.1 Esquemas de redundância de dados para sistema de armazenamento

Esquemas de redundância de dados são utilizados em sistemas de armazenamento (exemplo: sistemas RAID) para prover disponibilidade, tolerância a falhas e durabilidade de dados e em sistemas de comunicação (exemplo, sistemas *peer-to-peer*) para prover uma entrega confiável e segura de dados.

4.1.1 Replicação

Replicação é o esquema de redundância mais simples. A maioria dos sistemas, que utiliza redundância de dados, é baseada em replicação, mas esse esquema consome mais espaço que a codificação por apagamento, pois uma cópia completa de cada arquivo é armazenada em cada um dos servidores de dados.

A principal desvantagem da replicação é que ela requer um grande *overhead* de armazenamento para pouco ganho em disponibilidade e tolerância a falhas. Garantir que os dados permaneçam disponíveis quando todos os n dispositivos falham exige que, pelo menos, $n + 1$ cópias existam [33]. Por exemplo, no artigo sobre o sistema Glacier [37], os autores argumentam que o armazenamento aumenta de 11 vezes a quantidade de dados armazenados utilizando apenas replicação para conseguir 0.999999 (*six nines*) de confiabilidade num cenário com 60% de indisponibilidade dos *peers*.

Os autores em [43] afirmam que dados replicados permitem leituras de baixa latência, porque há muitas opções para a seleção de servidores, enquanto que dados codificados reduzem o consumo de largura de banda para escritas, em detrimento do aumento da latência de leituras.

A replicação é usada no Google File System [48] (GFS), no Hadoop Distributed File System [17] (HDFS) e no Kosmos distributed file system [54] (KFS), sistemas de arquivos distribuídos que apresentam características semelhantes. Um *cluster* do GFS ou do HDFS

ou do KFS é formado por um único servidor, master (GFS) ou namenode (HDFS) ou Meta server (KFS), que mantém os metadados e muitos servidores de dados, os chunkservers (GFS e KFS) ou os datanodes (HDFS) e é acessado por vários clientes. Os arquivos de dados são armazenados nos chunkservers (GFS e KFS) ou datanodes (HDFS) e são particionados em blocos de igual tamanho. GFS, HDFS e KFS foram projetados para aplicações que processam grande volume de dados.

Seus projetos consideram *clusters* de (*commodity hardware*), uma versão do *kernel* linux como sistema operacional para as máquinas e uma arquitetura de rede com dois níveis: vários *racks* interligados por um comutador e cada *rack* é formado por várias máquinas e seus discos, estes também interligados por um comutador. A estratégia de inserção de dados cria réplicas em *racks* distintos do *rack* onde está a 1ª réplica, assim, falhas que comprometam um *rack* não provocam a indisponibilidade de dados. Os arquivos de dados são alterados por concatenações ao invés de sobrescrever dados existentes. Após a criação, os arquivos de dados são usados apenas para leitura e esta leitura ocorre sequencialmente. O KFS permite escrever em posições randômicas nos arquivos. As APIs do cliente fornecidas pelo GFS, pelo HDFS e pelo KFS suportam operações de criação, leitura, escrita, remoção de arquivos, mas não implementam a interface POSIX.

O GFS está disponível para linux sob uma licença de software proprietário. O HDFS ¹ e o KFS ² estão disponíveis para linux sob uma licença Apache.

O Farsite, que utiliza apenas replicação, é um sistema de arquivos distribuídos, particionados em namespaces, explorando os desktops presentes dentro da Microsoft, sem servidor mestre, disponível para Windows sob uma licença de software proprietário. A escolha da replicação foi, pelos autores, considerada uma opção mais simples para disponibilidade, já que a codificação poderia significar latência adicional nas leituras dos arquivos. Ainda segundo os autores, estudos com experimentos já mostraram que a codificação pode apresentar

¹<http://hadoop.apache.org/>

²<http://code.google.com/p/kosmosfs/>

um bom desempenho e seria possível, então, alterar no futuro o esquema de redundância do Farsite.

Big Table (construído sob o GFS) e Dynamo (construído para Amazon.com) são dois sistemas de armazenamento que gravam e recuperaram dados através de uma chave e executam em um *pool* compartilhado de máquinas, utilizam apenas replicação.

Ceph [7] é um sistema de arquivos *open source* que possui três principais componentes: um *cluster* de servidores de metadados (que gerencia o namespace, nomes de arquivos e diretórios), um *cluster* de OSDs (dispositivos de armazenamento de objetos) que armazenam dados e metadados e os clientes que utilizam uma interface do sistema de arquivos. O Ceph agrupa dados em PGs (grupos de colocação) e usa uma função *hash* para distribuir os PGs nos OSDs, cujo algoritmo CRUSH é $O(\log n)$ e usa uma árvore-B para indexar os PGs. Existe um módulo em desenvolvimento que permite usar o Ceph como armazenamento para uma instância do Hadoop. O Ceph utiliza apenas replicação, implementa parcialmente a interface POSIX e está disponível para linux sob LGPL ³.

Lustre ⁴ tem na sua arquitetura os metadata server (disponibiliza os metadados para clientes), o metadata target (um por sistema de arquivos, armazena os metadados), object storage servers (armazena os dados), object storage target (armazena os objetos que contém os arquivos de dados) e clientes.

Moosefs ⁵ também foi projetado com uma arquitetura que se assemelha a do GFS, HDFS e KFS: master server (que armazena os metadados), chunk servers (que armazenam os dados), metalogger server (podem substituir algumas funções do master server, se ele falhar) e clientes (que solicitam dados e se comunicam com o master server e o chunk servers).

Ambos, Lustre e Moosefs, usam replicação, implementam a interface POSIX e estão disponíveis para linux sob uma licença GPL.

Com exceção do Farsite, os sistemas de armazenamento apresentados consideram *clusters*

³<http://ceph.newdream.net/>

⁴[git://git.lustre.org/prime/lustre.git](http://git.lustre.org/prime/lustre.git)

⁵<http://www.moosefs.org/>

de (*commodity hardware*) e uma versão do *kernel* linux como sistema operacional para as máquinas.

4.1.2 Codificação por Apagamento

Uma vantagem de codificação por apagamento é um custo menor de armazenamento se comparado a replicação, no caso de grande volume de dados. Outra vantagem com relação a replicação foi comentada em [12]: para um mesmo espaço de armazenamento, o tempo médio entre falhas (*mean time to failure*) é maior.

HDFS, Total Recall e OceanStore usam codificação para reduzir o tamanho do armazenamento de dados e todos os outros sistemas usam codificação para prover disponibilidade e confiabilidade.

Os sistemas que foram comparados na tabela 1, estão disponíveis para uma versão de sistema operacional linux ou unix. Além da codificação, todos eles implementam replicação.

Vamos avaliar algumas das métricas utilizadas em literatura para comparar redundância de dados em sistemas de armazenamento: sobrecarga de armazenamento, disponibilidade dos *peers*, corrupção de um dado e operações de criação, leitura, atualização consistente e remoção de dados redundantes. Inicialmente vamos definir alguns conceitos adaptados de [2, 19, 5, 56] para esses dois esquemas de redundância.

4.2 Caracterização da replicação e da codificação

A confiabilidade de um esquema de redundância é medida pelo número de falhas simultâneas que ele pode tolerar sem comprometer a capacidade de reconstruir os dados originais. Assim esta propriedade pode ser expressada como a **probabilidade de perda de dados, dado que ocorreram l falhas**, $P(l)$.

Para avaliar o armazenamento, vamos definir **fator de redundância** B , uma razão entre tamanho dos dados originais mais a redundância $|dado + red|$ e o tamanho dos dados originais $|dado|$ e também vamos definir grau de reparação.

sistema	codificação	arquitetura do sistema	licença
HDFS [17]	RAID, Reed-Solomon, ...	shared-disk file system for cluster	Apache
Tahoe-LAFS [6] 6	RAID	peer-to-peer filesystem	GPL2
Pergamum [26]	XOR parity, Reed-Solomon	disk-based archival storage	*
Potshards [27]	RAID e Reed-Solomon	disk-based archival storage	*
RobuStore [21]	Luby Transform (LT) codes	disk-based archival storage	*
Glacier [37]	Reed-Solomon com matrizes Cauchy	peer-to-peer storage system	*
Total Recall [29]	Maymounkov's online codes	peer-to-peer storage system	*
FAB [50]	Reed-Solomon	distributed disk array	*
GPFS [46]	RAID	shared-disk file system for cluster	*
Oceanstore [3] ⁷	Tornado codes e Reed-Solomon codes	desktops e notebooks conectados a servidores geograficamente distribuídos	BSD
xFS [8] ⁸	RAID	serverless network file system	GPL
Swift [13]	RAID	desktops sob unix conectados a uma intranet	*

Tabela 1: Comparação entre sistemas de armazenamento de grande volume de dados que utilizam *commodity hardware*

onde:

* = não disponível

O **grau de reparação** mede o que deve ser feito após parte da redundância ser perdida. Para isso, é feita uma leitura de dados disponíveis para produzir novos. O custo dessa leitura em um sistema de armazenamento distribuído inclui volume do tráfego da rede, política de reparação, algoritmo de coordenação. Nesse estudo vamos apenas avaliar a contribuição do esquema de redundância: a quantidade de dados a serem lidos para que dados novos sejam criados para reparar o dado corrompido, definido por d .

Definimos p como a probabilidade de um *peer* estar disponível e $q = 1 - p$ como a probabilidade de um *peer* não estar disponível. Nesse estudo, assumimos que elas são iguais para todos os *peers*.

Vamos definir as operações de acesso a dados redundantes como uma tupla (r, w, a) , onde r é o número mínimo de *peers* disponíveis que armazenam dados redundantes, w é o número mínimo de *peers* disponíveis que armazenam dados redundantes que devem ser acessados para armazenar novos valores e a é número de *peers* (provavelmente outros) que devem estar disponíveis para completar a operação.

4.2.1 Replicação

Para as definições, n é o número de réplicas dos dados.

número de falhas suportadas $l = n - 1$

probabilidade de perda de dados, dado que ocorreram l falhas

$$P(l) = \begin{cases} 0, & \text{se } l < n \\ 1, & \text{se } l = n \end{cases}$$

fator de redundância

$$B = |dado + red| / |dado| = n$$

acesso	tupla (r, w, a)
criar	$(0, 0, n)$
apenas leitura	$(1, 0, 0)$
atualização consistente	$(1, n, 0)$
apagar	$(0, n, 0)$

Tabela 2: Operações sobre dados redundantes na replicação

grau de reparação

$$d = 1$$

disponibilidade de um dado

$$1 - q^n$$

corrupção de um dado

$$e = q^n$$

Comparando as tuplas $(1, 0, 0)$ e $(1, n, 0)$ da tabela 2, podemos observar que a disponibilidade da operação "atualização consistente" é menor que a disponibilidade da operação "apenas leitura", quando o número de réplicas $n > 1$ é aplicado.

4.2.2 Codificação por Apagamento

Na Codificação (m, k) , k é o número de blocos originais do dado, m é o número de blocos codificados e $m - k$ é o número de blocos adicionados pela codificação.

número de falhas suportadas $l = m - k$

probabilidade de perda de dados, dado que ocorreram l falhas

$$P(l) = \begin{cases} 0, & \text{se } l \leq m - k \\ 1, & \text{se } l > m - k \end{cases}$$

acesso	tupla (r, w, a)
criar	$(0, 0, n)$
apenas leitura	$(k, 0, 0)$
atualização consistente	$(k, m - k + 1, k - 1)$
apagar	$(0, m - k + 1, 0)$

Tabela 3: Operações sobre dados redundantes na codificação por apagamento

fator de redundância

$$B = m/k$$

grau de reparação

$$d = k$$

disponibilidade de um dado

$$B(k, m, p) = \sum_{i=k}^m \binom{m}{i} p^i q^{m-i}$$

corrupção de um dado

$$e = q^{m-k}$$

Por outro lado, comparando-se as tuplas $(k, 0, 0)$ e $(k, m - k + 1, k - 1)$ da tabela 3, podemos observar que existe um caso no qual as operações "apenas leitura" e "atualização consistente" podem ter a mesma disponibilidade. Isto ocorre quando $m = 2k - 1$ (assumindo-se também que número total de *peers* disponíveis é maior ou igual a m), obtendo-se a tupla $(k, k, k - 1)$.

4.3 Sobrecarga de armazenamento

Em [12, 43], os autores afirmam que a codificação obtém o mesmo nível de disponibilidade como a replicação, usando muito menos espaço de armazenamento.

No esquema da replicação, para um objetivo de probabilidade de indisponibilidade de um dado $e = 0.0001$, dado uma probabilidade de um *peer* estar indisponível $q = 0.05$, obtemos o valor de $n = 3$ réplicas.

$$\begin{aligned} e &= q^n \\ n &= \log e / \log q = \log 0.0001 / \log 0.05 = 3.074487147 \\ n &\approx 3 \end{aligned}$$

No esquema da codificação, para o mesmo objetivo, para as mesmas probabilidades e e q , obtemos um valor de 3 blocos de paridade para uma codificação $(2k - 1, k)$.

$$\begin{aligned} e &= q^{m-k}, \text{ substituindo } m = 2k - 1 \\ e &= q^{k-1} \\ k - 1 &= \log e / \log q = \log 0.0001 / \log 0.05 = 3.074487147 \\ k &\approx 4 \end{aligned}$$

Para uma mesma probabilidade de indisponibilidade de um dado $e = 0.0001$ e uma mesma probabilidade de um *peer* estar indisponível $q = 0.05$, obtemos o valor de $n = 3$ réplicas para a replicação e um valor de $k = 4$ blocos iniciais para a codificação $(2k-1,k)$. O fator de redundância é $B = 3$ para a replicação e $B = m/k = (2k - 1)/k = 7/4 = 1.75$ para a codificação.

No HDFS, a codificação RS implementada é $(16, 10)$, 10 blocos iniciais e 6 blocos de paridade. Logo, tolera até $l = 6$ falhas por *stripe* de 10 blocos ($k = 10$) com sobrecarga de $B = 1.6$, como a replicação de $n = 6$ réplicas, que tem sobrecarga de $B = 6$. A codificação RAID implementada é RAID-5, portanto, é $(12, 10)$, 10 blocos iniciais e 2 blocos de paridade. Logo, tolera $l = 2$ falhas por *stripe* de 10 blocos ($k = 10$) com sobrecarga de $B = 1.2$ como a replicação de $n = 2$ réplicas, que tem sobrecarga de $B = 2$.

Na figura 8, podemos observar o gráfico ⁹ das funções $y = 2x$, $y = 3x$ e $y = (2x - 1)/x$,

⁹O gráfico foi gerado em formato PNG pelo Gnuplot.

para $x > 0$, representando a sobrecarga de armazenamento na replicação $2n$ e na $3n$ e na codificação $(2k - 1, k)$, respectivamente. O eixo x representa o tamanho de um dado e o eixo y , a sobrecarga de armazenamento desse dado.

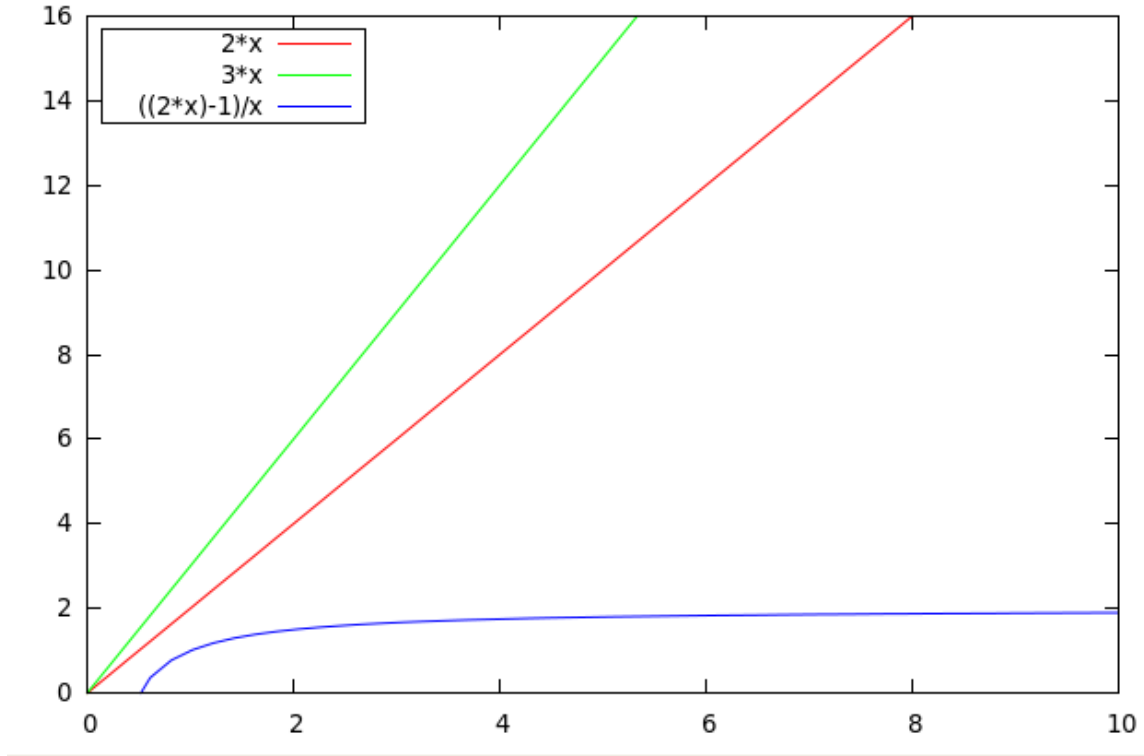


Figura 8: Sobrecarga de armazenamento para replicação $3n$, $2n$ e codificação $(2k - 1, k)$ representadas, respectivamente, pelas funções $y = 3x$, $y = 2x$ e $y = (2x - 1)/x$, para $x > 0$

Neste estudo, concluímos que, apenas avaliando a contribuição do esquema de redundância, a codificação por apagamento acarreta uma sobrecarga de armazenamento menor que a replicação para o mesmo número de falhas toleradas.

4.4 Disponibilidade dos peers

4.5 Leitura ou Atualização dos dados redundantes

Em [19], os autores concluem que:

- o acesso somente de leitura pode ser suportado tanto por replicação de dados simples como por codificação
- para privilegiar atualização consistente, uma codificação de alta disponibilidade é necessária que se caracteriza por fracionamento do original dados em pedaços k e adicionando exatamente $k - 1$ pedaços
- se ler e a disponibilidade de atualização consistente são de igual importância, isso requer codificação $(2k - 1, k)$

Podemos também concluir que a replicação é um caso onde $k = 1$ e $l = m - k = n - 1$, portanto $m = n$.

Os autores também concluíram que usar apenas a replicação tem sentido apenas em poucos casos.

5 Hadoop

Atualmente, o Google é uma empresa de consulta e publicidade e é capaz de fornecer os seus serviços devido a investimentos em armazenamento distribuído em larga escala e a capacidade de processamento, estes desenvolvidos *in-house*.

Essa capacidade é fornecida por um grande número de PCs, pelo Google File System (GFS), um sistema de arquivos redundantes em *cluster*, pelo sistema operacional GNU/Linux e pelo MapReduce, um *middleware* de processamento paralelo de dados.

Em 2004, um artigo [47], que foi publicado por profissionais da Google, propôs o MapReduce. Em 2006, estes profissionais, juntamente com Doug Cutting do Yahoo!, formaram um sub-projeto do Apache Lucene¹⁰ que foi chamado Hadoop¹¹.

Mais recentemente, o projeto Apache Hadoop tem desenvolvido uma reimplementação de partes do GFS e MapReduce e muitos grupos da comunidade de software livre posteriormente abraçaram essa tecnologia, permitindo-lhes fazer coisas que eles não poderiam fazer em máquinas individuais. O Hadoop está disponível em código fonte sob licenciamento Apache *license* (compatível com GPL).

O Hadoop é um *framework* para executar aplicações em armazenamento distribuído de grande volume de dados que pode ser construído com *commodity hardware*, que é facilmente acessível e disponível. O Hadoop não é um *framework* canônico. Ele foi projetado para aplicações que atualizam dados da seguinte forma: uma escrita e muitas leituras, através de acessos por *batch*, com tamanho da ordem de petabytes, organizados de forma não estruturada, com esquema dinâmico e integridade baixa. Uma lista de aplicações e organizações que usam o Hadoop pode ser encontrada em [61].

Em poucas palavras, o Hadoop disponibiliza um armazenamento compartilhado (HDFS) e um sistema de análise (MapReduce) que compõem o seu *kernel*.

5.1 MapReduce

O MapReduce utiliza algoritmos de ordenação para reconstruir sua base de dados. Um bom uso para o MapReduce são aplicações cujos dados são escritos uma vez e lidos muitas vezes. São dados não estruturados como texto ou imagens. O MapReduce tenta colocar esses dados no nó onde são feitas as computações, desta forma, o acesso aos dados é rápido, pois é local [55].

O MapReduce pode resolver problemas genéricos, cujos dados podem ser divididos em

¹⁰<http://www.apache.org>

¹¹<http://hadoop.apache.org/>

matrizes de dados, para cada matriz a mesma computação necessária (sub-problema) e não existe necessidade de comunicação entre as tarefas (sub-problemas). A execução de um típico *job* do MapReduce pode ser assim descrita:

- Iteração sobre um número grande de registros
- Map extrai algo de cada registro (chave, valor)
- Rearranjo (*shuffle*) e ordenação de resultados intermediários por (chave, valor)
- Reduce agrega os resultados intermediários
- Geração da saída

Um programas para execução no HDFS/MapReduce que podem ser escritos em várias linguagens como Java, Ruby, Python e C++.

5.2 Arquitetura do Hadoop *Distributed File System*

Um *cluster* do HDFS é composto por um único NameNode, um servidor-mestre que gerencia o sistema de arquivos e controla o acesso aos arquivos de clientes. Há uma série de DataNodes, geralmente um por nó do *cluster*, que gerenciam o armazenamento anexado ao nó em que são executados. A Figura 10 mostra o NameNode e os DataNodes.

Uma típica arquitetura de rede em dois níveis para um *cluster* Hadoop é construída por vários *racks* interligados por um comutador como mostra a Figura 9. Cada *rack* por sua vez é formado por vários nós (máquinas) e seus discos, estes também interligados por um comutador.

O NameNode executa operações no sistema de arquivos, como *open*, *close*, *rename* de arquivos e de diretórios.

HDFS disponibiliza espaço para sistema de arquivos e permite que os dados do usuário sejam armazenados em arquivos. Internamente, um arquivo é dividido em um ou mais

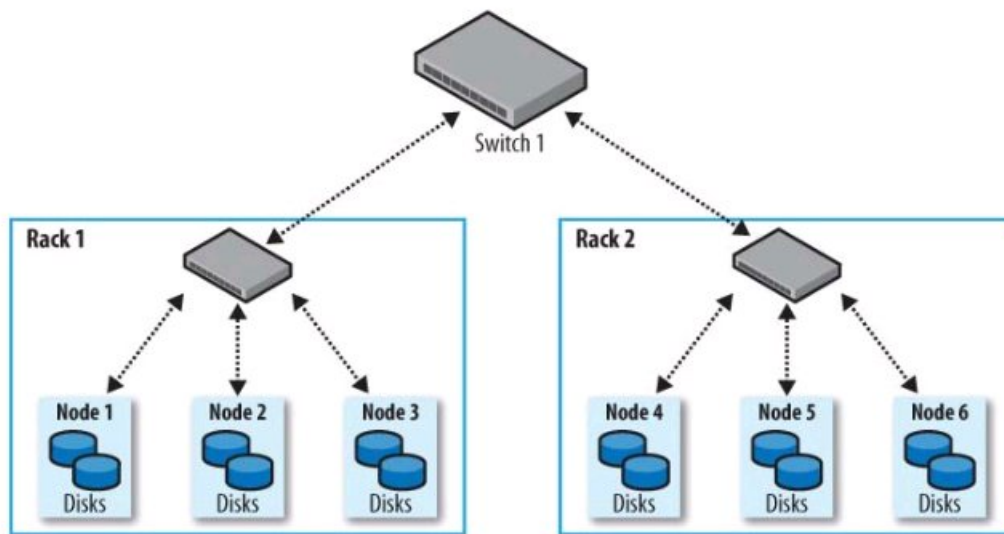


Figura 9: Arquitetura de rede em dois níveis para um cluster Hadoop [17]

blocos e esses blocos são armazenados em um conjunto de DataNodes. A Figura 11 mostra DataNodes e seus blocos. O tamanho *default* de cada bloco é 64MB.

Os DataNodes respondem aos pedidos de leitura e escrita de clientes do sistema de arquivos e também executam a criação, eliminação e replicação de blocos sob instrução do NameNode. O número de réplicas é geralmente 3. A 1ª réplica fica local, no mesmo nó do código do cliente. A 2ª réplica fica em um nó em outro *rack* e a 3ª réplica fica nesse último *rack* em outro nó. As 2ª e 3ª réplicas não são locais ao bloco replicado.

O NameNode e DataNode são partes do *software* projetado para rodar em *commodity hardware*. Essas máquinas normalmente executam um sistema operacional GNU/Linux.

HDFS é construído usando a linguagem Java. Qualquer máquina que suporte Java pode executar o NameNode ou o DataNode [17].

Os protocolos do HDFS usam o protocolo TCP/IP. O cliente fala o protocolo ClientProtocol com o NameNode através de uma porta. Os DataNodes falam o protocolo DataNodeProtocol com o NameNode. Esses protocolos executam uma *Remote Procedure Call* (RPC). O NameNode não inicia chamadas RPCs. Ele responde a chamadas RPCs feitas pelo DataN-

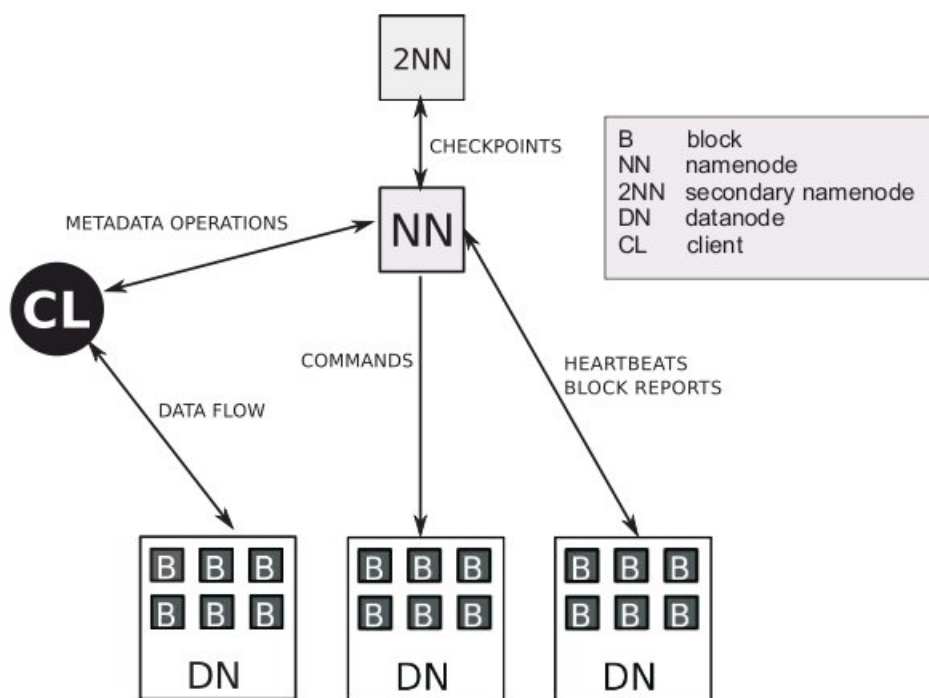


Figura 10: Arquitetura do HDFS [45]

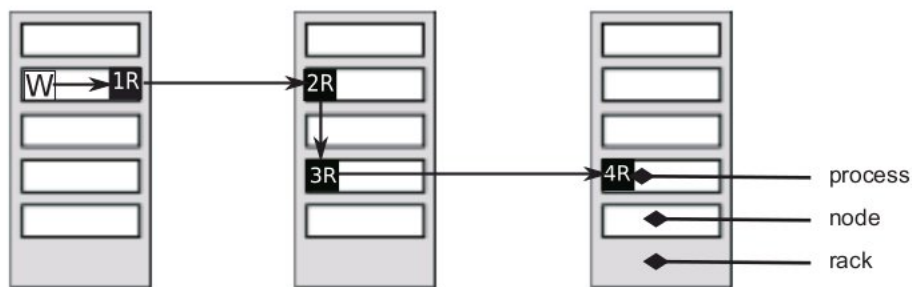


Figura 11: Arquitetura do HDFS - Datanodes e Blocos [55]

odes e pelos clientes.

5.3 Codificação por apagamento

Existe uma nova característica proposta em 2009 para implementação de uma camada de codificação por apagamento no Hadoop utilizando RAID [16] e uma mais recente utilizando códigos RS [18].

A versão atual do Hadoop utiliza apenas a técnica de replicação [55] para obter disponibilidade e confiabilidade de dados. A inclusão da codificação por apagamento será feita com o objetivo de reduzir o tamanho do armazenamento do HDFS.

Referências

- [1] Byers, J. W. Luby, M. Mitzenmacher, M. Rege, A. A digital fountain approach to reliable distribution of bulk data. *SIGCOMM Computer Communication Rev.*, 28(4):56–67, 1998.
- [2] Duminoco, A. *Data Redundancy and Maintenance for Peer-to-Peer File Backup Systems*. PhD thesis, École Doctorale d'Informatique, Télécommunications et Électronique de Paris, Paris, France, October 2009.

- [3] Kubiawicz, J. Bindel, D. Chen, Y. Czerwinski, S. Eaton, P. Geels, D. Gummadi, R. Rhea, S. Weatherspoon, H. Weimer, W. Wells, C. Zhao, B. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS '00: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 190–201, New York, NY, USA, 2000. ACM.
- [4] Oliveira, C. T. Moreira, M. D. D. Rubinstein, M. G. Costa, L. H. M. K. Duarte, O. C. M. B. Mc05: Redes tolerantes a atrasos e desconexões. In *Anais do 25o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, Belém, Pará, Brasil, May 2007.
- [5] Rodrigues, R. Liskov, B. High availability in dhds: Erasure coding vs. replication. In *Peer-to-Peer Systems IV*, pages 226–239. LNCS, 2005.
- [6] Wilcox-O’Hearn, Z. Warner, B. Tahoe: the least-authority filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, StorageSS ’08, pages 21–26, New York, NY, USA, 2008. ACM.
- [7] Weil, S. A. Brandt, S. A. Miller, E. L. Long, D. D. E. Maltzahn, C. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI ’06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.
- [8] Anderson, T. Dahlin, M. Neefe, J. Roselli, D. Wang, R. Patterson, D. Serverless network file systems. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1998.
- [9] Luby, M. Mitzenmacher, M. Shokrollah, A. Spielman, D. Analysis of low density codes and improved designs using irregular graphs. In *STOC ’98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 249–258, New York, NY, USA, 1998. ACM.
- [10] Plank, J. S. Simmerman, S. Schuman, C. D. Jerasure: A library in c/c++ facilitating erasure coding for storage applications - version 1.2. Technical Report CS-08-627, De-

- partment of Electrical Engineering and Computer Science, University of Tennessee, August 2007. URL=<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.140.3116>.
- [11] Vadala, D. *Managing RAID on Linux*, chapter 1, pages 1–10. O’Reilly, Sebastopol, CA, USA, 2002.
 - [12] Weatherspoon, H. Kubiawicz, H. J. D. Erasure coding vs. replication: A quantitative comparison. In *IPTPS ’01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 328–338, London, UK, 2002. Springer-Verlag.
 - [13] Cabrera, L. Long, D. D. E. Swift: Using distributed disk striping to provide high i/o data rates. Technical report, University of California at Santa Cruz, Santa Cruz, CA, USA, 1991.
 - [14] Shannon, C. E. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 623–656, 1948.
 - [15] Alan Sussman et al. lectures of peer-to-peer and grid computing course. URL=<http://www.cs.umd.edu/class/spring2007/cmsc818s/Lectures/lectures.htm>. Acessado em 03 de maio de 2010.
 - [16] Apache Software Foundation. Hdfs-503 - implement erasure coding as a layer on hdfs. URL=<https://issues.apache.org/jira/browse/HDFS-503>. Acessado em 08 de maio de 2010.
 - [17] Apache Software Foundation. Hdfs architecture. URL=http://hadoop.apache.org/common/docs/current/hdfs_design.html. Acessado em 08 de maio de 2010.
 - [18] Apache Software Foundation. Mapreduce-1969 - allow raid to use reed-solomon erasure codes. URL=<https://issues.apache.org/jira/browse/MAPREDUCE-1969>. Acessado em 20 de agosto de 2010.

- [19] Chiola, G. An empirical study of data redundancy for high availability in large overlay networks. In *Proceedings of the Second International Workshop on Hot Topics in Peer-to-Peer Systems*, pages 43–51, Washington, DC, USA, 2005. IEEE Computer Society.
- [20] Patterson, D. A. Gibson, G. Katz, R. H. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, New York, NY, USA, 1988. ACM.
- [21] Xia, H. *Robustore: a distributed storage architecture with robust and high performance*. PhD thesis, University of California at San Diego, La Jolla, CA, USA, 2006. AAI3225997.
- [22] Ferreira, A. J. material de apoio - códigos detectores e correctores de erros. URL=<http://www.cc.isel.pt/paginaspessoais/ArturFerreira.aspx>. Acessado em 24 de julho de 2010.
- [23] Lin, S. Costello, D. J. J. *Error Control Coding: Fundamentals and Applications*, chapter 1, 2, pages 1–50. Prentice-Hall Press, Englewood Cliffs, New Jersey, USA, 1983.
- [24] Plank, J. Cs560 - operating systems. URL=<http://www.cs.utk.edu/plank/plank/classes/cs560>. Acessado em 08 de maio de 2010.
- [25] Weatherspoon, H. Moscovitz, T. Kubiawicz, J. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. *21st IEEE Symposium on Reliable Distributed Systems*, 0:362–367, 2002.
- [26] Storer, M. W. Greenan, K. M. Miller, E. L. Voruganti, K. Pergamum: replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 1:1–1:16, Berkeley, CA, USA, 2008. USENIX Association.
- [27] Storer, M. W. Greenan, K. M. Miller, E. L. Voruganti, K. Potshards: a secure, recoverable, long-term archival storage system. *Trans. Storage*, 5:5:1–5:35, June 2009.

- [28] Almeida, G. M. Códigos corretores de erros em hardware para sistemas de telecomando e telemetria em aplicações espaciais. Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul - Faculdade de Informática, Porto Alegre, Brasil, março 2007.
- [29] Bhagwan, R. Tati, K. Cheng, Y. Savage, S. Voelker, G. M. Total recall: system support for automated availability management. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 25–25, Berkeley, CA, USA, 2004. USENIX Association.
- [30] Mitzenmacher, M. Digital fountains: A survey and look forward. In *ITW '04: Proceedings of the IEEE Information Theory Workshop*, pages 271–276, Piscataway, NJ, USA, 2004. IEEE Press.
- [31] Purser, M. *Introduction to Error-Correcting Codes*, chapter 1, pages 1–10. Artech House, Norwood, MA, USA, 1995.
- [32] Weber, J. M. *Bounds and Constructions for Binary Block Codes Correcting Asymmetric or Unidirectional Errors*. PhD thesis, Delft University of Technology, Delft, South Holland, Netherlands, 1985.
- [33] Woitaszek, M. *Tornado Codes for Archival Storage*. PhD thesis, Department of Computer Science of the University of Colorado, Boulder, Colorado, USA, dezembro 2007.
- [34] MathWorks. Error detection and correction. URL=<http://www.mathworks.com/access/helpdesk/help/toolbox/comm/ug/fp6603.html>. Acessado em 04 de julho de 2010.
- [35] Baker, M. Shah, M. Rosenthal, D. S. H. Roussopoulos, M. Maniatis, P. Giuli, TJ Bungle, P. A fresh look at the reliability of long-term digital storage. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '06, pages 221–234, New York, NY, USA, 2006. ACM.
- [36] Berlekamp, E. R. Peile, R. E. Pope, S. P. The application of error control to communications. *IEEE Communications Magazine*, 25(4):44–57, April 1987.

- [37] Haeberlen, A. Mislove, A. Druschel, P. Glacier: highly durable, decentralized storage despite massive correlated failures. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 143–158, Boston, MA, USA, 2005. USENIX Association.
- [38] Wikipedia project. Data striping. URL=http://en.wikipedia.org/wiki/Data_striping. Acessado em 01 de agosto de 2010.
- [39] Wikipedia project. Erasure code. URL=http://en.wikipedia.org/wiki/Erasure_code. Acessado em 04 de maio de 2010.
- [40] Wikipedia project. Low-density parity-check code. URL=http://en.wikipedia.org/wiki/Low-density_parity-check_code. Acessado em 04 de maio de 2010.
- [41] Wikipedia project. Wimax. URL=<http://en.wikipedia.org/wiki/WiMAX>. Acessado em 04 de maio de 2010.
- [42] Curtis, A. R. Space today online - communicating with interplanetary spacecraft. URL=<http://www.spacetoday.org/SolSys/DeepSpaceNetwork/DeepSpaceNetwork.html>. Acessado em 03 de maio de 2010.
- [43] Dabek, F. Li, J. Sit, E. Robertson, J. Kaashoek, M. F. Morris, R. Designing a dht for low latency and high throughput. In *NSDI'04: Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, pages 85–98, Berkeley, CA, USA, 2004. USENIX Association.
- [44] Gallager, G. R. *Low-Density Parity-Check Codes*. revised doctoral dissertation, Department of Electrical Engineering, MIT, Cambridge, Massachusetts, 1963.
- [45] Oriani, A. Garcia, I. C. Schmidt, R. The Search for a Highly-Available Hadoop Distributed Filesystem. Technical Report IC-10-24, Institute of Computing, University of Campinas, August 2010.

- [46] Schmuck, F. Haskin, R. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.
- [47] Dean, J. Ghemawat, S. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association.
- [48] Ghemawat, S. Gobioff, H. Leung, S. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.
- [49] Plank, J. S. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Software Practice Experience*, 27(9):995–1012, 1997.
- [50] Saito, Y. Frølund, S. and A. Spence, S. Veitch, A. Merchant. Fab: building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 48–58, New York, NY, USA, 2004. ACM.
- [51] Hefez, A. Villela, M. L. T. *Códigos Corretores de Erros*, chapter 1, 2, 3, 4, 5, 6, pages 1–125. IMPA, Rio de Janeiro, RJ, Brasil, 2008.
- [52] Hourì, Y. Jobmann, M. Fuhrmann, T. Self-organized data redundancy management for peer-to-peer storage systems. In *IWSOS '09: Proceedings of the 4th IFIP TC 6 International Workshop on Self-Organizing Systems*, pages 65–76, Berlin, Heidelberg, 2009. Springer-Verlag.
- [53] Klove, T. *Codes for Error Detection*, chapter 1, pages 1–32. World Scientific Publishing Company, Englewood Cliffs, New Jersey, USA, 2007.
- [54] Schürmann, T. The kosmos distributed fs, 2008. URL=<http://www.linux-magazine.com/Issues/2008/90/Kosmos-FS>. Acessado em 30 de junho de 2011.

- [55] White, T. *Hadoop: The Definitive Guide*, chapter 1, 2, 3, pages 1–74. O’Reilly, Sebastopol, CA, USA, 2009.
- [56] Williams, C. Huibonhoa, P. Holliday, J. Hospodor, A. Schwarz, T. Redundancy management for p2p storage. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID ’07, pages 15–22, Washington, DC, USA, 2007. IEEE Computer Society.
- [57] Huffman, W. C. Pless, V. *Fundamentals of Error-Correcting Codes*, chapter 1, pages 1–52. Cambridge University Press, New York, USA, 2003.
- [58] Kurose, J. F. Ross, K. W. *Redes de Computadores e a Internet: uma abordagem top-down*, chapter 5, pages 318–376. Addison-Wesley, São Paulo, Brasil, 2010.
- [59] Sniffin, R. W. Telemetry data decoding. URL=<http://deepspace.jpl.nasa.gov/dsndocs/810-005/208/208A.pdf>. Acessado em 03 de maio de 2010.
- [60] Yeung, R. W. *Information Theory and Network Coding*, chapter 1, pages 1–4. Springer, Englewood Cliffs, New Jersey, USA, 2008.
- [61] Hadoop Wiki. Hadoop wiki - poweredby. URL=<http://wiki.apache.org/hadoop/PoweredBy>. Acessado em 08 de maio de 2010.
- [62] Plank, J. S. Xu, L. Luo, J. Schuman, C. D. Wilcox-O’Hearn, Z. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST ’09: Proceedings of the 7th Conference on File and Storage Technologies*, pages 253–265, Berkeley, CA, USA, 2009. USENIX Association.