Instituto de Computação
Universidade Estadual de Campinas

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

Celina d'Ávila Samogin

Este exemplar corresponde à redação da Dissertação apresentada para a Banca Examinadora antes da defesa da Dissertação.

Campinas, 05 de Março de 2012.

Profa. Dra. Islene Calciolari Garcia
(Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Instituto de Computação
Universidade Estadual de Campinas

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

Celina d'Ávila Samogin[1]

30 de Janeiro de 2012

Banca Examinadora:

- Profa. Dra. Islene Calciolari Garcia (Orientadora)
- Prof. Ph.D. Luiz Eduardo Buzato
- Prof. Ph.D. Ricardo Dahab
- Prof. Dr. Edmundo Roberto Mauro Madeira (suplente)

[1]Suporte financeiro de: Bolsa do CNPq (processo XYZ) 2010–2010.

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.

Agradecimentos

Eu gostaria de agradecer ...

Dedico ao meu pai Antônio (in memoriam), à minha mãe Maria (in memoriam), às pessoas que, de alguma maneira, com suas atitudes, contribuíram para a realização deste trabalho e a todos que dedicam sua vida a pesquisa científica.

Há homens que lutam um dia, e são bons; há outros que lutam um ano, e são melhores; há aqueles que lutam muitos anos, e são muito bons. Porém há os que lutam toda a vida, esses são os imprescindíveis. (Bertolt Brecht, "Os Que Lutam")

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Abreviaturas

bit BInary digiT

BCH Bose, Chaudhuri and Hocquenghem

BSD Berkeley Software Distribution

CD Compact Disk

CCSDS Consultative Committee for Space Data Systems

CRC Cyclic Redundancy Check

DSN Deep Space Network

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

WLAN Wireless Local Area Network

WMAN Wireless Metropolitan Area Network

WiMAX Worldwide Interoperability for Microwave Access

Wi-Fi marca da Wi-Fi Alliance

XOR Exclusive OR

Capítulo 1

Introdução

A codificação por apagamento (*erasures codes*) introduz redundância em um sistema de transimissão 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 [35, 46, 62].

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 [39, 53]. 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* [36, 42, 5, 4, 6, 56] e armazenamento de grande volume de dados [9, 3, 50, 54, 25, 32, 33], como também o sistema de arquivos distribuído do Hadoop (HDFS) [18].

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 [18] e mais recentemente do algoritmo Reed-Solomon [20].

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 extenção da interface atual para aceitá-los;
- integração do código atual com o HDFS.

O texto a seguir está organizado da seguinte maneira: os Capítulos 2 e 3 introduzem os conceitos básicos da álgebra abstrata e da codificação por apagamento, respectivamente, o Capítulo 4 apresenta uma discussão sobre esquemas de redundância de dados, o Capítulo 5 comenta o *framework* Hadoop e seu sistema de arquivos, o Capítulo 6 comenta as codificações Tornado e Turbo-*Like* que foram implementadas em uma versão do Hadoop e o Capítulo 7 apresenta as contribuições e conclusões deste trabalho.

Capítulo 2

Álgebra Abstrata

Essa capítulo tem por objetivo apresentar conceitos matemáticos fundamentais dentro do escopo de álgebra abstrata para entendimento de Códigos de Blocos [55].

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 Definições

Definição 2.1 Congruência *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:

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

Definição 2.2 Classe Residual *Seja $m \in \mathbb{N}$ e $m > 1$. A classe residual módulo m do elemento $a \in \mathbb{Z}$ pode ser assim definida:*

$$\bar{a} = \{x \in \mathbb{Z} \mid x \equiv a(mod\ m)\}$$

$$\bar{a} = \{x \in \mathbb{Z} \mid x \equiv a(mod\ m)\}$$

Exemplos:

- se $m = 2$, então temos 2 classes residuais: $\bar{0} = \{... - 4, -2, 0, 2, 4, ...\}$ e $\bar{1} = \{... - 3, -1, 1, 3, ...\}$
- 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 2.3 $\mathbf{Z}_m = \{\bar{0}, \bar{1}, ..., \overline{m-1}\}$ *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.

Propriedade 1 *Se* $a \equiv b(mod\ m)$, *então* $\bar{a} = \bar{b}$

2.2 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 e inteiros *mod p* (p é primo) sob as operações de adição e multiplicação usuais.

2.3 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 $\bigoplus$ em G: $0 \bigoplus 0 = 0$, $0 \bigoplus 1 = 1$, $1 \bigoplus 0 = 1$, $1 \bigoplus 1 = 0$.

A operação $\bigoplus$ é chamada adição módulo 2. Portanto G é fechado em $\bigoplus$ e $\bigoplus$ é comutativa. Também é possível demonstrar que $\bigoplus$ 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 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 2: $GF(2^2) = GF(4) = \{0, 1, 2, 3\}$
- 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: $GF(2^3) = GF(8) = \{0, 1, 2, 3, 4, 5, 6, 7\}$

2.4.1 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: $x^q - x = 0, q = p^n$.

2.4.2 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 $GF(2^m)$ [28].

2.4.3 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

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 $n > k$. Este processo de codificação deve ser concebido de modo que a decodificação tenha solução única. Note-se que do universo de $2n$ blocos binários de comprimento n apenas $2k$ 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

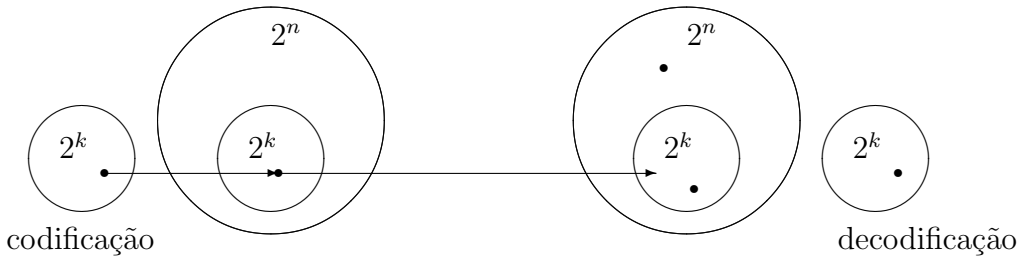

Figura 2.1: Codificação de canal

Capítulo 3

Codificação por Apagamento

A transmissão e o armazenamento de dados 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. 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 [12, 55].

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.

A codificação de mensagens no emissor antes da transmissão e a decodificação das mensagens (possivelmente danificadas) que chegam ao receptor, possibilita reparar os efeitos de um canal físico com ruídos [16] sem sobrecarregar a taxa de transmissão de informação ou o *overhead* de armazenamento [28].

Um dos principais parâmetros de um código para detecção de erros é a probabilidade de detecção de erro.

A probabilidade de erro no canal determina a capacidade de transferência de informação no canal. Os modelos estudados por pesquisadores envolvem canais simétricos, assimétricos e outros, com ou sem memória [38].

3.1 Shannon: conceitos e teoremas fundamentais

Shannon introduziu dois conceitos fundamentais sobre informação que é transmitida em um sistema de comunicação [63]:

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. O trabalho do Shannon [16] parece ser o primeiro trabalho publicado que usa o termo *bit*.

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 téorica 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 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 forma 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 [65]. Também segundo [29], existem poucos pesquisadores que são programadores de sistemas e que fazem propostas neste tema.

A idéia básica da 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 [14].

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 [57].

Em canais binários simétricos, assume-se que ambos os erros $0 \rightarrow 1$ e $1 \rightarrow 0$ ocorrem com igual probabilidade [38].

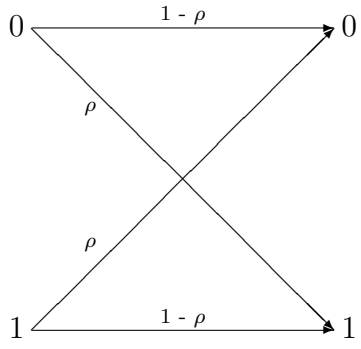

Figura 3.1: Canal binário simétrico [38]

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 [38].

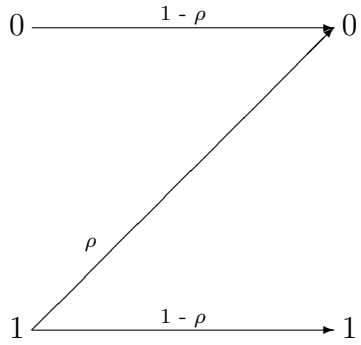

Figura 3.2: Canal binário assimétrico [38]

ρ = probabilidade da ocorrência de um erro

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 *Automatic Repeat reQuest*, os códigos são utilizados para detectar erros e se estes existirem, é feito um pedido de retransmissão. Com *Forward Error Correction*, os códigos são usados para detectar e corrigir erros e não é necessário um caminho de retorno.

3.3 *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 [28].

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 [61].

3.4 *Forward Correction Code* - FEC

Técnicas FEC representam 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 [28].

O destinatário corrige a mensagem recebida através da codificação por apagamento. 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. 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.

3.4.1 Como FEC funcionam

Técnicas FEC incluem códigos de blocos (*block codes*), que são códigos sem memória e códigos convolacionais, que são códigos com memória [41].

Códigos de Blocos

Na Figura 3.3, vemos um sistema que utiliza código de blocos. A fonte envia uma sequência de dados para o codificador. O codificador divide esta sequência em m blocos de k *bits* cada chamados mensagens. Uma mensagem é representada por uma k-tupla binária $u = u_1, u_2, \ldots, u_k$. O codificador insere *bits* redundantes (ou de paridade) para cada mensagem u, gerando uma sequência de saída de n *bits* chamada *codeword* ou palavra código representada por uma n-tupla de símbolos discretos $v = v_1, v_2, \ldots, v_n$. Os $n - k$ bits são os bits redundantes que provêm à codificação a capacidade de tratar os ruídos do canal.

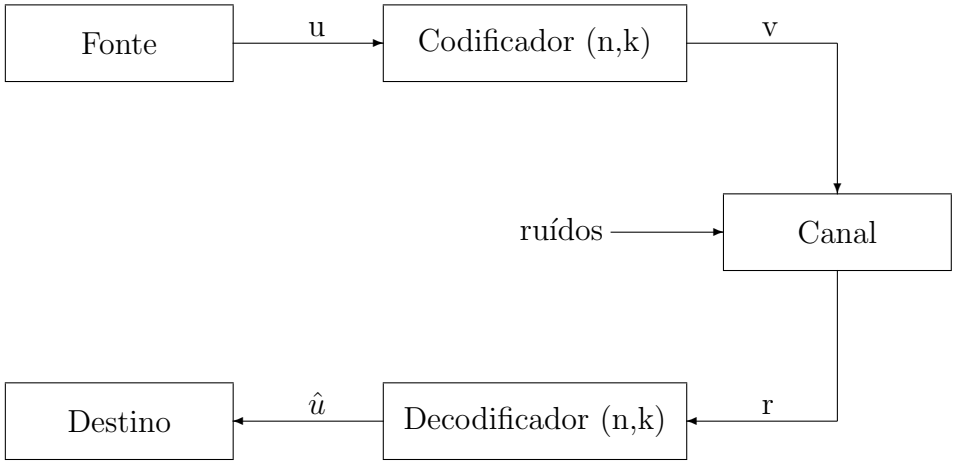

Figura 3.3: Códigos de bloco

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.

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 a sobrecarga de armazenamento são calculados a partir de m blocos originais [4, 17]. 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 a sobrecarga 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.

Outra métrica utilizada é a redundância que pode ser definida por $\frac{(n-k)}{n}$. A alta redundância 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).

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.

Existem vários códigos de blocos [1, 3]. Alguns dos mais utilizados são códigos Reed-Solomon (RS), códigos Low-Density Parity-Check (LDPC) e códigos Tornado [37, 4].

Segundo [27], 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 [1] 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 $d_{min} \geq l + t + 1$ e $l > t$

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

Códigos Reed-Solomon são códigos de blocos, lineares e cíclicos. São códigos parametrizáveis, cuja capacidade de correção de erros pode ser alterada facilmente.

Em [53], 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 $D_1, D_2, ..., D_n$ (*data devices*) e outros m dispositivos de armazenamento $C_1, C_2, ..., C_m$ (*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 $D_1, D_2, ..., D_n, C_1, C_2, ..., C_m$, o conteúdo dos dispositivos que falharam possa ser reconstituído a partir dos dispositivos que não falharam.

Segundo [35], 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 [11].

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

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.

[1] 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.

RAID foi introduzido por D. A. Patterson na Universidade da California, Berkeley (UC Berkeley) em 1988 [24].

Segundo [39], para sistemas de armazenamento, a codificação por apagamento baseada em operações simples, tais como XOR RAID, são preferíveis. Embora 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.

São conceitos básicos [13]:

data striping é uma técnica para segmentar dados sequênciais, como um arquivo, de maneira que o acesso a segmentos sequênciais 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 [43].

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 nivel 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 nivel 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 nivel 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 nivel 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 nivel 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* [**?**].

Códigos Low-Density Parity-Check e Tornado são códigos de blocos, lineares e acíclicos.

Código Low-Density Parity-Check (LPDC) é uma codificação baseada em grafos regulares [48]. Códigos LDPC são conhecidos também como códigos Gallager [44]. Uma aplicação dessa codificação é a rede *wireless* WMAN WiMAX (IEEE 802.16e *standard for microwave communications*) para internet móvel [45].

Códigos Tornado são uma classe de códigos LDPC (Low Density Parity Check) que utiliza grafos irregulares e que foi proposta por M. Luby [39]. Segundo [3], são mais rápidos para codificar e decodificar e necessitam de um pouco mais de m fragmentos para reconstruir a informação. Em [1], o autor comentou o tempo de decodificação para códigos RS e Tornado. Códigos Tornado usam equações com um número pequeno de variáveis em contraste com códigos RS.

Em [10], os autores apresentam códigos Tornado baseados em grafos irregulares. Segundo [29], as implicações práticas desses códigos ainda não foram bem estudadas.

Códigos Convolacionais

P. Elias introduziu códigos convolucionais em 1955. Um código convulacional 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 sequêncial [28].

Em um código convolucional, os $n - k$ bits redundantes, que provem 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 communications e na rede WLAN Wi-Fi IEEE 802.11.

Capítulo 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.

Para implementar redundância de dados em sistemas, são utilizadas várias técnicas: codificação por apagamento, replicação, espelhamento, *Cyclic redundancy check* (CRC), *bits* de paridade, *checksum* e assinatura digital. Esquemas de redundância podem implementar um conjunto destas técnicas [22].

Os autores em [2] estudaram redundância de dados em sistemas *peer-to-peer* para *backup* e propuseram um esquema hídrido para implementar redundância (replicação e codificação) de dados. Em [32], 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 [32].

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 [31, 40]. 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.

A Figura 4.1 apresenta um sistema de armazenamento com um arquivo de dados particionado em 8 blocos. O fator de replicação é 4. Os clientes precisam que, para cada um dos 8 distintos blocos, uma das quatro cópias esteja disponível. A Figura 4.2 apresenta o mesmo sistema que a Figura 4.1, mas utilizando códigos RS. O arquivo está particionado em 8 blocos e 32 blocos codificados foram gerados. Os clientes podem utilizar quaisquer 8 blocos para obter o arquivo inicial. A Figura 4.2 também se aplica a códigos Tornado.

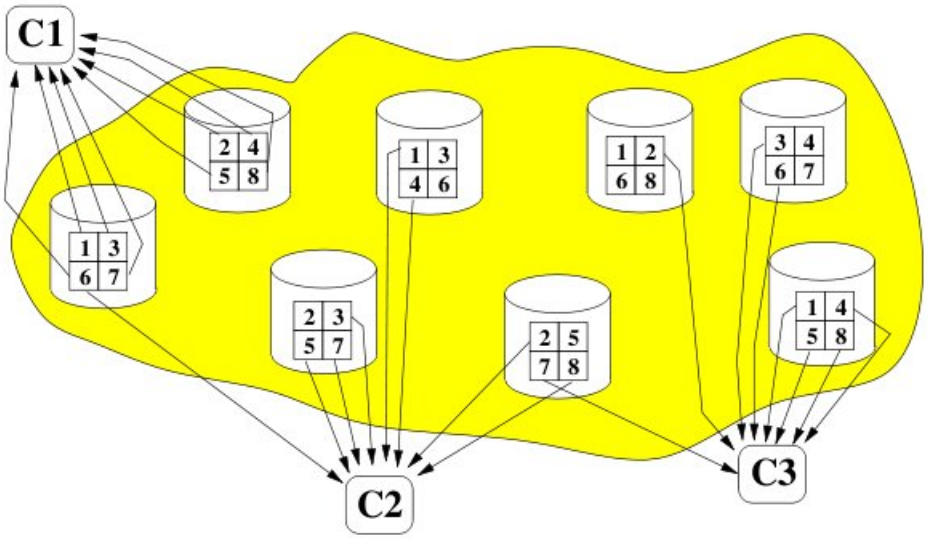

Figura 4.1: Sistema com replicação pura [23]

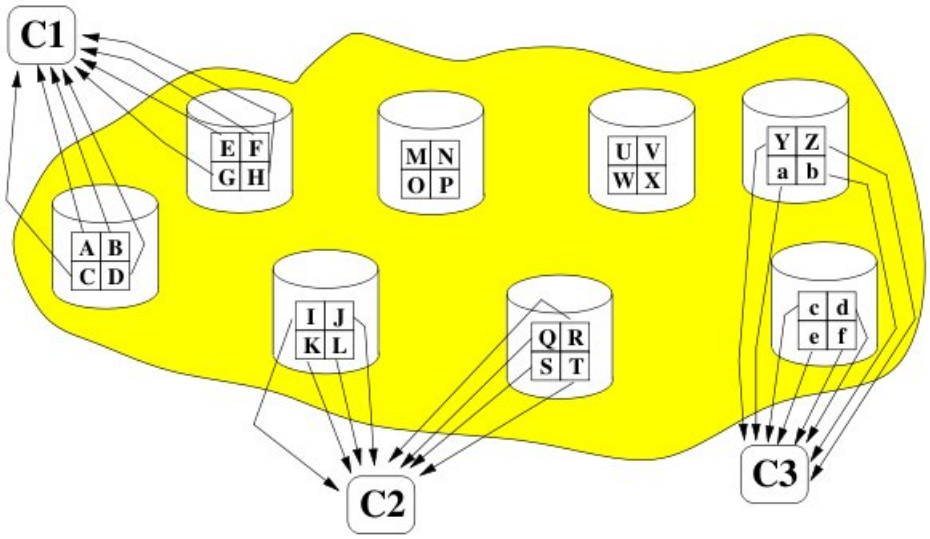

Figura 4.2: Sistema com códigos RS [23]

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 consume 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 [39]. Por exemplo, no artigo sobre o sistema Glacier [42], 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 [47] 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 [52] (GFS), no Hadoop Distributed File System [19] (HDFS) e no Kosmos distributed file system [58] (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 1^a 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 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 [8] 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 $O(logn)$ e usa uma árvore-B

[1] http://hadoop.apache.org/
[2] http://code.google.com/p/kosmosfs/

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* 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 [14]: 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 4.1, 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, 21, 5, 60] para esses dois esquemas de redundância.

[3] http://ceph.newdream.net/
[4] git://git.lustre.org/prime/lustre.git
[5] http://www.moosefs.org/

sistema	**codificação**	**arquitetura do sistema**	**licença**
HDFS [19]	RAID, Reed-Solomon, ...	shared-disk file system for cluster	Apache
Tahoe-LAFS [6] [6]	RAID	peer-to-peer filesystem	GPL2
Pergamum [32]	XOR parity, Reed-Solomon	disk-based archival storage	*
Potshards [33]	RAID e Reed-Solomon	disk-based archival storage	*
RobuStore [25]	Luby Transform (LT) codes	disk-based archival storage	*
Glacier [42]	Reed-Solomon com matrizes Cauchy	peer-to-peer storage system	*
Total Recall [36]	Maymounkov's online codes	peer-to-peer storage system	*
FAB [54]	Reed-Solomon	distributed disk array	*
GPFS [50]	RAID	shared-disk file system for cluster	*
Oceanstore [3] [7]	Tornado codes e Read-Solomon codes	desktops e notebooks conectados a servidores geograficamente distribuídos	BSD
xFS [9] [8]	RAID	serverless network file system	GPL
Swift [15]	RAID	desktops sob unix conectados a uma intranet	*

Tabela 4.1: Comparação de codificação entre sistemas de armazenamento de grande volume de dados que utilizam *commodity hardware*

onde:
* = não disponível

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.

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 inclue 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

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

grau de reparação

$$d = 1$$

disponibilidade de um dado

$$1 - q^n$$

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

Tabela 4.2: Operações sobre dados redundantes na replicação

corrupção de um dado

$$e = q^n$$

Comparando as tuplas (1, 0, 0) e (1, n, 0) da tabela 6.1, 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**

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

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 4.3, 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 $(k, k, k - 1)$.

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 4.3: Operações sobre dados redundantes na codificação por apagamento

4.3 Sobrecarga de armazenamento

Em [14, 47], os autores afirmam que a codificação obtem o mesmo nível de disponibilidade como a replicação, usando muito menos espaço de armazenamento.

Em [36], concluiu-se que se a disponibilidade do peer for 0.5, então isso requer 10 cópias de cada arquivo para garantir a disponibilidade de 0.999 dos arquivos.

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.

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

No esquema da codificação, para o mesmo objetivo, para as mesmas probabilidades e e q, obtemos um valor de 4 blocos de paridade para uma codificação $(2k-1, k)$.

$$\begin{gathered} e \;=\; q^{m-k} \\ m \;-\; k \;=\; log\ e \;/\; log\ q \end{gathered}$$

Substituindo m por $2k-1$ e $log\ e/\ log\ q$ por 3.074487147,

$$\begin{gathered} k-1 \;=\; 3.074487147 \\ k \;\approx\; 4 \end{gathered}$$

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.

Na figura 4.3, podemos observar o gráfico das funções $y = 2x$, $y = 3x$ e $y = (2x-1)/x$, 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.

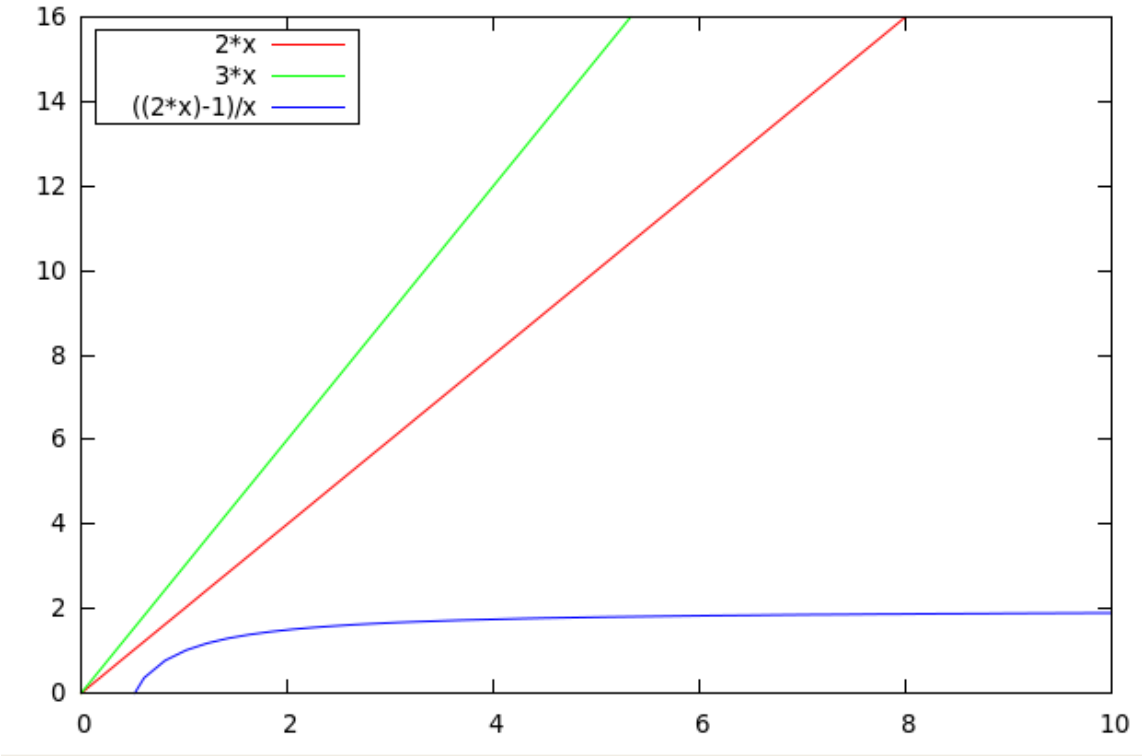

Figura 4.3: Sobrecarga de armazenamento para replicação $3n$, $2n$ e codificação $(2k-1, k)$ representadas, respectivamente, pelas funções $y = 3x$, $y = 2x$ e $y = (2x-1)/x$, para $x > 0$

Neste estudo, concluímos que a codificação por apagamento acarreta uma sobrecarga de armazenamento menor que a replicação para o mesmo número de falhas toleradas.

Nas máquinas do Facebook, a codificação RS é (13, 10) com sobrecarga de $(B = 1.3)$ e tolera até 3 falhas por *stripe*. A codificação XOR (RAID-5) é (11, 10) com sobrecarga de $(B = 1.1)$. Outro exemplo é a codificação RS(255, 223), utilizada pela DSN [62], que

é padrão CCSDS [9] e tolera 32 falhas com sobrecarga de $B = 1.1435$. Uma codificação BCH(63, 56), também padrão CCSDS, pode tolerar até 7 falhas, dependendo do polinômio gerador da síndrome [35] .

4.4 Disponibilidade dos peers

4.5 Leitura ou Atualização dos dados redundantes

Em [21], 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 $(2k - 1, k)$

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

Concluímos que, apenas avaliando a contribuição do esquema de redundância, a codificação por apagamento

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

[9] http://public.ccsds.org/

Capítulo 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 [51], 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 Lucene[1] que foi chamado Hadoop[2].

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 [64].

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

[1] http://www.apache.org
[2] http://hadoop.apache.org/

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 [59].

O MapReduce pode resolver problemas genéricos, cujos dados podem ser divididos em 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 5.2 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 5.1. 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.

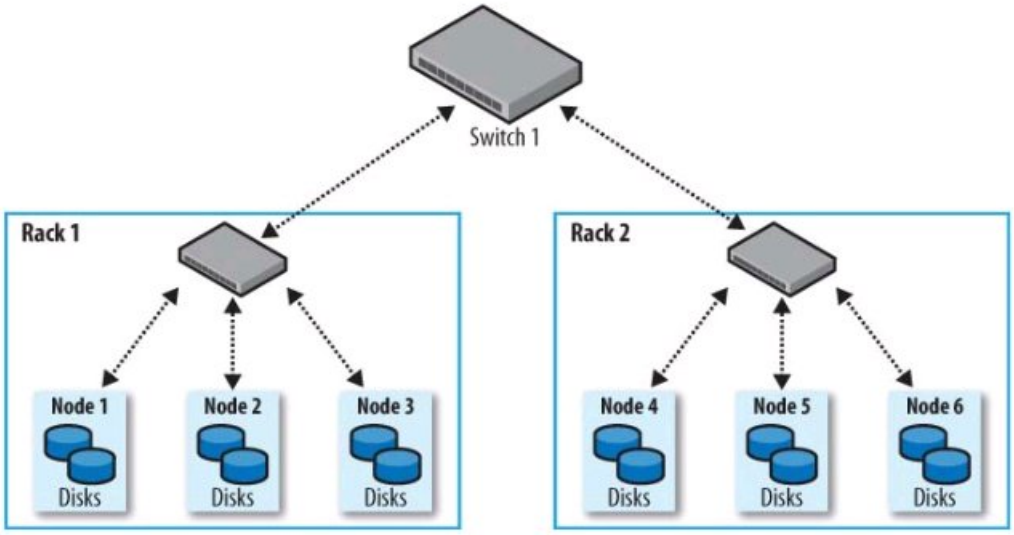

Figura 5.1: Arquitetura de rede em dois níveis para um cluster Hadoop [19]

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 blocos e esses blocos são armazenados em um conjunto de DataNodes. A Figura 5.3 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 1^a réplica fica local, no mesmo nó do código do cliente. A 2^a réplica fica em um nó em outro *rack* e a 3^a réplica fica nesse último *rack* em outro nó. As 2^a e 3^a 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 [19].

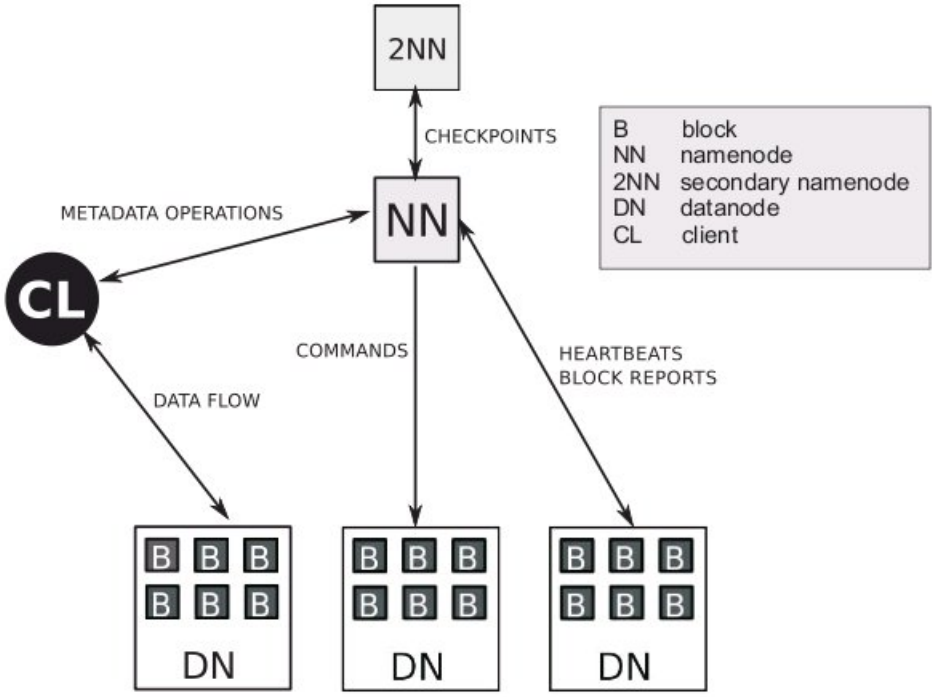

Figura 5.2: Arquitetura do HDFS [49]

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 DataNodes 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 XOR (RAID-5) [18] e uma mais recente utilizando códigos RS [20]. A inclusão da codificação por apagamento foi proposta com o objetivo de reduzir o tamanho do armazenamento do HDFS.

A versão atual do Hadoop não utiliza apenas a técnica de replicação [59] para obter disponibilidade e confiabilidade de dados. Ela pode ser configurada para usar também codicação XOR (RAID-5) e RS.

No HDFS, a codificação RS *default* é (8, 5) e a XOR *default* é (6, 5). Logo, toleram até

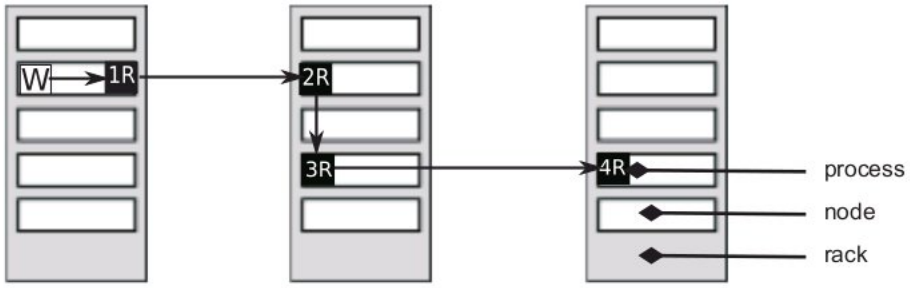

Figura 5.3: Arquitetura do HDFS - Datanodes e Blocos [59]

$l = 3$ e até $l = 1$ falhas, respectivamente, por *stripe* de 5 blocos ($k = 5$) com sobrecarga de $B = 1.6$ e de $B = 1.2$, respectivamente.

Na codificação XOR, é possível configurar o número de blocos por *stripe*. Já na codificação RS, tanto o número de blocos por *stripe* e como o número de blocos de paridade são configuráveis.

5.3.1 Algoritmos da Camada RAID

Exemplo do algoritmo de codificação

O tamanho da *stripe* é 5 blocos e existe um arquivo */a/arquivo.txt* com exatamente 5 blocos. Nesse caso, o algoritmo de codificação da camada RAID faz o seguinte:

```
bloco[0] = primeiro bloco
bloco[1] = segundo bloco
bloco[2] = terceiro bloco
bloco[3] = quarto bloco
bloco[4] = quinto bloco

bloco_paridade = iniciado com 0 em todos os bytes

para i de 0 até número de bytes em um bloco:
   para j de 0 até 4:
      bloco_paridade = bloco_paridade xor bloco[j][i]

para i de 0 até 4:
   escreva bloco_paridade no arquivo /raid/a/arquivo.txt
```

5.3.2 Algoritmos da Camada RS

Exemplo do algoritmo de codificação

O tamanho da *stripe* é 5 blocos, são 3 blocos de paridade, o polinômio primitivo é $x^8 + x^4 + x^3 + x^2 + 1$ e existe um arquivo */a/arquivo.txt* com exatamente 5 blocos. Nesse caso, o algoritmo de codificação da camada RS faz o seguinte:

```
// polinômio primitivo g representado por 285
g = x^8 + x^4 + x^2 + x + 1
m = 8
k = 5
i = 0
L = m-k
j = 0

bloco[0] = primeiro bloco
bloco[1] = segundo bloco
bloco[2] = terceiro bloco
bloco[3] = quarto bloco
bloco[4] = quinto bloco

enquanto j >= L faça
   // para GF = 2
   // x^0 representado por 1; x^1 por 2; x^2 por 4; x^3 por 8;
   // x^4 por 16; x^5 por 32; x^6 por 64; x^7 por 128;
   a = x^j
   i = 0
   bloco_paridade[j] = iniciado com 0 em todos os bytes
   enquanto i >= k faça
      r = novo bloco
      resto(g, a, bloco[i], r)
      bloco_paridade[j] = bloco_paridade[j] xor r
      i++
   j++

para j de 0 até L
   escreva bloco_paridade[j] no arquivo /raidrs/a/arquivo.txt

resto (g, a , bloco, r)
```

```
   para i de 0 até número de bytes em um bloco:
      r[i] = resto(g, a, bloco[i])

resto (g, a, b)
retorna resto = (byte) rem (g(b), a(b))
```

Capítulo 6

Implementação das Codificações

Uma boa escolha de codificação para um canal depende da sobrecarga de armazenamento, do tamanho do bloco, da complexidade dos algoritmos de codificação e de decodificação e se uma eventual não detecção de erros é aceitável. Códigos LDPC e Tornado são versáteis para serem projetados para muitas opções de tamanho do bloco e de sobrecarga de armazenamento e segundo muitos estudos, códigos LDPC são a codificação que mais se aproxima do limite de Shannon (quantidade máxima de dados transmitidos em um canal em bytes por segundo).

6.1 Codificação Tornado

A idéia básica deste algoritmo da codificação Tornado está descrita em [39, 30].

O cálculo do checksum de 64 bits foi feito para cada 100 bytes. Para um bloco de 64MB, o checksum é de 5.12MB.

A sobrecarga de armazenamento de checksum é $864/800 = 1.08$. A sobrecarga da paridade é $2n/n = 2$. Portanto, a sobrecarga total de armazenamento é $1.08 * 2 = 2.16$.

O número de falhas suportadas é $l = \textit{numero de blocos da stripe} = 10$.

Seja $s = \textit{numero de blocos da stripe} = 10$.

A operação mais custosa dos algoritmos é o cálculo do XOR entre 2 blocos de 64M. São $s/2$ operações, no caso de s par e $(n/2)-1$ operações, no caso de s ímpar. Nesse caso, podemos afirmar que os algoritmos são lineares no tamanho da entrada, ou seja, $O(s)$.

O grafo da codificação para uma *stripe* de tamanho 10 blocos é:

```
1 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0
```

```
0 0 0 1 1 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0
0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 1 1
```

6.1.1 Algoritmo de Codificação

Apresentamos um exemplo do algoritmo de codificação para uma *stripe* de tamanho 10 blocos e para um arquivo */a/arquivo.txt* com exatamente 10 blocos. Nesse caso, o algoritmo de codificação faz o seguinte:

- bloco[0] = primeiro bloco da stripe
- bloco[1] = segundo bloco da stripe
- ...
- bloco[9] = último bloco da stripe
- para i de 0 até 9
  - bloco_checksum[i] = checksum do bloco[i]
- bloco_paridade[0] = bloco[0]
- bloco_paridade[1] = bloco[0] *xor* bloco[1]
- bloco_paridade[2] = bloco[1] *xor* bloco[2]
- bloco_paridade[3] = bloco[2] *xor* bloco[3]
- bloco_paridade[4] = bloco[3] *xor* bloco[4]
- bloco_paridade[5] = bloco[4] *xor* bloco[5]
- bloco_paridade[6] = bloco[5] *xor* bloco[6]
- bloco_paridade[7] = bloco[6] *xor* bloco[7]
- bloco_paridade[8] = bloco[7] *xor* bloco[8]
- bloco_paridade[9] = bloco[8] *xor* bloco[9]

- para i de 0 até 9:
  - escreva bloco_checksum[i] no arquivo */checksum/a/arquivo.txt*
- para i de 0 até 9:
  - escreva bloco_paridade[i] no arquivo */tor/a/arquivo.txt*

6.1.2 Algoritmo de Decodificação

O algoritmo de decodificação faz o seguinte:

- bloco_checksum[0] = primeiro bloco de checksum da stripe
- bloco_checksum[1] = segundo bloco de checksum da stripe
- ...
- bloco_checksum[9] = último bloco de checksum da stripe
- $i = 0$
- $erro = false$
- enquanto ($i <= 9$) e ($nao\ erro$) faça
  - se bloco_checksum[i] = checksum do bloco[i]
  - então $i++$
  - senão $erro = true$
- se erro
- então
  - Ler o arquivo */tor/a/arquivo.txt*
  - bloco_paridade[0] = primeiro bloco de paridade da stripe
  - bloco_paridade[1] = segundo bloco de paridade da stripe
  - ...
  - bloco_paridade[9] = último bloco de paridade da stripe
  - bloco[0] = bloco_paridade[0]
  - bloco[1] = bloco_paridade[1] xor bloco[0]

- bloco[2] = bloco_paridade[2] *xor* bloco[1]
- bloco[3] = bloco_paridade[3] *xor* bloco[2]
- bloco[4] = bloco_paridade[4] *xor* bloco[3]
- bloco[5] = bloco_paridade[5] *xor* bloco[4]
- bloco[6] = bloco_paridade[6] *xor* bloco[5]
- bloco[7] = bloco_paridade[7] *xor* bloco[6]
- bloco[8] = bloco_paridade[8] *xor* bloco[7]
- bloco[9] = bloco_paridade[9] *xor* bloco[8]

6.2 Codificação Simples Turbo-*Like*

Esse algoritmo foi baseado nos estudos de [26, 7] e é muito simples de entender.

A sobrecarga total de armazenamento é $2n/n = 2$.

O número de falhas suportadas é $l = numero\ de\ blocos\ da\ stripe = 10$.

Seja $s = numero\ de\ blocos\ da\ stripe = 10$.

A operação mais custosa dos algoritmos é o cálculo do XOR entre 2 blocos de 64M. São s operações. Nesse caso, podemos afirmar que os algoritmos são lineares no tamanho da entrada, ou seja, $O(s)$.

O grafo da codificação para uma *stripe* de tamanho 10 blocos na sequência $7, 1, 4, 8, 2, 5, 9, 3, 0, 6$:

```
0 0 0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 1 0
0 0 1 0 0 0 0 0 1 0
0 0 1 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 1
0 0 0 1 0 0 0 0 0 1
1 0 0 1 0 0 0 0 0 0
1 0 0 0 0 0 1 0 0 0
```

6.2.1 Algoritmo de Codificação

Apresentamos um exemplo do algoritmo de codificação para uma *stripe* de tamanho 10 blocos e para um arquivo */a/arquivo.txt* com exatamente 10 blocos. Nesse caso, o algoritmo de codificação faz o seguinte:

- bloco[0] = primeiro bloco da stripe
- bloco[1] = segundo bloco da stripe
- ...
- bloco[9] = último bloco da stripe
- Gerar uma permutação dos índices dos blocos da stripe: $p = [b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9]$
- bloco_paridade[0] = bloco[b_0]
- bloco_paridade[1] = bloco_paridade[0] *xor* bloco[b_1]
- bloco_paridade[2] = bloco_paridade[1] *xor* bloco[b_2]
- bloco_paridade[3] = bloco_paridade[2] *xor* bloco[b_3]
- bloco_paridade[4] = bloco_paridade[3] *xor* bloco[b_4]
- bloco_paridade[5] = bloco_paridade[4] *xor* bloco[b_5]
- bloco_paridade[6] = bloco_paridade[5] *xor* bloco[b_6]
- bloco_paridade[7] = bloco_paridade[6] *xor* bloco[b_7]
- bloco_paridade[8] = bloco_paridade[7] *xor* bloco[b_8]
- bloco_paridade[9] = bloco_paridade[8] *xor* bloco[b_9]
- para i de 0 até 9:
  - escreva bloco_paridade[i] no arquivo */turbo/a/arquivo.txt*

6.2.2 Algoritmo de Decodificação

O algoritmo de decodificação faz o seguinte:

- Ler a permutação do índices dos blocos da stripe: $p = [b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9]$
- Ler o arquivo */turbo/a/arquivo.txt*
- bloco_paridade[0] = primeiro bloco de paridade da stripe
- bloco_paridade[1] = segundo bloco de paridade da stripe

- ...
- bloco_paridade[9] = último bloco de paridade da stripe
- bloco[b_0] = bloco_paridade[0]
- bloco[b_1] = bloco_paridade[0] *xor* bloco_paridade[1]
- bloco[b_2] = bloco_paridade[1] *xor* bloco_paridade[2]
- bloco[b_3] = bloco_paridade[2] *xor* bloco_paridade[3]
- bloco[b_4] = bloco_paridade[3] *xor* bloco_paridade[4]
- bloco[b_5] = bloco_paridade[4] *xor* bloco_paridade[5]
- bloco[b_6] = bloco_paridade[5] *xor* bloco_paridade[6]
- bloco[b_7] = bloco_paridade[6] *xor* bloco_paridade[7]
- bloco[b_8] = bloco_paridade[7] *xor* bloco_paridade[8]
- bloco[b_9] = bloco_paridade[8] *xor* bloco_paridade[9]

Codificação	**Número de falhas suportadas**	**Espaço de armazenamento**	**Grau de reparacao**	**Corrupção de um dado**	**Complexidade dos algoritmos**
Replicação 2X	$l = 2$	$B = 2$	$d = 1$	$e = q^2$	$O(1)$
RAID (6, 5)	$l = 1$	$B = 1.2$	$d = 5$	$e = q$	$O(5log6)$
RS (8, 5)	$l = 3$	$B = 1.6$	$d = 5$	$e = q^3$	$O(5(3log8))$
Tornado (10, 5)	$l = 5$	$B = 2$	$d = 5$	$e = q^5$	$O(10)$
Turbo-*Like* (10, 5)	$l = 5$	$B = 2$	$d = 5$	$e = q^5$	$O(10)$
Replicação nX	$l = n$	$B = n$	$d = 1$	$e = q^{n-1}$	$O(n)$
RS $(2k, k) ou (2k - 1, k)$	$l = k$	$B = 2$	$d = k$	$e = q^k$	$O(k^2 log2k)$[34]
Tornado $(2k, k) ou (2k - 1, k)$	$l = k$	$B = 2$	$d = k$	$e = q^k$	$O(2k)$[34]
Turbo-*Like* $(2k, k) ou (2k - 1, k)$	$l = k$	$B = 2$	$d = k$	$e = q^k$	$O(2k)$

Tabela 6.1: Comparação entre as codificações implementadas (nem todas ainda!) no HDFS

Capítulo 7

Conclusões

Referências Bibliográficas

[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] Kubiatowicz, 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 dhts: 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] MacKay, D. J. C. *Information Theory, Inference, and Learning Algorithms*, chapter 49, pages 582–587. Cambridge University Press, Cambridge, England, 2003.

[8] 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.

[9] 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.

[10] 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.

[11] 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, Department 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.

[12] Ritter, D. Um estudo sobre códigos corretores de erros em espaços sobre posets. Master's thesis, Instituto de Matemática, Estatística e Computação Científica - Departamento de Matemática, Campinas, Brasil, fevereiro 2009.

[13] Vadala, D. *Managing RAID on Linux*, chapter 1, pages 1–10. O'Reilly, Sebastopol, CA, USA, 2002.

[14] Weatherspoon, H. Kubiatowicz, 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.

[15] 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.

[16] Shannon, C. E. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 623–656, 1948.

[17] 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.

[18] 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.

[19] Apache Software Foundation. Hdfs architecture. URL=http://hadoop.apache.org/common/docs/current/hdfs_design.html. Acessado em 08 de maio de 2010.

[20] 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.

[21] 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.

[22] Fan, B. Tantisiriroj, W. Xiao, L. Gibson, G. Diskreduce: Raid for data-intensive scalable computing. In *PDSW '09: Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 6–10, New York, NY, USA, 2009. ACM.

[23] Plank, J. S. Thomason, M. G. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *DSN*, pages 115–124, 2004.

[24] 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.

[25] 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.

[26] Divsalar, D. Jin, H. McEliece, R. J. Coding theorems for "turbo-like"codes. In *Proceedings of the 36th Allerton Conference on Communication, Control, and Computing*, pages 201–210, 1998.

[27] 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.

[28] 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.

[29] Plank, J. Cs560 - operating systems. URL=http://www.cs.utk.edu/plank/plank/classes/cs560. Acessado em 08 de maio de 2010.

[30] Stoten, J. Digital fountains. URL=http://www.jonathanstoten.com/fountains.html. Acessado em 10 de julho de 2011.

[31] Weatherspoon, H. Moscovitz, T. Kubiatowicz, J. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. *21st IEEE Symposium on Reliable Distributed Systems*, 0:362–367, 2002.

[32] 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.

[33] 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.

[34] M. Luby. Lt codes. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, pages 271–, Washington, DC, USA, 2002. IEEE Computer Society.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Woitaszek, M. *Tornado Codes for Archival Storage*. PhD thesis, Department of Computer Science of the University of Colorado, Boulder, Colorado, USA, dezembro 2007.

[40] Baker, M. Shah, M. Rosenthal, D. S. H. Roussopoulos, M. Maniatis, P. Giuli, TJ Bungale, 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.

[41] 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.

[42] 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.

[43] Wikipedia project. Data striping. URL=http://en.wikipedia.org/wiki/Data_striping. Acessado em 01 de agosto de 2010.

[44] 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.

[45] Wikipedia project. Wimax. URL=http://en.wikipedia.org/wiki/WiMAX. Acessado em 04 de maio de 2010.

[46] 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.

[47] 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.

[48] Gallager, G. R. *Low-Density Parity-Check Codes*. revised doctoral dissertation, Department of Electrical Engineering, MIT, Cambridge, Massachusetts, 1963.

[49] 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.

[50] 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.

[51] Dean, J. Ghemawat, S. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association.

[52] 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.

[53] Plank, J. S. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Software Practice Experience*, 27(9):995–1012, 1997.

[54] 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.

[55] 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.

[56] Houri, 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.

[57] Klove, T. *Codes for Error Detection*, chapter 1, pages 1–32. World Scientific Publishing Company, Englewood Cliffs, New Jersey, USA, 2007.

[58] 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.

[59] White, T. *Hadoop: The Definive Guide*, chapter 1, 2, 3, pages 1–74. O'Reilly, Sebastopol, CA, USA, 2009.

[60] 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.

[61] 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.

[62] 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.

[63] Yeung, R. W. *Information Theory and Network Coding*, chapter 1, pages 1–4. Springer, Englewood Cliffs, New Jersey, USA, 2008.

[64] Hadoop Wiki. Hadoop wiki - poweredby. URL=http://wiki.apache.org/hadoop/PoweredBy. Acessado em 08 de maio de 2010.

[65] 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.