**Universidade de Aveiro** Departamento de Electrónica, Telecomunicações e Informática,
**2009**

**Pedro Miguel Boto Saraiva**

# NFS Fountain - Sistema de Ficheiros Distribuído com Códigos Fountain

**Pedro Miguel Boto Saraiva**

# NFS Fountain - Sistema de Ficheiros Distribuído com Códigos Fountain

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica de José Vieira e André Zúquete, Professores do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

**o júri / the jury**

presidente / president

**Doutor António Rui Oliveira Silva Borges**
Professor Associado da Universidade de Aveiro

vogais / examiners committee

**Doutor João Pedro Barreto**
Professor Auxiliar do Instituto Superior Técnico de Lisboa

**Doutor José Manuel Neto Vieira**
Professor Auxiliar da Universidade de Aveiro (orientador)

**Doutor André Ventura da Cruz Marnôto Zúquete**
Professor Auxiliar da Universidade de Aveiro (co-orientador)

**agradecimentos / acknowledgements**

Dedico este trabalho aos meus pais e avós.

**Resumo**

Actualmente, mais do que nunca, o processamento e armazenamento de informação é um requisito essencial em qualquer sociedade. Com o crescimento da procura de armazenamento de informação em formato digital, aumentam os riscos da sua perda. Isto deve-se à falta de fiabilidade dos sistemas de armazenamento actuais. Uma possível solução para este problema, passa pelo armazenamento de *backups* da informação em vários sistemas de armazenamento. No entanto, esta solução não é de todo a melhor devido à necessidade de replicar totalmente a informação em cada sistema.

Com a introdução dos códigos *Fountain*), surge um novo método para transmissão de informação sem erros. Estes códigos permitem a recuperação da informação original, através de um subconjunto quaisquer de blocos codificados. Imagine-se que a informação é codificada e distribuída por vários sistemas de armazenamento. Mesmo que alguns sistemas avariem, existe um boa probabilidade de recuperar a informação original.

O trabalho desta dissertação reflecte a construção de um sistema de armazenamento distribuído com tolerância a falhas de informação. A informação é armazenada em formato codificado com redundância, através de uma implementação do código *Fountain*. É avaliado do uso deste tipo de códigos, comprovando-se que a construção de sistemas com integração de códigos *Fountain* pode ser uma boa solução para aplicar em sistemas de armazenamento num futuro próximo.

**Abstract**

Nowadays, more than ever, the processing and storage of data is a critical requirement in any society. With the growth in demand of digital data storage, the risk of data loss is increased. This happens due the lack of reliability of the current storage systems. A possible solution to this problem, is the storage of multiple backups of the data in multiple storage systems. However, that is not a good solution because the data must be fully replicated in each system.

With the introduction of rateless codes for erasure channels (Fountain codes), arises a new method to transmit data without errors. These codes allow the recovery of the original data through any subset of encoded blocks of data. Imagine that the data is encoded and distributed across multiple storage systems. Even if some systems fails, it would still be possible to recover the original data with great probability.

The work in this dissertation reflects on the development of a distributed storage system with data failure tolerance. The data is stored in encoded format with redundancy, through a Fountain code. It is avaliated the use of this type of codes, proving that the development of systems with Fountain codes can be a good solution to be applied on future storage systems.

# Conteúdo

# Lista de Tabelas

# Lista de Figuras

# Abreviaturas

**BCRS** Block Compressed Row Storage

**CCS** Compressed Column Storage

**CDC** Codificador e Descodificador de Codewords

**CFS** Cryptographic File System

**CRS** Compressed Row Storage

**CSC** Cliente do Servidor de Codewords

**CSM** Coordinate Storage Matrix

**GFS** Google File System

**IETF** Internet Engineering Task Force

**IP** Internet Protocol

**JDS** Jagged Diagonal Storing

**LT** Luby Transform

**MRU** Most Recently Used

**NFS** Network File System

**P2P** Peer To Peer

**RPC** Remote Procedure Call

**RAID** Redundant Array of Inexpensive Disks

**SC** Servidor de Codewords

**TCP** Transmission Control Protocol

**UDP** User Datagram Protocol

**VDC** Visualizador do Descodificador de Codewords

**VFS** Virtual File System

**XDR** eXternal Data Representation

# Capítulo 1

# Introdução

## 1.1 Motivação

O crescente aumento das necessidades de armazenamento de informação coloca sobre os sistemas de armazenamento uma crescente pressão para que estes forneçam grande fiabilidade, escalabilidade e baixo custo. O *Google File System* (GFS) [12] é um bom exemplo de um sistema com uma arquitectura construída para responder às novas necessidades dos sistemas de armazenamento.

Um dos aspectos mais descurado nos sistemas de armazenamento é a capacidade dos sistema de ficheiros poder em aumentar a sua capacidade de forma arbitrária, transparente e escalável.

Outra característica importante é a resistência a falhas do sistema de armazenamento. É comum um sistema de armazenamento ficar inoperacional devido à falha de apenas alguns sectores de um dos discos. Os sistemas RAID [19] e o GFS, bem como outros sistemas de ficheiros, permitem minorar este problema através da introdução de redundância. Nos sistemas RAID 5, com 4 discos, são necessários apenas 3 discos para recuperar completamente a informação original. No entanto, estes sistemas são pouco flexíveis e escaláveis, já que não permitem acrescentar mais discos para expandir a capacidade. O GFS é mais flexível, usa apenas replicação integral e permite aumentar, balancear e ajustar o nível de replicação através da adição de servidores de réplicas (*chunk* servers).

O aparecimento dos códigos "*rateless*" ou "*Fountain Codes*" [21, 20, 22] veio abrir um novo conjunto de soluções para este tipo de problemas. No caso do armazenamento de dados, é possível distribuir a informação de um ficheiro com redundância por um número virtualmente infinito de discos. Estes códigos permitem recuperar a informação original de qualquer conjunto não ordenado de palavras codificadas. Assim, pode-se imaginar um sistema de armazenamento de ficheiros que utiliza um número elevado de discos dispersos por um conjunto heterogéneo de máquinas, mas capaz de recuperar a informação original mesmo que algumas das máquinas não estejam operacionais.

## 1.2 Requisitos/Objectivos

Tendo em atenção estes aspectos, seria interessante construir um sistema de armazenamento de dados com capacidade de armazenar informação num conjunto elevado de discos existentes numa rede. Este sistema deve ser capaz de recuperar a informação original mesmo

que uma percentagem elevada, mas limitada, de discos esteja inacessível. A informação é codificada usando um código da família dos códigos *Fountain*, designado código LT. Pretendeu-se estudar e avaliar a capacidade, desempenho e complexidade deste tipo de códigos. Pretendeu-se também estudar como se podia adaptar um sistema de codificação de ficheiros a um sistema de armazenamento de ficheiros.

## 1.3 Solução

Nesta dissertação de mestrado pretende-se construir um sistema de ficheiros distribuído, onde a informação original não é armazenada directamente mas sob uma forma codificada. Os dados originais são divididos em blocos de dimensão adequada (símbolos) e cada bloco codificado (*codeword*) é obtido pelo "ou" exclusivo de um número aleatório de blocos dos dados originais. O conjunto de blocos será igualmente obtido de forma aleatória. O número de blocos a utilizar para formar cada *codeword* segue uma distribuição de probabilidade que garante algumas características do código. As *codewords* são distribuídas pelos diferentes sistemas de armazenamento, localmente ou remotamente. O nível de redundância, isto é, o número de *codewords* a gerar em relação ao número símbolos, está relacionado com a disponibilidade média de cada um dos sistemas de armazenamento.

A recuperação dos dados originais pode ser realizada a partir de qualquer conjunto de *codewords*. O código pode ser projectado de modo a que uma pequena percentagem acima do número de blocos dos dados originais seja suficiente para garantir a reconstrução da informação original com uma probabilidade de sucesso tão grande quanto se queira.

## 1.4 Resultados

Foi construído um sistema de ficheiros usando uma aproximação *user-level*. Este é gerido através de um servidor NFS (*Network File System*) no *loopback*. O servidor NFS usa a interface VFS (*Virtual File System*) do Unix de modo a armazenar em cache a informação original em disco.

Existe um *deamon* que regista e verifica o acesso à informação em cache. Sempre que existe informação em cache que não é acedida à um tempo previamente definido, a informação é codificada e enviada para servidores de *codewords*. A informação original é eliminada da cache. Quando a informação que não está presente em cache volta a ser acedida, o servidor NFS tenta reconstruir a informação original obtendo *codewords* referentes a essa informação aos servidores de *codewords*. Após a descodificação, a informação volta a ficar em cache.

## 1.5 Organização da Dissertação

Os capítulos encontram-se ordenados de acordo com a ordem de trabalho seguida no desenvolvimento deste trabalho, e podem ser caracterizados da seguinte forma:

**Capítulo 2 - Contexto:** Neste capítulo são apresentados alguns conceitos de tecnologias que foram usadas no desenvolvimento do sistema NFS Fountain. Começa com uma introdução aos códigos LT, onde é abordado o processo de codificação e descodificação de informação. De seguida, são apresentadas estruturas de dados computacionais capazes de armazenar matrizes de natureza esparsa, necessárias para o uso do código LT.

Seguidamente é apresentado um componente fundamental do sistema NFS Fountain, o protocolo NFS. São abordadas as suas características e forma de comunicação entre clientes e servidores NFS.

**Capítulo 3** Neste capítulo são apresentados sistemas de ficheiros já desenvolvidos que se assemelham, de alguma forma, ao trabalho desenvolvido nesta dissertação. No início é apresentado o CFS, um sistema de ficheiros que armazena informação num estado encriptado. De seguida, é apresentado o GFS, um sistema de ficheiros distribuído desenvolvido com o objectivo de lidar com as questões de reabilidade de armazenamento e processamento de grandes quantidades de informação. Finalmente, é apresentado o sistema Avalanche, um sistema distribuído P2P que usa uma variante dos códigos Fountain para codificar a informação. O sistema Avalanche tem como objectivo a partilha de ficheiros com grande dimensão.

**Capítulo 4 - Arquitectura do NFS Fountain:** Neste capítulo é apresentada a arquitectura geral do sistema NFS Fountain. Inicialmente, é apresentado o modo de funcionamento do sistema e a interacção entre os diversos componentes que o constituem, sendo estes descritos brevemente. De seguida, é apresentada a arquitectura de cada componente que o constitui em detalhe.

**Capítulo 5 - Realização:** Neste capítulo são apresentados os detalhes do desenvolvimento do sistema NFS Fountain. É uma vista mais aprofundada do capítulo anterior, onde são mostrados os detalhes da implementação dos componentes que constituem o sistema NFS Fountain.

**Capítulo 6 - Avaliação:** Neste capítulo são apresentados os resultados de simulações usando os componentes do sistema NFS Fountain. É discutida a eficiência do codificador e descodificador de códigos LT construído e do sistema NFS Fountain em geral.

**Capítulo 7 - Conclusões e Trabalho Futuro:** Este capítulo conclui a dissertação. São apresentadas as conclusões principais do trabalho realizado bem como uma breve discussão geral dos resultados obtidos. Para terminar, são apresentadas algumas ideias para trabalho futuro sobre o tema desenvolvido neste trabalho.

**Anexo A - Visualizador do Descodificador de *Codewords*:** Neste anexo é apresentado uma interface gráfica construída neste trabalho com o objectivo de o utilizador ter uma percepção visual do processo de descodificação de informação.

**Anexo B - Bibliotecas de Programação:** Neste anexo são descritas brevemente as bibliotecas de programação usadas no desenvolvimento do sistema NFS Fountain.

**Anexo C - Resultados das Simulações:** Este anexo mostra os resultados obtidos nas simulações de codificação e descodificação do sistema NFS Fountain, usados para obter os resultados médios apresentados no capítulo 6.

# Capítulo 2

# Contexto

## 2.1 Sumário

Um aspecto importante no trabalho realizado nesta dissertação é o conceito de códigos Fountain. Os códigos Fountain, também designados por códigos *rateless* para canais de apagamentos, são uma classe de códigos de correcção que possuem a característica de poder gerar uma infinidade de blocos codificados a partir de um conjunto de blocos originais. Outra característica essencial é particularidade de conseguirem recuperar os blocos originais a partir de um qualquer subconjunto de blocos codificados com tamanho igual ou ligeiramente superior ao número de blocos originais, com grande probabilidade. Diz-se que um código Fountain é óptimo se conseguir recuperar $k$ blocos originais a partir de $k$ blocos codificados.

Para o desenvolvimento do sistema NFS Fountain desta dissertação, foi desenvolvida uma implementação do código LT. O código LT é um código Fountain quase-óptimo, pelo que normalmente são necessários um pouco mais de $k$ blocos codificados para recuperar $k$ blocos originais. Neste capítulo, são abordadas as características deste tipo de códigos, e são apresentadas algumas estruturas de dados que podem servir para a sua implementação.

O sistema NFS Fountain é um sistema de ficheiros distribuído. O seu desenvolvimento passou pela implementação de um servidor que processa pedidos de clientes. Este servidor é responsável pelo armazenamento e envio de informação a pedido dos clientes. O desenvolvimento deste servidor foi realizado através de uma implementação do protocolo NFS. O NFS é um protocolo de comunicação que permite a construção de sistemas de ficheiros distribuídos. Mais à frente neste capítulo, são abordadas as características deste protocolo.

## 2.2 Código LT

O conceito de código Fountain surgiu da metáfora onde uma fonte (codificador) verte um número, teoricamente ilimitado de gotas (blocos codificados). No processo de codificação, a informação é dividida em $k$ símbolos com o mesmo tamanho $t$. São geradas $N$ *codewords*, onde $N$ é um valor muito grande, teoricamente ilimitado. A ideia é conseguir descodificar a informação original através $k(1+\varepsilon)$ *codewords*, onde $\varepsilon \approx 0$.

Existem vários códigos da família Fountain. A primeira realização prática de um código Fountain, foi o código LT. O código LT (*Luby Transform*) foi desenvolvido por Michael Luby em 2002 [20]. Foram a primeira aproximação ao conceito de *Digital Fountain*, apresentando boa eficiência no processo de codificação e descodificação.

| Codeword | Grau | Expressão | Valor |
|---|---|---|---|
| C1 | 3 | $S1 \oplus S2 \oplus S4$ | $1 \oplus 1 \oplus 0 = 0$ |
| C2 | 2 | $S1 \oplus S3$ | $1 \oplus 0 = 1$ |
| C3 | 1 | $S2$ | 1 |
| C4 | 2 | $S2 \oplus S4$ | $1 \oplus 0 = 1$ |

Tabela 2.1: *Codewords* geradas pela codificação LT.

Apesar de o código LT não ser o código mais eficiente actualmente, foi feito um estudo e uma implementação deste. Actualmente, o código Fountain mais utilizado na prática é o código *Raptor*, definido pelo IETF, que consegue alcançar um tempo de codificação e descodificação linear. A escolha do código LT, prende-se com o facto de ser o primero código desenvolvido e também o mais simples de compreender. O código *Raptor* é uma evolução do código LT, pelo que será interessante desenvolver uma implementação deste no futuro.

### 2.2.1 Codificação

Uma *codeword* é construída a partir de um ou mais símbolos. A dependência de símbolos é traduzida na prática, através de uma combinação linear de um ou mais símbolos, escolhidos com aleatoriedade. Habitualmente, a combinação linear é feita através da operação lógica "ou-exclusivo" ($\oplus$) entre símbolos. O número de símbolos que fazem parte da combinação linear da *codeword* define-se como grau. O grau de uma *codeword* é extremamente relevante para a descodificação dos símbolos. Concretamente, deve-se assegurar que o grau das *codewords* respeita certas condições, garantindo uma descodificação dos símbolos com sucesso. Para assegurar essas condições, o grau da *codeword* é determinado de forma aleatória seguindo uma distribuição de probabilidade específica ao código LT. As características desta distribuição são apresentadas mais tarde na secção 2.2.3.

O algoritmo de codificação pode ser descrito iterativamente usando o seguinte procedimento:

1. É escolhido aleatoriamente o grau $d$ da *codeword*.
2. São seleccionados aleatoriamente $d$ símbolos.
3. Os símbolos seleccionados são combinados usando a operação $\oplus$.
4. Voltar a (1) para criar outra *codeword*.

Considere-se um exemplo simples. Uma mensagem com 4 símbolos, $S(i) = 1100$, onde cada símbolo $S_i$ é representado por 1 *bit*. São cridas 4 *codewords* usando o algoritmo de codificação descrito atrás. A tabela 2.1, mostra um possível exemplo das *codewords* criadas na codificação da mensagem $S$. Temos então, duas *codewords* de grau 2, uma com grau 1 e outra com grau 3.

Uma forma de visualizar as *codewords* criadas no processo de codificação, é através de uma matriz esparsa binária. Numa matriz, a linha $i$ representa a *codeword* $i$. A coluna $j$ indica todas as *codewords* que dependem do símbolo $j$. A figura 2.1 mostra a matriz $A$, usando como exemplo as *codewords* codificadas na tabela 2.1. Caso o elemento $A_{ij} = 1$, então a *codeword* $i$ depende do símbolo $j$.

$$A = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Figura 2.1: Matriz $A$ das *codewords* na tabela 2.1.

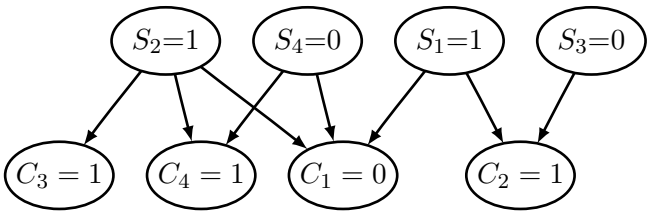


Figura 2.2: Grafo $G$ que representa as dependências de símbolos nas *codewords*.

A matriz $A$ pode ser representada vizualmente através de um grafo $G$. A figura 2.2 apresenta as dependências de símbolos nas *codewords* através do grafo $G$. O sentido das setas que ligam os nós do grafo, indicam que um símbolo é usado na combinação linear que compõe uma *codeword*. Por outras palavras, mostra a dependência de símbolos nas *codewords*.

### 2.2.2 Descodificação

No processo de descodificação, supõe-se que o descodificador tem conhecimento da matriz $A$ de codificação. Esta informação é composta pelo grau e dos símbolos que compõe as *codewords*.

O algoritmo de descodificação pode ser descrito iterativamente usando os seguintes passos:

1. Procurar uma *codeword* com grau 1, dependente apenas do símbolo $S_i$.
2. O símbolo $S_i$ é descodificado.
3. Seleccionam-se todas as *codewords* que dependem do símbolo $S_i$.
4. A essas *codewords* é removida a dependência do símbolo $S_i$ através da operação $\oplus$.
5. Voltar a (1) enquanto houver símbolos por descodificar.

Considere-se o exemplo anterior da figura 2.1. Como matriz de codificação $A$ é conhecida, pode-se recorrer a grafos para visualizar o processo de descodificação. As figuras 2.3 a 2.14 visualizam e descrevem o processo de descodificação.

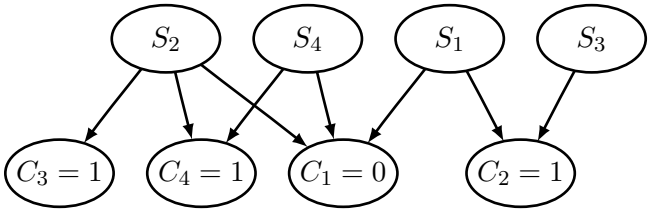


Figura 2.3: Inicialmente é apenas conhecido o grau e as dependências de cada *codeword*.

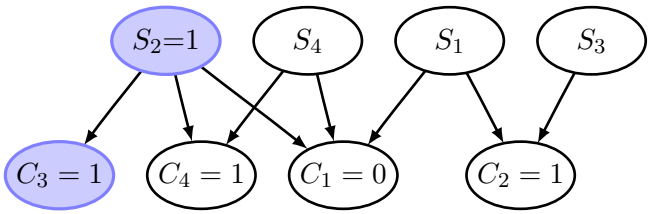


Figura 2.4: O primeiro passo do algoritmo passa por seleccionar uma *codeword* com grau 1. Verifica-se que a *codeword* $C3$ é de grau 1, porque depende apenas do símbolo $S2$. Assim, o símbolo $S2$ é descodificado porque o valor da *codeword* $C3$ é igual ao símbolo $S2$.

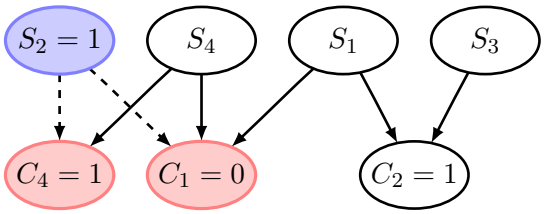


Figura 2.5: De seguida, é necessário encontrar todas as *codewords* que dependem do símbolo $S2$. No grafo, as dependências de símbolos são representadas por ligações entre símbolos e *codewords*. Observa-se que as *codewords* $C1$ e $C4$ dependem do símbolo $S2$.

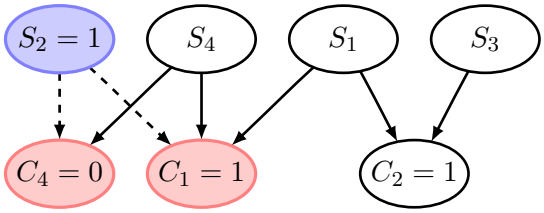


Figura 2.6: Como o valor do símbolo $S2$ já é conhecido, deve-se remover a sua dependência nas *codewords* $C1$ e $C4$. A remoção da dependência é efectuada com a mesma operação usada na codificação, o "ou-exclusivo". O grau das *codewords* após a remoção da dependência é então decrementado. A informação das *codewords* fica então: $C1 = C1 \oplus S2 = 0 \oplus 1 = 1$; $C4 = C4 \oplus S2 = 1 \oplus 1 = 0$.

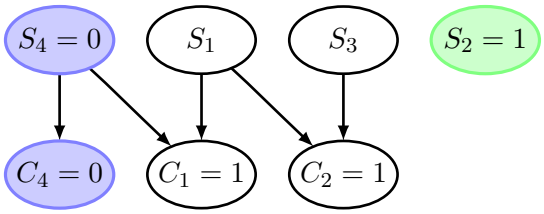


Figura 2.7: Com a remoção da dependência do símbolo $S2$ na *codeword* $C4$, observa-se que esta passou a ter grau 1. O símbolo $S4$ é então descodificado, visto que a *codeword* $C4$ apenas depende deste.

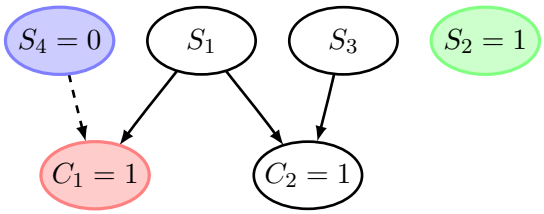


Figura 2.8: Agora, é necessário encontrar todas as *codewords* que dependem do símbolo $S4$. Observa-se que apenas a *codeword* $C1$ depende do símbolo $S4$.

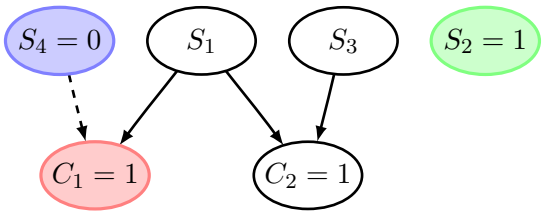


Figura 2.9: É então removida a dependência do símbolo $S4$ na *codeword* $C1$. O resultado da informação da *codeword* fica: $C1 = C1 \oplus S4 = 1 \oplus 0 = 1$.

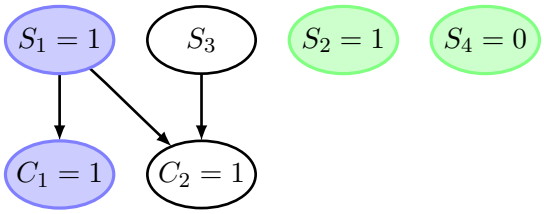


Figura 2.10: Novamente, com a remoção da dependência do símbolo $S4$ na *codeword* $C1$, a *codeword* passou a ter grau 1. O símbolo $S1$ é descodificado, visto que a *codeword* $C1$ apenas depende deste.

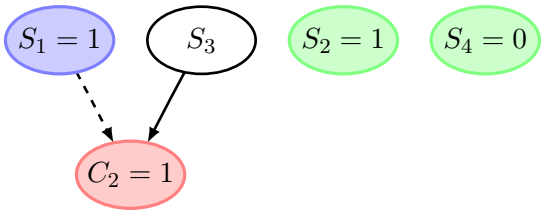


Figura 2.11: São seleccionadas as *codewords* que dependem do símbolo $S1$. Observa-se que apenas a *codeword* $C2$ depende do símbolo $S1$.

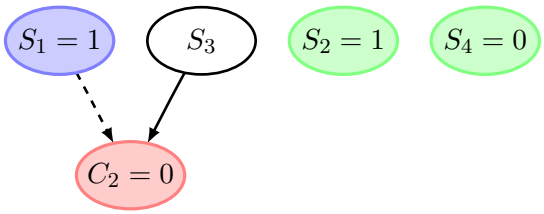


Figura 2.12: É removida a dependência do símbolo $S1$ na *codeword* $C2$. O resultado da informação da *codeword* fica: $C2 = C2 \oplus S1 = 1 \oplus 1 = 0$.

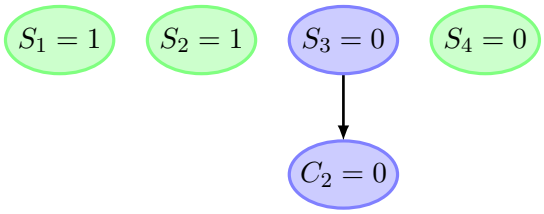


Figura 2.13: O símbolo $S3$ é então descodificado, visto que a *codeword* $C2$ apenas depende deste símbolo.

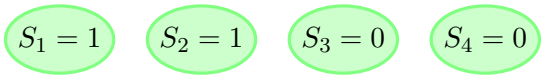


Figura 2.14: Finalmente o processo de descodificação termina, visto que todos os símbolos foram descodificados.

### 2.2.3 Função de Distribuição dos Graus

Quando o código LT foi inicialmente teorizado [20], era composto pela distribuição *ideal soliton*. A distribuição *ideal soliton*, $\rho$, representa o comportamento ideal da probabilidade de selecção de um certo grau, onde no processo de codificação seriam geradas poucas *codewords* com grau 1. No entanto, na prática verificou-se [20, 21] que esta distribuição não funcionava. Aquando a descodificação, não haveria *codewords* de grau 1 suficientes capazes de desencadear a descodificação.

A solução encontrada foi modificar a distribuição *ideal soliton* de modo a compensar o número de *codewords* com grau 1. A esta distribuição deu-se o nome de *robust soliton*. A distribuição *robust soliton*, $\mu$, é composta pela distribuição $\rho$ combinada com uma distribuição de compensação $\tau$. A distribuição $\mu$ depende de dois parâmetros, $c$ e $\delta$. Estes parâmetros são usados para controlar o número de *codewords* com grau 1 que são geradas na codificação.

A figura 2.15, apresenta o histograma da distribuição *ideal soliton*, $\rho$, que reflecte a probabilidade de ocorrência de um determinado grau $d$, usando como parâmetros $k = 10000$, $c = 0.1$ e $\delta = 0.05$, onde $k$ é o número de símbolos da mensagem. Pode observar-se um pico no grau $d = 2$, no histograma da distribuição *ideal soliton*. Isto significa na prática, que cerca de metade das *codewords* criadas terão grau 2, o que é desejável, pelo facto de serem estas *codewords* as que mais contribuem para a descodificação de símbolos. Também se verifica que a probabilidade de ocorrência do grau $d = 1$ é muito baixa, confirmando a ineficiência desta distribuição.

Na mesma figura, é apresentado o histograma da distribuição de compensação $\tau$ que é combinada com a distribuição $\rho$, resultando na distribuição *robust* $\mu$. Como se pode observar, com esta compensação consegue-se um aumento na probabilidade de ocorrência de *codewords* com grau 1. No entanto, também se verifica um pico num grau elevado (neste caso, o grau 83). Por vezes, quando restam alguns símbolos a descodificar, devido à falta de *codewords* que dependem destes símbolos, é mais difícil proceder à sua descodificação. De forma a tentar contornar este problema, são criadas algumas *codewords* com combinações lineares de muitos símbolos.

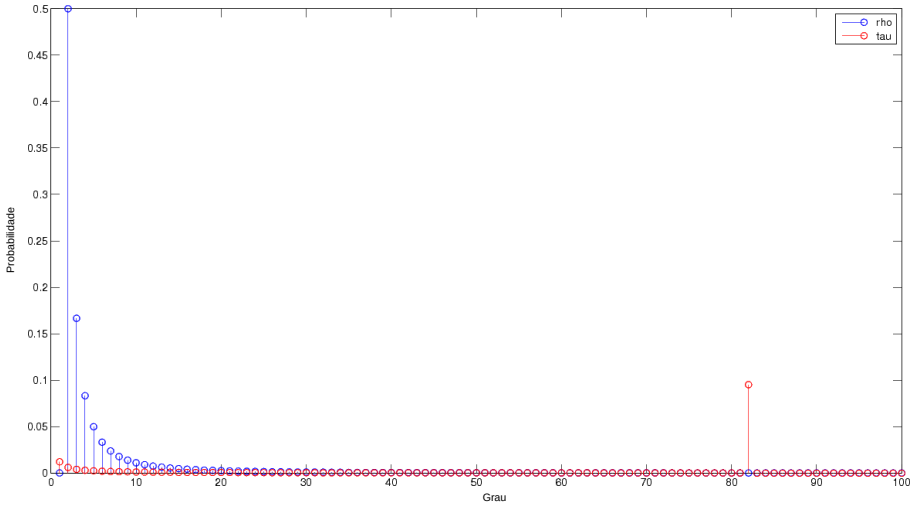


Figura 2.15: Distribuição ideal $\rho$ e a compensação $\tau$, onde a distribuição $\mu = \rho + \tau$. Distribuição para $c = 0.1$, $\delta = 0.05$ e $k = 10000$.

A selecção dos símbolos que são combinados linearmente de forma a criar *codewords*, é feito de forma aleatória. Por esse facto, existe uma probabilidade associada à não selecção de símbolos. Deve-se portanto, estudar a probabilidade de um possível símbolo não ser incluído na codificação das *codewords*. Neste trabalho foi realizadao uma simulação com o objectivo de apresentar uma estatística de símbolos não usados no processo de codificação. A figura 2.16, mostra os resultados obtidos. A simulação revela o número de símbolos não incluídos para $N = 2k$ *codewords* geradas, variando os parâmetros $c$ e $\delta$. Observa-se, que o possível número de símbolos não incluídos é baixo e pode ser controlado. Pode-se, portanto, desprezar este problema.

Num código perfeito, seria sempre possível a descodificação de um ficheiro com $k$ símbolos apenas com $k$ *codewords*. O código LT está longe de ser perfeito, sendo frequentemente necessárias mais *codewords* para o processo de descodificação ser concluído com sucesso. No entanto, foi demonstrado [21] que o código LT pode ser configurado de modo a ser possível reconstruir um ficheiro com $k = 10000$ símbolos, através de $k(1+\varepsilon)$ *codewords*, onde o *overhead* $\varepsilon$, é cerca de 5% de $k$. Na figura 2.17, é apresentada uma estatística do número esperado de *codewords*, necessárias à descodificação de todos os símbolos. A simulação foi realizada em função do parâmetro $\delta$, para $k = 10000$ símbolos e alguns valores de $c$. Ao observar o gráfico, conclui-se que os resultados obtidos em [21] são válidos.

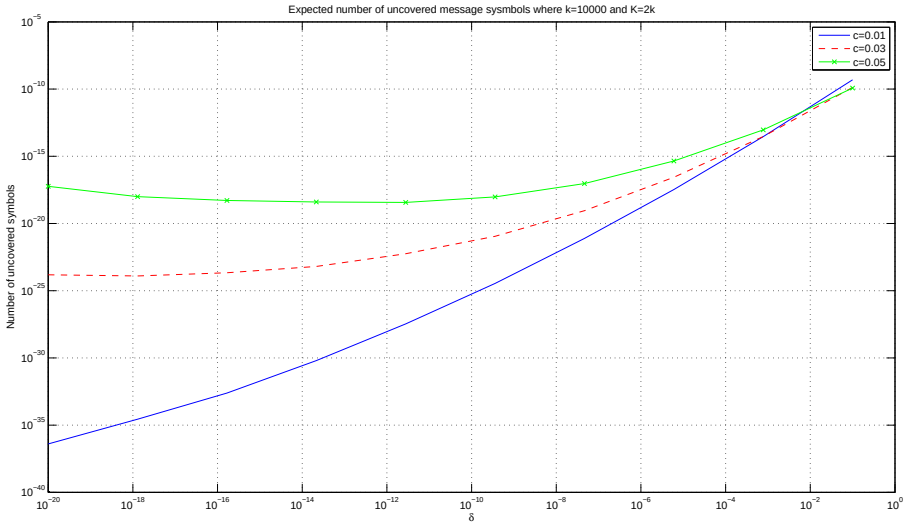


Figura 2.16: Número esperado de símbolos não seleccionados na codificação para vários valores de $c$, com $k = 10000$ e $K = 2k$, onde $k$ é o número de símbolos originais e $N$ o número total de *codewords* geradas na codificação.

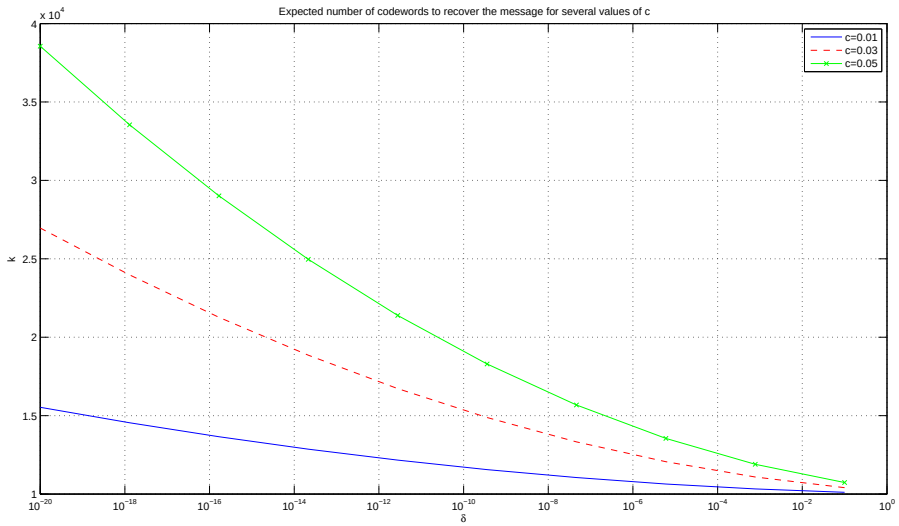


Figura 2.17: Número esperado de *codewords* de modo a descodificar todos os símbolos com $k = 10000$.

$$M = \begin{bmatrix} 1 & 0 & 0 & 8 & 3 \\ 0 & 0 & 4 & 1 & 0 \\ 1 & 5 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figura 2.18: Matriz $M$.

## 2.3 Estruturas de Dados para Armazenar Matrizes Esparsas

Na secção 2.2 foi mostrado que a codificação de códigos LT produz uma matriz $A$ de codificação que é necessária para o processo de descodificação. A matriz $A$ é binária e de natureza esparsa, ou seja, em média existem poucos índices diferentes de 0 na matriz. No processo de descodificação, é apenas necessário ter conhecimento dos índices que são diferentes de 0. Por outro lado, o armazenamento de toda a matriz $A$ não é eficiente, especialmente quando se trabalha com muita informação. É então importante estudar estruturas de dados computacionais que possam armazenar matrizes de natureza esparsa.

Nas subsecções seguintes são apresentadas algumas estruturas de dados, normalmente encontrados na literatura [26], capazes de armazenar matrizes esparsas.

### 2.3.1 *Coordinate Storage Matrix*

A estrutura *Coordinate Storage Matrix* (CSM), é a forma mais genérica para representar uma matriz esparsa. É também a mais simples, no entanto, não é muito eficiente em termos de memória e computação.

A matriz é armazenada usando 3 vectores. O primeiro vector, *val*, armazena os valores não nulos da matriz. O segundo vector, *row_ind*, armazena o índice da linha da matriz para cada elemento do vector *val*. O último vector, *col_ind*, armazena o índice da coluna da matriz para cada elemento do vector *val*.

Como exemplo, considere-se a matriz $M$ definida em 2.18. Na tabela 2.2 é apresentada a matriz $M$ definida em 2.18 usando a estrutura CSM.

| $k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $val[k]$ | 1 | 8 | 3 | 4 | 1 | 1 | 5 | 1 | 3 | 1 | 1 |
| $row\_ind[k]$ | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 |
| $col\_ind[k]$ | 0 | 3 | 4 | 2 | 3 | 0 | 1 | 4 | 1 | 0 | 3 |

Tabela 2.2: Matriz $M$ representada com a estrutura CSM. Note-se que $k = 0, 1, 2, ...nzz - 1$, sendo $nzz$ o número de valores não nulos da matriz.

### 2.3.2 *Compressed Column (Row) Storage*

As estruturas de dados *Compressed Row Storage* (CRS) e *Compressed Column Storage* (CCS), são formatos genéricos de armazenamento de matrizes, porque não fazem assumpções quanto à estrutura esparsa da matriz. Apenas armazenam os elementos não nulos da matriz. Apesar de serem as estruturas mais usadas na prática, não são as mais eficientes para realizar operações matemáticas em matrizes, porque recorrem a um método de endereçamento indirecto para aceder a um valor. Os dois métodos são identicos, a diferença reside no facto de

a estrutura CCS armazenar a matriz no seu formato transposto. O *software* de computação cientifica *Matlab*, usa estruturas do tipo CCS para o armazenamento de matrizes esparsas.

A estrutura de dados é composta por três vectores. O primeiro vector, *val*, armazena os valores não nulos da matriz. A matriz é varrida da esquerda para a direita e de cima para baixo. O segundo vector, *col_ind* armazena o índice da coluna na matriz, para cada elemento do vector *val*. O último vector, *row_pt* armazena o índice no vector *val* onde começa uma nova linha da matriz. Na tabela 2.3, é apresentada a matriz $M$ definida em 2.18, usando o método CRS.

| $k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $val[k]$ | 1 | 8 | 3 | 4 | 1 | 1 | 5 | 1 | 3 | 1 | 1 |
| $col\_ind[k]$ | 0 | 3 | 4 | 2 | 3 | 0 | 1 | 4 | 1 | 0 | 3 |

| $i$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $row\_pt[i]$ | 0 | 3 | 5 | 8 | 9 |

Tabela 2.3: Matriz $M$ representada com a estrutura CRS. Note-se que $k = 0, 1, 2, ...nzz - 1$ e $i = 0, 1, 2, ...n - 1$, sendo $nzz$ o número de valores não nulos e $n$ o número de linhas da matriz.

Em vez de varrer as linhas de uma matriz, pode-se varrer as colunas. Ou seja, usar o método CRC para a transposta da matriz. Este método é denominado CCS. Na tabela 2.4, é apresentada a matriz $M$ definida em 2.18, usando o método CCS.

### 2.3.3 Listas Duplamente Ligadas

As listas ligadas são estruturas de dados que são muito usadas na computação. Existem vários tipos de listas ligadas, como as listas ligadas simples e as listas duplamente ligadas. Normalmente apenas diferem no número de ligações para elementos da estrutura. Para armazenar matrizes, por vezes é conveniente ter uma estrutura que permita varrer a matriz tanto por linhas, como por colunas. Com esse objectivo, pode-se usar uma lista duplamente ligada onde cada elemento da matriz é armazenada num nó. Um nó, é uma estrutura composta pelo índice da linha, índice da coluna e valor do elemento. Um nó também é composto por referências para o próximo elemento na mesma linha e para próximo elemento na mesma coluna. No entanto, neste tipo de estrutura não é possível aceder a um determinado índice directamente, sendo necessário varrer a estrutura atá ao índice desejado.

Como exemplo, considere-se a matriz $N$ definida em 2.19. Na figura 2.20 é apresentada a matriz $N$ usando uma lista duplamente ligada.

| $k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $val[k]$ | 1 | 1 | 1 | 5 | 3 | 4 | 8 | 1 | 1 | 3 | 1 |
| $row\_pt[k]$ | 0 | 2 | 4 | 2 | 3 | 1 | 0 | 1 | 4 | 0 | 2 |

| $i$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $col\_ind[i]$ | 0 | 3 | 5 | 6 | 9 |

Tabela 2.4: Matriz $M$ representada com a estrutura CCS. Note-se que $k = 0, 1, 2, ...nzz - 1$ e $i = 0, 1, 2, ...m - 1$, sendo $nzz$ o número de valores não nulos e $m$ o número de colunas da matriz.

$$N = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 5 \\ 0 & 6 & 7 \end{bmatrix}$$

Figura 2.19: Matriz N.

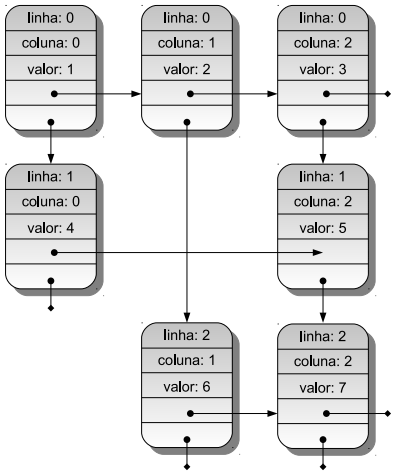


Figura 2.20: Representação da matriz $N$ usando uma lista duplamente ligada.

### 2.3.4 Outras Estruturas de Dados

Existem outras estruturas de dados para armazenar matrizes esparsas. No entanto, são especializadas em características específicas que a matriz pode apresentar. Embora algumas destas estruturas possam comportar-se como uma estrutura genérica de armazenamento, a sua eficiência não é muito boa.

***Compressed Diagonal Storage (CDS)*** Esta estrutura é útil caso a natureza esparsa da matriz demonstre uma largura de banda ligeiramente constante em cada linha. Diz-se que uma matriz tem largura de banda, quando existem duas constantes não negativas $p$ e $q$, onde $a_{ij} \neq 0$ apenas se $i - p \leq j \geq i + q$. Pode-se aproveitar essa característica para guardar sub-diagonais da matriz em locais consecutivos. Assim, para além de se conseguir eliminar o vector que identifica a linha e a coluna, consegue-se guardar os elementos diferentes de 0 de forma a facilitar a computação do produto entre matrizes. Por vezes, este tipo de formato armazena zeros e até valores que não estão sequer na matriz.

***Jagged Diagonal Storing (JDS)*** Estas estruturas de dados são espacialmente mais eficientes e mais genéricas do que as estruturas CDS, mas com o custo de serem menos eficientes a procurar elementos. São normalmente usadas para implementar sistemas iterativos em processadores vectoriais. Uma versão simplificada do JDS é o chamado formato *Itpack*. Este formato assume que existem $N_d$ elementos por linha, onde $N_d$ é um valor pequeno. A estrutura de dados é composta por dois vectores com dimensão $n * N_d$, onde $n$ é um número suficiente grande para armazenar todos os elementos da matriz. O primeiro vector *coef* armazena valores não nulos da matriz, sendo cada linha completa com zeros onde necessário. O segundo vector *jcoef* armazena índices de coluna

de cada elemento no vector *coef*. Os índices de coluna para elementos nulos do vector *coef*, são por convenção, iguais ao índice da linha.

***Block Compressed Row Storage (BCRS)*** Se a matriz é composta por blocos de elementos diferentes de zero, segundo um determinado padrão, pode-se modificar as estruturas CRS ou CCR para explorar esse facto. Consegue-se reduzir a eficiência espacial, assim como o endereçamento indirecto é reduzido em matrizes com blocos grandes.

***Skyline Storage*** Para matrizes com largura de banda variável. Podem ser usadas para processar blocos de diagonais em factorizações de blocos da matriz.

## 2.4 NFS

O *Network File System* (NFS) [30] é um protocolo de comunicações entre sistemas computacionais, desenvolvido pela Sun Microsystems. É usado por omissão na partilha de ficheiros em sistemas baseados em Unix. No entanto, existem implementações do NFS virtualmente para qualquer plataforma. Actualmente, existem 4 versões do protocolo NFS. No entanto, a primeira versão não está disponível e as implementações da versão 4 do protocolo estão ainda em desenvolvimento. No âmbito do trabalho desta dissertação foi optado pela versão 2 [30] do protocolo devido à sua banalidade e simplicidade, existindo implementações para todas as arquitecturas computacionais. O uso do termo NFS nas subsecções seguintes refere-se à versão 2 do protocolo.

### 2.4.1 Características do NFS

O NFS proporciona acesso remoto a ficheiros partilhados em rede. Este acesso é totalmente transparente para o utilizador. O protocolo NFS foi desenvolvido com a intenção de ser portável entre diferentes máquinas, sistemas de operação e arquitecturas de redes. Pode ser transportado usando os protocolos TCP e UDP. A portabilidade do protocolo deve-se ao facto do NFS usar o protocolo *Remote Procedure Call* (RPC) [29] e o *eXternal Data Representation* (XDR) [28]. O protocolo RPC foi desenvolvido com a programação remota em mente, um programa pode fazer uma chamada a uma sub-rotina que é executada num sistema computacional remoto. O protocolo XDR é um *standard* usado para codificar e representar informação. É útil para transmitir informação entre sistemas computacionais com arquitecturas distintas.

O NFS segue uma arquitectura cliente/servidor. O servidor NFS é responsável pela exportação de um sistema de ficheiros remoto que o cliente pode montar no seu sistema computacional. O cliente pode interagir com o sistema de ficheiros remoto do modo usual, como se tratasse de um sistema de ficheiro local. O protocolo NFS foi desenvolvido com a intenção de ser o mais *stateless* possível, ou seja, o servidor não necessita de armazenar o estado de operação dos clientes, de modo a funcionar correctamente. Em caso de falha, o servidor é reiniciado sem que os clientes percebam que ocorreu uma falha.

Na realidade o NFS é composto por dois protocolos: o protocolo NFS e o protocolo MOUNT. O protocolo MOUNT permite ao servidor NFS atribuir privilégios de acesso remoto a um conjunto restrito de clientes NFS. Por sua vez, o protocolo NFS permite ao servidor NFS fornecer aos clientes uma interface remota de gestão de ficheiros muito semelhante à existente nos sistemas de operação usada para gerir os sistema de ficheiros locais.

| Procedimento | Descrição |
|---|---|
| NULL | Não faz nada. Útil para testar a conexão com o servidor. |
| MOUNT | Permite a um cliente com permissões montar um sistema de ficheiros. |
| DUMP | Retorna uma lista com todos os sistemas de ficheiros montados. |
| UMOUNT | Permite a um cliente desmontar um sistema de ficheiros montado. |
| UNMTALL | Permite a um cliente desmontar todos os sistema de ficheiros montados. |
| EXPORT | Retorna informação sobre os sistemas de ficheiros disponíveis. |

Tabela 2.5: Interface de procedimentos do protocolo MOUNT.

| Procedimento | Descrição |
|---|---|
| NULL | Não faz nada. Útil para testar a conexão com o servidor. |
| GETATTR | Retorna os atributos de um ficheiro. |
| SETATTR | Associa a um ficheiro atributos. |
| ROOT | Obsoleto. É usado o procedimento MOUNT do protocolo MOUNT. |
| LOOKUP | Procura e retorna os atributos de um ficheiro num directorio. |
| READLINK | Lê o caminho (*path*) dentro de um *link* simbólico. |
| READ | Lê um bloco de tamanho fixo de um ficheiro. |
| WRITECACHE | Usado apenas no protocolo NFSv3. |
| WRITE | Escreve um bloco de tamanho fixo para um ficheiro. |
| CREATE | Cria um ficheiro novo num directorio. |
| REMOVE | Remove um ficheiro num directório. |
| RENAME | Renomeia um ficheiro ou directório. |
| LINK | Cria um *hard link* para um ficheiro. |
| SYMLINK | Cria um *link* simbólico para um ficheiro. |
| MKDIR | Cria um directório novo. |
| RMDIR | Remove um directório vazio. |
| READDIR | Lê os nome de todos os ficheiros de um directório. |
| STATFS | Retorna informação sobre o sistema de ficheiros exportado. |

Tabela 2.6: Interface de procedimentos do protocolo NFS.

### 2.4.2 Protocolo MOUNT

O protocolo MOUNT é responsável por proporcionar uma interface remota que permite a um cliente montar um sistema de ficheiros remoto. Esta interface é composta por procedimentos que são invocados pelo cliente. Na tabela 2.5 apresentam-se os procedimentos definidos no protocolo MOUNT.

### 2.4.3 Protocolo NFS

O protocolo NFS é responsável por proporcionar uma interface remota que permite a um cliente gerir um sistema de ficheiros remoto. Esta interface é composta por procedimentos que são semelhantes às funções de gestão de um sistema de ficheiros local no cliente. Na tabela 2.6 apresentam-se os procedimentos definidos no protocolo NFS.

### 2.4.4 Modo de Funcionamento

Nesta subsecção é descrito o modo de funcionamento do NFS. Imagine-se que um utilizador deseja ver um documento que está num computador remoto. Se esse documento estiver num sistema de ficheiros partilhado usando o protocolo NFS, o utilizador pode montar o sistema de ficheiros remoto no seu computador cliente e aceder ao ficheiro de forma transparente. Para se conseguir esta camada de abstracção é necessária uma interacção entre diversos componentes. A figura 2.21 visualiza os componentes que interagem quando o cliente envia um pedido de leitura de um ficheiro ao servidor. Assumindo que o cliente já tem o sistema de ficheiros remoto montado, a figura 2.21 é descrita da seguinte forma:

1. O cliente envia o pedido de leitura do ficheiro ao serviço local NFS. O serviço NFS cliente é um modulo do *kernel* do sistema de operação.
2. O serviço NFS cliente usa o RPC para converter o pedido num pacote de dados que o servidor NFS consegue perceber.
3. O pacote de dados é serializado e transportado usando UDP/TCP sobre IP.
4. Ao chegar ao lado do servidor, o pacote é deserializado e passado ao servidor NFS.
5. O servidor NFS processa o pedido do cliente e lê o ficheiro do disco local.
6. O caminho contrário é usado para mandar o ficheiro até ao cliente.

Falta responder à questão de como o cliente consegue aceder a um ficheiro num sistema de ficheiros remoto. Quando o cliente monta um sistema de ficheiros remoto, o servidor retorna um *File Handle* do directório raiz a exportar. Um *File Handle* é um identificador único que referencia um ficheiro ou directório. Estes identificadores são trocados entre o cliente e o servidor sempre que um ficheiro ou directório é referenciado num pedido. Assim, consegue-se evitar o uso de caminhos completos para ficheiros.

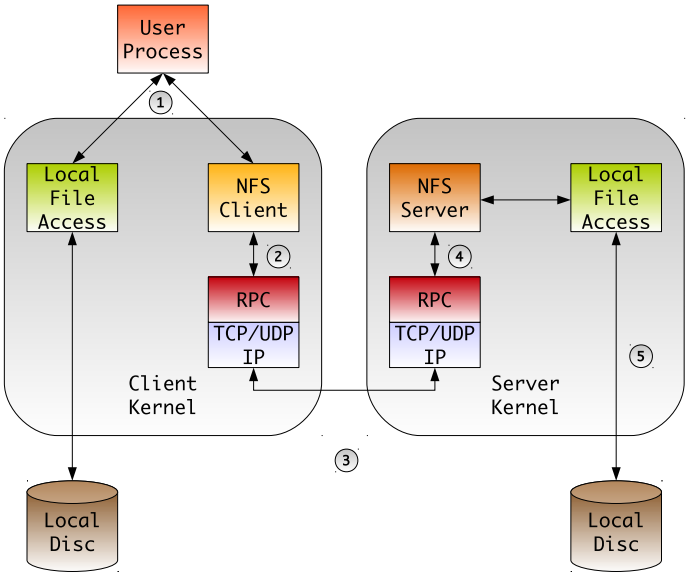


Figura 2.21: Modo de funcionamento do NFS. (1) O cliente NFS faz um pedido ao servidor NFS. (2) O pedido do cliente é convertido para o formato do protocolo RPC. (3) O pedido é enviado para o servidor NFS via IP sobre UDP/TCP. (4) O pedido do cliente é convertido para o formato original. (5) O servidor NFS processa o pedido e acede à informação.

# Capítulo 3

# Sistemas de Armazenamento

## 3.1 Sumário

Neste capítulo são apresentados alguns sistemas de ficheiros que existem actualmente e que se relacionam com o trabalho desenvolvido nesta dissertação.

Para começar, é apresentado o CFS. Este sistema de ficheiros tem como objectivo o armazenamento de informação na forma cifrada. Tal como o sistema NFS Fountain, foi desenvolvido através de uma implementação de um servidor NFS em *loopback*.

Seguidamente é apresentado o GFS. O GFS é um sistema de ficheiros distribuído desenvolvido na Google de modo a satisfazer as necessidades de armazenamento e processamento de grandes quantidades de informação, que são necessárias pelos serviços fornecidos na Google. Tal como o sistema NFS Fountain, procura resolver os problemas dos sistema de ficheiros actuais, como a reabilidade da informação armazenada. Consegue-o através da replicação da informação em vários servidores.

Finalmente é apresentado o sistema Avalanche. O sistema Avalanche é um projecto de investigação da Microsoft que pretende desenvolver um sistema de ficheiros distribuído baseado na tecnologia P2P. Assim como o sistema NFS Fountain, este sistema recorre a uma variante dos códigos Fountain de modo a codificar a informação, obtendo assim reabilidade no armazenamento de informação.

## 3.2 *Cryptographic File System*

O *Cryptographic File System* (CFS) [8] é um sistema de ficheiros que usa serviços de encriptação para armazenar a informação em disco. Suporta um armazenamento de informação seguro através da interface de sistema de ficheiros do Unix (VFS), conseguindo usar qualquer tipo de sistema de ficheiros, local ou remoto, desde que suportado pelo *kernel* do sistema de operação. Para encriptar a informação, os utilizadores podem associar uma chave criptográfica aos directórios que desejam proteger. Os ficheiros dentro desses directórios são encriptados e desencriptados de forma transparente, com a chave associada, sem qualquer tipo de intervenção do utilizador. É seguro na medida em que a informação é sempre armazenada e/ou transmitida no seu formato codificado.

### 3.2.1 Arquitectura

O CFS foi implementado em *user level*, comunicando com o *kernel* do Unix através da interface do protocolo NFS, que foi apresentada na secção 2.4. Em todos os clientes é executado um servidor NFS especial, denominado CFS Daemon.

O CFS Daemon corre na interface *loopback* (*localhost*) e é responsável pelo processamento dos pedidos do sistema de ficheiros CFS. Foi desenvolvido através de uma implementação do protocolo NFS, o qual foi estendido com mais procedimentos. Esses procedimentos adicionais têm como objectivo adicionar, remover e controlar os directórios sujeitos a encriptação.

O cliente NFS é responsável pela inicialização da comunicação com o servidor CFS Daemon. Para aumentar a eficiência do sistema, o cliente armazena em cache os blocos de informação provenientes do servidor. A integridade da cache é gerida através de um simples protocolo que é responsável pela sua consistência.

Inicialmente, a raiz do sistema de ficheiros CFS apresenta um directório vazio. O cliente pode inferir um pedido ao CFS Deamon de modo a encriptar um directório. A informação desse pedido é traduzido no caminho absoluto do directório a encriptar, o nome do directório que vai ser associado ao outro directório no CFS, e a respectiva chave criptográfica. Na raiz do CFS é então criado um directório com o nome associado proveniente do pedido. O acesso a este directório é apenas permitido ao utilizador que efectuou o pedido. Todos os pedidos de gestão de ficheiros dentro desse directório são efectuados através dos procedimentos base do protocolo NFS. A informação é então encriptada e armazenada no directório. Para cada ficheiro acedido, o CFS Deamon cria um *File Handle* único que é usado para o cliente referenciar o mesmo. Um pedido de acesso à informação encriptada, caso o utilizador seja autenticado, obriga à desencriptação e ao envio da informação para o cliente.

A figura 3.1 mostra a arquitectura e o processo de comunicação entre os seus componentes e o *kernel* do sistema de operação.

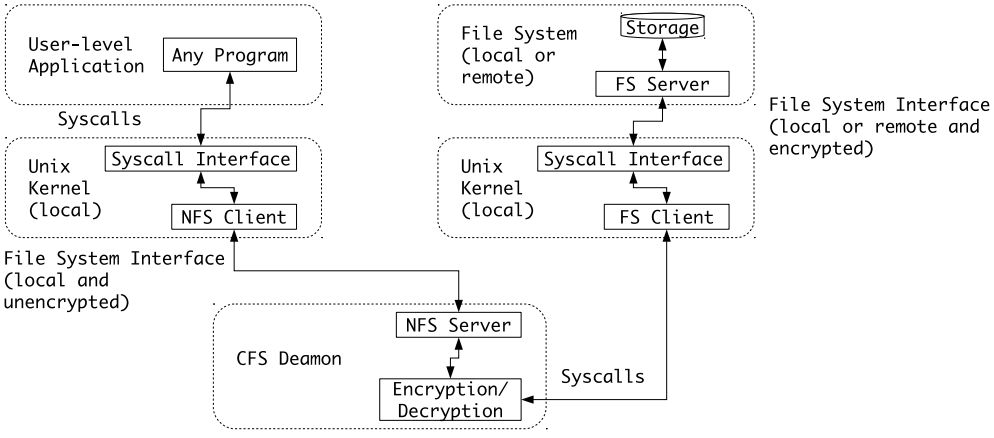


Figura 3.1: Arquitectura do CFS [8].

## 3.3 *Google File System*

O *Google File System* (GFS) [12] é um sistema de ficheiros distribuído e escalável, optimizado para aplicações que interagem com grandes blocos de informação. É usado na Google

como a plataforma de armazenamento e processamento de informação dos serviços que usam grandes blocos de informação. O GFS corre num aglomerado de centenas de *terabytes* de armazenamento, distribuídos por milhares de discos em cerca de 1 milhar de sistemas computacionais. É acedido por centenas de clientes. Partilha muitos dos objectivos de sistemas de ficheiros distribuídos como o desempenho, escalabilidade, reabilidade e disponibilidade.

### 3.3.1 Arquitectura

Um aglomerado GFS consiste num servidor central denominado *Single Master* e múltiplos servidores denominados *Chunk Servers*, que reportam a este. O servidor *Single Master* mantém a meta-data de todo o sistema de ficheiros, nomeadamente a informação de controlo de acesso e a localização da informação. A informação é guardada em blocos de tamanho fixo em *Chunk Servers*. Obtém-se fiabilidade com a replicação de blocos em vários *Chunk Servers*.

O cliente GFS implementa uma API de sistema de ficheiros e comunica com os servidores *Single Master* e *Chunk Servers* para ler e escrever informação. A comunicação com o *Single Master* é estritamente para operações sobre a meta-data da informação. A transmissão de informação propriamente dita é efectuada directamente entre o cliente e os *Chunk Servers*.

A figura 3.2 mostra as interacções entre os componentes arquitectura do GFS para uma simples leitura de informação. Inicialmente, o cliente envia um pedido ao *Single Master* com o nome do ficheiro e com um identificador do bloco. O identificador do bloco é necessário para o servidor *Single Master* identificar os *Chunk Servers* que armazenam blocos que correspondem ao ficheiro. O *Single Master* responde então com os identificadores dos *Chunk Servers* que armazenam o bloco. O cliente guarda esta meta-data em cache. O cliente faz depois um pedido a um *Chunk Server*, normalmente o mais próximo, para leitura do bloco. Enquanto a meta-data em cache não expirar, não será necessária a comunicação entre o cliente e o *Single Master*.

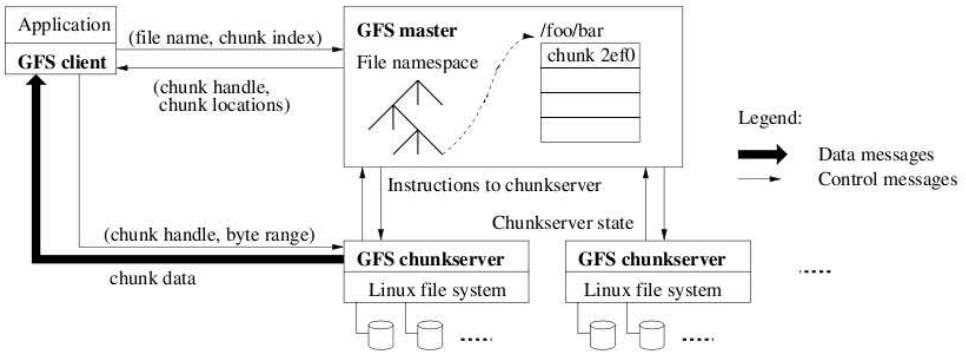


Figura 3.2: Arquitectura do GFS [12].

## 3.4 Avalanche

O sistema Avalanche [15, 13, 14] é um projecto de investigação da Microsoft. É um sistema P2P que tem como objectivo ser mais escalável e mais eficiente em termos de largura de banda do que os outros sistemas P2P. Actualmente, é aplicado num sistema da Microsoft denominado MSCD (*Microsoft Secure Content Downloader*).

A arquitectura do sistema Avalanche é muito semelhante à do conhecido sistema BitTorrent. No entanto, pretende ser mais eficiente em termos da distribuição de informação pelos sistemas computacionais (nós) que fazem parte de uma rede Avalanche. Em vez de enviar blocos de informação originais como no sistema BitTorrent, usa uma variante dos códigos Fountain, denominado *Network Coding*, para codificar os blocos de informação. O uso do *Network Coding* assegura, em princípio, uma distribuição eficiente da informação por toda a rede Avalanche.

### 3.4.1 *Network Coding*

O *Network Coding* [11] pode ser entendido como uma generalização dos códigos Fountain. Como acontece nos códigos Fountain, a informação inicialmente é dividida em $k$ símbolos iguais, e na codificação, estes símbolos são combinados linearmente para formar *codewords*. A diferença provem do facto de a codificação poder também ser aplicada às *codewords*. Por exemplo, imagine-se que um nó da rede Avalanche disponibiliza informação para os outros nós. Esse nó, denominado servidor, codifica os símbolos da informação em *codewords* e envia-as para os outros nós. Os outros nós, ao receberem as *codewords*, verificam se já têm posse de *codewords* relativas à mesma informação, e em caso afirmativo produzem *codewords* a partir da combinação linear dessas *codewords*, transmitindo-as para outros nós. Assim, consegue-se teoricamente uma melhor distribuição de *codewords* por todos os nós da rede, visto que não é necessário um nó ter a informação original para codificar e enviar *codewords*. A figura 3.3 mostra uma ilustração da propagação de *codewords* pelos nós de uma rede Avalanche. No entanto, existem estudos [10] que demonstram que o uso de *Network Coding* não torna mais eficiente a distribuição de informação num sistema distribuído P2P.

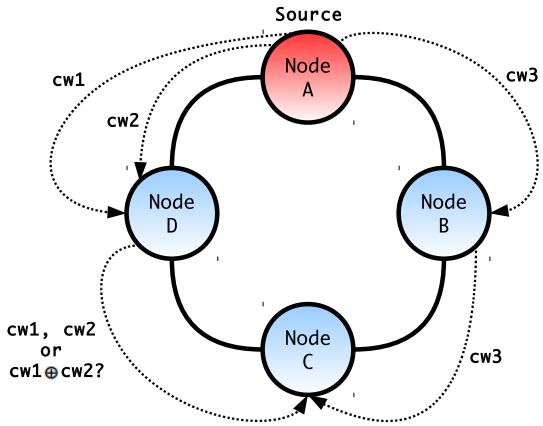


Figura 3.3: Propagação de *codewords* usando *Network Coding* [**?**].

### 3.4.2 Arquitectura

O sistema Avalanche é constituído por 3 entidades: Nó(s), o *Registar*, e o *Logger*.

**Nó:** Os nós são responsáveis pelo armazenamento, codificação e envio de informação para outros nós. Os nós são agrupados em grupos de nós (menos de 10), pelo que um nó, só envia *codewords* para nós que pertencem ao seu grupo. Inicialmente, quando um nó

pretende partilhar informação com outros nós, este passa a "semear" (*seed*) a informação para os outros nós. O processo de *seed* consiste no processo de codificação de *codewords*, através de combinações lineares dos símbolos originais da informação, e no envio destas para os nós. Os restantes nós ao receberem estas *codewords* são responsáveis pela codificação de novas *codewords* através da combinação linear das *codewords* existentes, e pelo seu envio para outros nós.

***Registar*:** O *registar* proporciona a descoberta de novos nós. Os nós que estão activos reportam periodicamente ao registar, pelo que este responde com um conjunto aleatório de nós que tenham poucos vizinhos.

***Logger*:** O *logger* é um ponto de agregação para as mensagens de operação dos nós e do *registar*. Todos os nós reportam estatísticas detalhadas ao *logger*, de modo a avaliar a eficiência do sistema.

De todos os componentes o nó é o mais complexo e a sua funcionalidade está dividida em 2 componentes: o *Network Transport* e o *Content Manager*.

***Network Transport*:** O *network transport* mantém conexões para outros nós, com a finalidade de enviar *codewords*. São usadas duas ligações por par de nós, uma em cada direcção. Cada nó mantém cerca de 6 a 8 conexões com os seus nós vizinhos. Periodicamente, um nó abandona um nó vizinho ao acaso, de modo a evitar que se formem nós isolados na rede.

***Content Manager*:** O *content manager* é responsável pela codificação, descodificação, validação e persistência de informação.

# Capítulo 4

# Arquitectura do Sistema NFS Fountain

## 4.1 Sumário

Actualmente, pode-se armazenar ficheiros nos discos de computadores pessoais e até em servidores remotos. No entanto, essa informação corre o risco de ser perdida caso do disco ou o servidor avariem. Para garantir a fiabilidade da informação é necessário recorrer ao seu armazenamento com redundância em múltiplos sistemas de armazenamento. É então importante ter atenção ao aspecto da escalabilidade e do custo que podem comprometer o sistema. O sistema NFS Fountain pretende resolver estes problemas. O NFS Fountain é um sistema de ficheiros distribuído com suporte a falhas de servidores, que recorre a códigos LT para codificar a informação original. Esta codificação cria *codewords* que são distribuídas em diversos servidores. O código LT garante com grande probabilidade a reconstrução da informação original através de um subconjunto das *codewords* criadas. É possível a recuperação da informação, com grande probabilidade, mesmo se alguns servidores falharem.

Quando a informação é armazenada no NFS Fountain, fica em cache no disco local do servidor. Enquanto a informação estiver em cache, o servidor NFS Fountain comporta-se como um servidor NFS genérico. No entanto, quando a informação em cache não é acedida há um tempo predefinido, é codificada e retirada da cache. A codificação é feita através do código LT. As *codewords* criadas na codificação, são enviadas para outros servidores, denominados Servidores de *Codewords*, que são responsáveis pelo seu armazenamento. Agora, quando se tenta aceder à informação, o sistema NFS Fountain verifica que não está presente em cache e procede à sua descodificação. No processo de descodificação, são efectuados pedidos aos servidores de *codewords*, com o objectivo de recuperar as *codewords* que referem a informação pretendida. Após a descodificação terminar, a informação volta a ser armazenada em cache.

Neste capítulo é apresentada a arquitectura do sistema NFS Fountain, sendo descritos os componentes que o compõem.

## 4.2 Arquitectura Geral

O sistema NFS Fountain é constituído essencialmente por 5 entidades: Cliente(s), Servidor NFS, *Deamon* MRU, Codificador e Descodificador de *Codewords*, Servidor(es) de *Codewords*

e Cliente do Servidor de *Codewords*. A figura 4.1, mostra a arquitectura do sistema NFS Fountain.

**Cliente:** O cliente é um cliente NFS genérico, como foi caracterizado na secção 2.4. Na realidade, para a construção do sistema NFS Fountain, não foi necessário desenvolver o cliente NFS, visto que a implementação do servidor NFS funciona com as implementações nativas dos clientes NFS. Assim, garante-se que qualquer sistema computacional que apresente nativamente um cliente NFS, consegue usar o sistema NFS Fountain. O cliente é responsável por montar o sistema de ficheiros remoto exportado pelo servidor NFS, e por efectuar operações de gestão de ficheiros sobre este.

**Servidor:** O servidor NFS desenvolvido para o sistema NFS Fountain, apresenta todas as operações e características de um servidor NFS genérico, como caracterizado na secção 2.4. Na verdade, pode funcionar como um simples servidor NFS. A diferença mais notável deve-se ao facto de funcionar em *user space*, enquanto que uma implementação nativa é realizada a nível do *kernel*. Como funciona em *user space*, o seu desenvolvimento foi relativamente simples. No entanto, apresenta a desvantagem de ser mais lento do que se fosse implementado no *kernel* do sistema de operação. O servidor NFS do sistema NFS Fountain, foi desenvolvido de modo a armazenar e processar informação referente ao acesso de ficheiros em cache. Sempre que um ficheiro é acedido, o servidor NFS armazena a meta-data referente à sua data de acesso. Basicamente, é guardada uma lista MRU (*Most Recently Used*), dos ficheiros que estão presentes em cache. Esta lista contém todos os ficheiros que estão em cache, pelo que estão presentes no disco local do servidor. Se uma referência a um ficheiro não existir na lista MRU, significa que esse ficheiro está armazenado remotamente no formato codificado em Servidores de *Codewords*.

***Deamon* MRU** O *Deamon* MRU é um serviço que corre em paralelo com o servidor NFS. O seu papel é olhar periodicamente para a lista MRU e verificar se existem ficheiros que não foram acedidos há um tempo predefinido. Caso isso aconteça, o sistema NFS Fountain codifica e remove esses ficheiros da cache.

**Codificador e Descodificador de *Codewords*:** O Codificador e Descodificador de *Codewords* (CDC) é o componente responsável pela codificação e descodificação das *codewords* referentes aos ficheiros. É usado o código LT no processo de codificação.

**Servidor de *Codewords*:** O componente Servidor de *Codewords* (SC), representa os servidores que são responsáveis pelo armazenamento e fornecimento de *codewords*, que são o resultado da codificação de ficheiros.

**Cliente do Servidor de *Codewords*:** O componente Servidor de *Codewords* (CSC) tem como objectivo fornecer uma interface de serviços que o SC disponibiliza. O CDC usa esta interface de modo a processar de envio e recepção de *codewords* para os SCs.

### 4.2.1 Modo de Operação

De modo perceber o modo de funcionamento do sistema NFS Fountain, recorre-se a dois exemplos de caso de uso. O primeiro exemplo, descreve a interacção entre os componentes do sistema NFS Fountain, para um pedido de armazenamento de informação por parte do cliente. Os passos descritos de seguida, podem ser seguidos visualmente através da figura 4.1.

1. Inicialmente, o cliente monta o sistema de ficheiros remoto exportado pelo servidor NFS do sistema NFS Fountain. O cliente envia um ficheiro de modo a ser armazenado no servidor.

2. O ficheiro é armazenado em cache no servidor NFS. O servidor NFS actualiza a meta-informação relativa ao último tempo de escrita do ficheiro na lista MRU. Note-se que caso o utilizador faça um pedido ao servidor NFS de leitura, o ficheiro é acedido directamente da cache.

3. Periodicamente, o *Deamon* MRU verifica se o ficheiro em cache já não é acedido há algum tempo. Caso isso aconteça, o CDC é invocado de modo a codificar o ficheiro.

4. O CDC lê a informação em cache de modo a codificar o ficheiro. Após a codificação, o ficheiro original é eliminado da cache.

5. Ao codificar o ficheiro, o CDC envia as *codewords* criadas para os Servidores de *Codewords* através da interface disponibilizada pelo CSC. As *codewords* ficam armazenadas nos discos locais dos servidores.

O segundo exemplo, descreve a interacção entre os componentes do sistema NFS Fountain, para um pedido de leitura de um ficheiro que não está presente em cache. Os passos descritos de seguida podem ser seguidos visualmente através da figura 4.2.

1. O cliente monta o sistema de ficheiros remoto exportado pelo servidor NFS do sistema NFS Fountain e envia um pedido de leitura de um ficheiro ao servidor NFS.

2. O servidor NFS verifica que o ficheiro não está na cache e invoca o CDC.

3. O CDC pede *codewords* relativas ao ficheiro aos Servidores de *Codewords*. Os Servidores de *Codewords* respondem se tiverem *codewords*.

4. O ficheiro descodificado é colocado novamente em cache.

5. Finalmente o Servidor NFS lê o ficheiro da cache e envia o ficheiro ao cliente.

## 4.3 Servidor NFS

Como já foi referido, o servidor NFS desenvolvido para o sistema NFS Fountain apresenta todas as características de um servidor NFS genérico. Nesta secção, são apresentadas as características que o distinguem e o seu relacionamento com os restantes componentes do sistema NFS Fountain.

### 4.3.1 File Handle

O *File Handle*, como já foi referido, é um identificador único que é transmitido nos pedidos entre o cliente e o servidor NFS. Tem como objectivo identificar um dado ficheiro, não sendo necessário recorrer a caminhos (*paths*) de ficheiros. No servidor NFS construído, o *file handle* é constituído pela seguinte informação: o identificador do inó do ficheiro, o identificador da

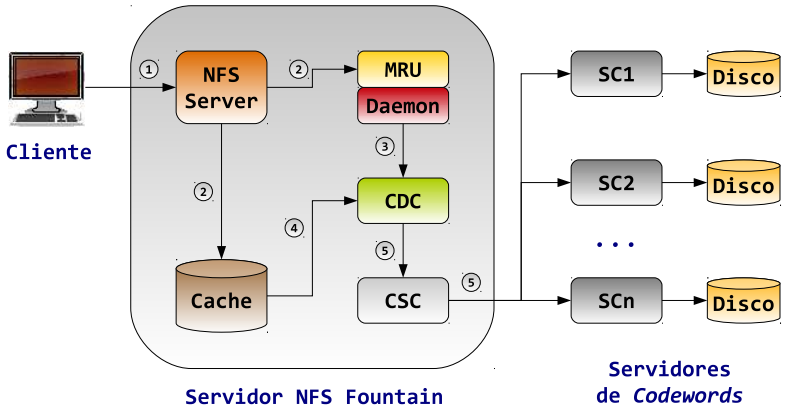


Figura 4.1: Interacções entre os componentes da arquitectura do sistema NFS Fountain para um pedido de armazenamento por parte do cliente. (1) O cliente envia a informação para o servidor NFS. (2) O servidor NFS armazena a informação em cache e actualiza a MRU. (3) O Deamon MRU detecta que a informação não é acedida à algum tempo e invoca o CDC. (4) O CDC codifica a informação que está em cache. (5) O CDC envia as *codewords* para os SCs através do CSC.

partição do disco, um identificador se é directório ou não, e um identificador que diferencia o *file handle* para os *hard links* de um ficheiro.

O inó de um ficheiro é uma estrutura dos sistemas Unix que armazena a informação relativa a um ficheiro presente num sistema de ficheiros. Cada ficheiro (ou directoria, visto que a nível de armazenamento não existe diferença) possui um único inó associado. O inó fornece informações como o tamanho do ficheiro, o tipo de ficheiro, as permissões, os tempos de acesso e o número de referências para o inó. Assim, cada inó possuí um identificador único no sistema de ficheiros capaz de referenciar um dado ficheiro.

O identificador da partição de disco é obviamente um identificador único que referencia a partição de disco onde se encontra o ficheiro.

O identificador de directório explicita se o *File Handle* é relativo a um directório ou não.

Em Unix existe a noção de *hard link*. Um *hard link* é uma referência para um ficheiro. No entanto, esta referência é na verdade para o inó do ficheiro. Quando um ficheiro é apagado em Unix, o número de referências do inó referente a esse ficheiro é decrementado. Apenas quando o número de referências for nulo, o ficheiro é realmente eliminado. Como o identificador inó referenciado pelos *hard links* é igual, é necessário haver um identificador que os distinga. O identificador de *hard link* é então um identificador que tem como objectivo diferenciar os possíveis *hard links* para um ficheiro.

### 4.3.2 FileInfoDB

A FileInfoDB é uma base de dados onde é persistida informação necessária ao correcto funcionamento do sistema NFS Fountain. É composta por vários atributos relativos aos ficheiros que são armazenados no sistema. Todos estes atributos são referenciados através do *File Handle* para cada ficheiro. A tabela 4.1 mostra os atributos persistidos na FileInfoDB.

O servidor NFS foi implementado em *user level*, pelo que não existe nenhum método simples de aceder ao inó de um ficheiro através do seu identificador único. No entanto, é

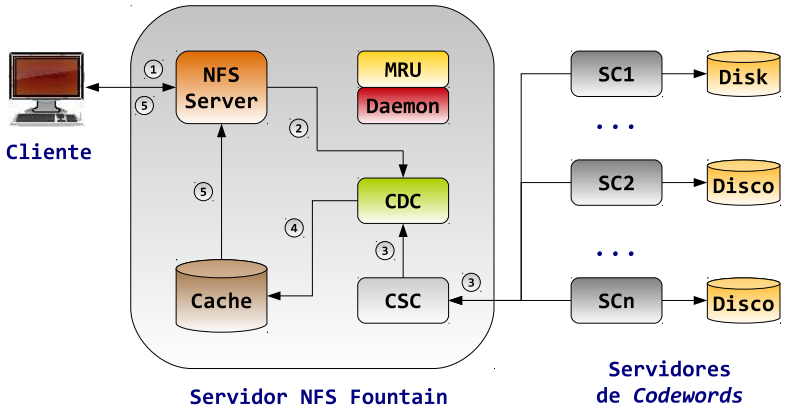


Figura 4.2: Interacções entre os componentes da arquitectura do sistema NFS Fountain para um pedido de leitura por parte do cliente quando a informação não está em cache. (1) O cliente envia o pedido de leitura para o servidor NFS. (2) O servidor NFS invoca o CDC, visto que a informação não está em cache. (3) O CDC pede *codewords* aos Servidores de *Codewords* relativas à informação pretendida. (4) O CDC descodifica a informação e coloca-a em cache. (5) O servidor NFS lê a informação e envia-a ao cliente.

necessário um mecanismo de tradução de *File Handles* de modo a que o servidor consiga localizar um dado ficheiro no disco. Para resolver este problema, é realizado um mapeamento relativo ao *File Handle* e o *path* onde é armazenado em cache.

Por outro lado, existe a necessidade de armazenar um identificador global único associado a um ficheiro. Este identificador, denominado *FileId*, é usado para referenciar um ficheiro fora do servidor NFS. É usado, por exemplo, para identificar as *codewords* que pertencem a um ficheiro. Como as *codewords* são armazenadas em Servidores de *Codewords* existe a necessidade deste identificador global. O *FileId* de um ficheiro é calculado através de uma função de *Hash*.

Quando um ficheiro é codificado, é retirado da cache. No entanto, o cliente pode navegar pelo sistema de ficheiros e ler os atributos dos ficheiros. Quando o ficheiro não está em cache, é necessário armazenar os seus atributos, como o seu tamanho. Na FileInfoDB, é armazenado o tamanho dos ficheiros quando são codificados. É também armazenado um identificador que explicita se um determinado ficheiro está ou não presente em cache. Os restantes atributos dos ficheiros são armazenados de outra forma e são abordados mais à frente na secção 4.3.3.

| *File Handle* | *Path* | FileId | Size | OnCache |
|---|---|---|---|---|
| fh1 | /file1 | 1 | - | true |
| fh2 | /file2 | 2 | 1000 | false |

Tabela 4.1: Atributos persistidos na FileInfoDB.

### 4.3.3 Cache

A cache não é mais do que o sistema de ficheiros exportado pelo servidor NFS. O cliente pode efectuar sobre este sistema de ficheiros operações de gestão de ficheiros como se tratasse de um sistema de ficheiros local. Quando o cliente monta um sistema de ficheiros remoto, o

servidor NFS responde com o *File Handle* relativo à raiz do sistema de ficheiros exportado. A raiz do sistema de ficheiros exportado pode ser qualquer directório no sistema de ficheiros local do servidor e é representado nos sistemas Unix com o caractere "/". O cliente após montar o sistema de ficheiros, pode aceder (se tiver permissões) à árvore do sistema de ficheiros a partir desse directório raiz. A figura 4.3, mostra uma ilustração da percepção do sistema de ficheiros remoto pelo cliente. Os ficheiros podem ser armazenados no sistema de ficheiros exportado na raiz ou em directórios filhos da raiz.

No sistema NFS Fountain, os ficheiros que são acedidos com frequência estão presentes na cache do servidor NFS. Por sua vez, ficheiros que não são acedidos há algum tempo são codificados e a sua informação é eliminada da cache. Este comportamento é necessário porque a constante codificação e descodificação de informação não é eficiente. No entanto, quando um ficheiro é codificado, a estrutura da árvore do sistema de ficheiros da cache não é alterada. Isto significa que o ficheiro que foi codificado continua em cache, no entanto, não possui nenhuma informação, pelo que o seu tamanho é nulo. Como é necessário armazenar a meta-informação do ficheiro codificado, esta abordagem é vantajosa no ponto de vista de não ser necessário recorrer a base de dados para armazenar esta informação. Esta árvore do sistema de ficheiros da cache também pode ser codificada, com o objectivo de poder recuperar a árvore do sistema de ficheiros da cache em caso de falhas.

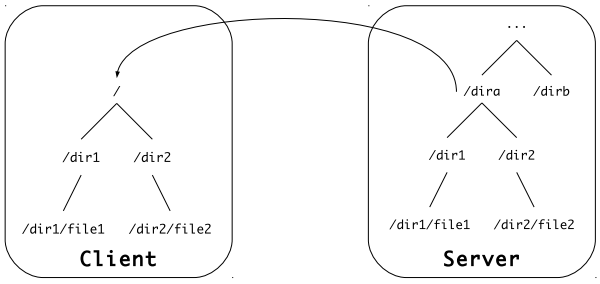


Figura 4.3: Percepção do sistema de ficheiros exportado no servidor pelo cliente NFS.

### 4.3.4 MRU

A MRU é uma base de dados do tipo *Most Recently Used*, onde é persistida informação relativa ao tempo de acesso dos ficheiros em cache. É composta basicamente pelo identificador e data de último acesso de um ficheiro. Os datas são referenciadas pelo *FileId* do ficheiro. A tabela 4.2 mostra os atributos persistidos na FileInfoDB.

Sempre que um ficheiro em cache é acedido no modo de escrita, o servidor NFS actualiza a informação relativa à data de acesso na MRU. Esta base de dados, fornece uma interface que permite o conhecimento dos ficheiros em cache que não são acedidos à um determinado tempo. Ao apenas ser actualizada aquando o acesso no modo de escrita, assegura-se que a informação em cache é apenas codificada quando existem alterações na informação.

### 4.3.5 Integração com o CDC

O servidor NFS apenas comunica com o CDC para o processo de descodificação de um ficheiro. Quando o cliente tenta aceder a um ficheiro que não está em cache, o servidor NFS invoca o CDC de modo a descodificar o ficheiro pretendido. Esta invocação é síncrona, pelo

| FileId | *File Handle* | Date |
|---|---|---|
| 1 | fh1 | 12/12/2009 |
| 2 | fh2 | 9/12/2009 |

Tabela 4.2: Informação persistida na MRU.

que o servidor NFS só responde ao cliente quando o CDC terminar o processo de descodificação (com ou sem sucesso). Para o sistema funcionar, é necessário que o cliente monte o sistema de ficheiros com o parâmetro *hard*. Este parâmetro faz com que os pedidos do cliente não respondidos pelo servidor sejam sempre replicados enquanto não houver resposta.

## 4.4 *Deamon* MRU

O *Deamon* MRU é responsável pela gestão dos ficheiros em cache. Periodicamente, o *Deamon* MRU procura na MRU por ficheiros que não foram acedidos à um tempo predefinido. Sempre que existir na MRU um ficheiro que não é acedido à algum tempo, o *Deamon MRU* invoca o CDC de modo a codificar o ficheiro em questão, retirando-o da cache. O *Deamon* MRU usa a interface fornecida pela MRU para determinar os ficheiros que não foram acedidos à um certo tempo.

O servidor NFS e o *Deamon MRU* correm em paralelo. A sincronização entre estes dois componentes no acesso à MRU e cache é feita através do principio de exclusão mútua.

O tempo de ciclo de verificação na MRU é um parâmetro configurável que permite definir o tempo que o *Deamon* MRU espera até voltar a verificar a MRU. Outro parâmetro configurável é o tempo limite de acesso. Este tempo considera quando um ficheiro está armazenado sem actividade na cache. Caso ultrapasse esse tempo, o ficheiro é codificado e removido da cache.

Existe também a possibilidade de fazer *flush* manual aos ficheiros em cache. Este processo força a verificação na MRU por ficheiros não acedidos recentemente.

## 4.5 CDC

O CDC é o componente responsável pela codificação e descodificação de ficheiros. Este componente é composto por duas entidades: o codificador e o descodificador.

### 4.5.1 Codificador

O codificador é responsável pela codificação dos ficheiros. No processo de codificação o ficheiro é dividido em $k$ símbolos com o mesmo tamanho $t$. O valor $t$ é por omissão $10kb$, no entanto, pode ser configurado pelo utilizador. Caso o tamanho da informação não seja múltiplo de $t$, é efectuado um *padding* no último símbolo com informação nula, de modo a perfazer o tamanho $t$. A figura 4.4 visualiza uma ilustração da informação e dos seus símbolos.

O código associado ao codificador é o código LT. Como foi referido na secção 2.2, o código LT depende de dois parâmetros $c$ e *delta* que influenciam o número de *codewords* de grau 1 geradas. Estes valores têm por omissão os valores $c = 0.03$ e $\delta = 0.5$, no entanto, podem ser alterados pelo utilizador. A escolha destes valores prende-se com o facto de resultar numa boa relação entre *codewords* com grau 1 e *codewords* necessárias à descodificação, como se mostra em [21].

A criação de *codewords* é feita usando o algoritmo descrito na secção 2.2.1. Inicialmente, é determinado um grau aleatório que segue a distribuição do código LT. O grau $d$ da *codeword* representa o número de símbolos que são combinados linearmente para formar a *codeword*, onde $1 \leq d \geq k$. De seguida, são escolhidos com aleatoriedade $d$ símbolos que são combinados linearmente usando a operação $\oplus$. Este procedimento é repetido para a criação de todas as *codewords*. O número de *codewords* geradas é por omissão $2k$, no entanto, este valor pode ser definido pelo utilizador ou determinado em função do número Servidores de *Codewords* existentes e da redundância pretendida.

Na secção 2.2.1, foi também referido que no processo de descodificação é necessário haver conhecimento da matriz de codificação de *codewords*. Por outras palavras, é necessário que a cada *codeword* seja associado o seu grau e uma lista com identificadores dos símbolos que a compõem. Esta informação é armazenada para cada *codeword* gerada numa estrutura, denominada *Codeword*. A estrutura *Codeword* é composta pelo grau, a semente e pela informação da *codeword*. A figura 4.5 mostra a estrutura *Codeword*. O grau da *codeword* é então o número de símbolos combinados na *codeword*. Como o armazenamento de uma lista de identificadores de símbolos não é prático, é usado uma semente de números aleatórios. A semente é um valor que alimenta um gerador de números aleatórios e que permite gerar a lista de identificadores dos símbolos que compõem a *codeword*. Finalmente, a informação da *codeword* refere obviamente a *codeword*.

### 4.5.2 Descodificador

O descodificador é responsável pela descodificação dos ficheiros através de *codewords*. O primeiro passo do algoritmo de descodificação passa pela descodificação de um símbolo através de uma *codeword* com grau 1. No entanto, podem chegar inicialmente *codewords* ao descodificador com grau diferente de 1. Estas *codewords* são armazenadas até serem usadas no processo de descodificação. No descodificador construído as *codewords* são armazenadas em memória.

Na secção 2.3, foram apresentadas algumas estruturas de dados capazes de armazenar matrizes de natureza esparsa, com o objectivo de estudar a melhor forma de estruturar as *codewords* de forma eficiente. Com o intenção de construir um descodificador eficiente, resolveu-se construir uma nova estrutura de dados que se adaptasse à descodificação de códigos LT. Da visualização do processo de descodificação na secção 2.2.2, dá para perceber as características que a estrutura de dados deve implementar. Nomeadamente, é de extrema importância que a estrutura disponibilize um acesso eficiente a todas as *codewords* que dependem de um dado símbolo. Esta característica é importante devido à necessidade no processo de descodificação de encontrar todas as *codewords* que dependem de um símbolo descodificado, de modo a remover a sua dependência nestas. Esta forma de armazenamento assemelha-se à estrutura CCS no facto de referenciar a matriz de codificação através das colunas, neste caso, dos símbolos que dependem das *codewords*.

O número de símbolos $k$ a descodificar é conhecido. Pode-se definir uma estrutura, *SimbolStruct*, com $k$ elementos, onde o elemento $SimbolStruct_i$ identifica o símbolo $S_i$. Como é necessário armazenar todas as *codewords* que dependem de um símbolo, cada elemento $SimbolStruct_i$ referencia uma outra estrutura denominada *CodewordDepStruct*. É também armazenado para cado elemento $SimbolStruct_i$, o número de elementos que constituiem a estrutura *CodewordDepStruct* referente.

A estrutura *CodewordDepStruct* é responsável por armazenar referências para todas as

*codewords* que dependem do símbolo $S_i$. No entanto, o número de *codewords* que são dependentes de um símbolo não é conhecido em princípio. Para resolver este problema, a estrutura *CodewordDepStruct* é alocada dinamicamente. Esta estrutura representa uma lista dinâmica de *codewords* dependentes de um dado símbolo. Cada elemento desta lista é uma referência para a estrutura *Codeword*, que representa a *codeword* gerada no codificador.

A Figura 4.6 mostra as estruturas mencionadas anteriormente.

### 4.5.3 Integração com o Cliente do Servidor de *Codewords*

As *codewords* geradas pelo codificador são armazenadas remotamente nos Servidores de *Codewords*. Para o codificador/descodificador enviar/receber *codewords*, usa uma interface disponibilizada pelo Cliente do Servidor de *Codewords*. O modo de comunicação e distribuição de *codewords* é descrito mais à frente na secção 4.6.

## 4.6 Cliente e Servidor de *Codewords*

O Servidor de *Codewords* (SC) é o componente responsável pelo armazenamento de *codewords* geradas no CDC. O CDC pode comunicar com múltiplos SCs através de uma interface disponibilizada pelo cliente CSC.

O CSC fornece dois serviços ao CDC: armazenar e requisitar *codewords* aos SCs. O CSC interage com uma base de dados, *ServerDB*, que persiste os endereços virtuais dos servidores SCs. Os atributos que são persistidos nesta base de dados, não são mais do que o endereço IP e a respectiva porta de escuta do servidor SC. Existe a possibilidade de adicionar e remover dinamicamente os endereços dos servidores da *ServerDB*. A tabela 4.3 mostra os atributos persistidos na textitServerDB.

Por omissão, o CSC tenta distribuir equitativamente as *codewords* por todos os servidores SC. Para isso, liga-se previamente a todos os servidores SCs presentes na *ServerDB*. No entanto, pode ocorrer a situação onde os servidores não estão disponíveis, pelo que a ligação com estes SCs não é possível. O pedido de envio/recepção de *codewords* é feito para cada SC, um de cada vez, presente na *ServerDB*.

No processo de armazenamento/requisição de *codewords*, o CSC envia para os SCs um pedido constituído pelo tipo de serviço pretendido (armazenar ou requisitar *codewords*, o *FileId* do ficheiro em questão e o número máximo de *codewords* $N$ que está disposto a enviar/receber no momento.

No processo de armazenamento de *codewords*, os SCs ao receberem o pedido ficam à escuta de $N$ *codewords* relativas ao *FileId* especificado no pedido. As *codewords* são processadas e armazenadas numa base de dados, denominada *CodewordDB*. Na *CodewordDB*, as *codewords* são armazenadas e referenciadas através do seu *FileId*. A tabela 4.4 mostra a informação persistida na base de dados *CodewordDB*. No pedido do processo de armazenamento é também enviado uma *flag* de eliminação de *codewords*. Caso esteja activa, as *codewords* relativas ao *FileId* são removidas previamente antes do armazenamento das novas *codewords*. Este comportamento é desejado no caso de alteração da informação, pelo que as *codewords* armazenadas anteriormente já não são válidas.

No caso de requisição de *codewords*, os SCs enviam $N$ *codewords* relativas ao *FileId* especificado no pedido. O CSC fica então à escuta de $N$ *Codewords* enviadas pelo SC. No pedido do processo de requisição é também enviado um *offset* de *codewords*, que permite ao SC saber que *codewords* já enviou previamente ao CSC numa sessão anterior. O processo de requisição

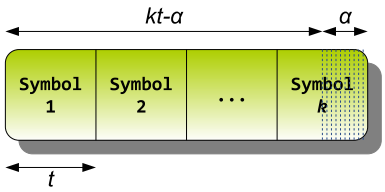


Figura 4.4: Símbolos de um ficheiro. Onde $t$ é o tamanho de símbolo; $k$ o número de símbolos; $kt - \alpha$ o tamanho do ficheiro; e $\alpha$ o *padding* adicionado.

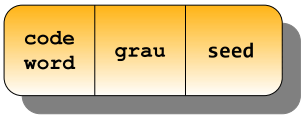


Figura 4.5: Estrutura *Codeword*.

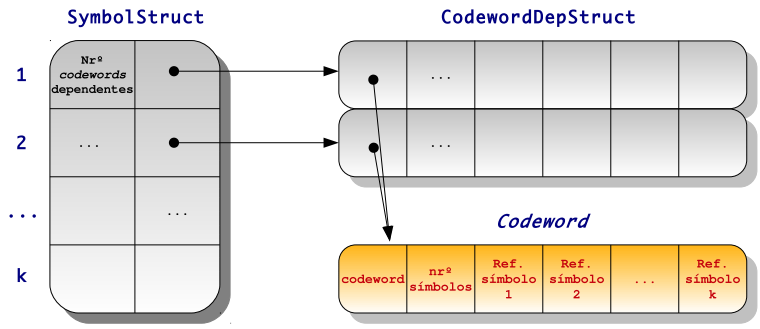


Figura 4.6: Estrutura de dados do descodificador.

| Endereço IP | Porta |
|---|---|
| 192.10.10.1 | 10000 |
| 192.10.10.2 | 10001 |

Tabela 4.3: Informação persistida na *ServerDB*.

| *FileId* | Grau | Semente | *Codeword* | Tamanho da *Codeword* (bytes) |
|---|---|---|---|---|
| 1 | 2 | 12345 | cw1 | $10k$ |
| 2 | 1 | 54321 | cw2 | $10k$ |

Tabela 4.4: Informação persistida na *CodewordDB*.

é repetido até o *CSC* desejar ou até acabarem as *codewords* no SC. Quando um SC não pode enviar mais *codewords* relativas ao *FileID* pretendido, é enviado para o CSC uma *codeword* com informação nula, indicando esse facto.

# Capítulo 5

# Realização

## 5.1 Sumário

Neste capítulo são apresentados alguns detalhes da implementação dos componentes que constituem a arquitectura do sistema NFS Fountain.

## 5.2 Servidor NFS

O servidor NFS foi desenvolvido em modo *user space* que corre em *loopback*. Isto significa que a comunicação entre o processo do servidor e o *kernel* do sistema de operação, realiza-se através de chamadas de sistema (*syscalls*). Por consequente, a comunicação entre o servidor e o sistema de ficheiros do disco de armazenamento local, também se realiza através de *syscalls*. As *syscalls* são fornecidas pela camada VFS (*Virtual File System*) presente no *kernel* do sistema de operação Linux. O VFS é uma camada de abstracção que proporciona uma API genérica de gestão de sistema de ficheiros. Através desta API, consegue-se armazenar informação em qualquer tipo de sistema de ficheiros, desde que seja suportado pelo *kernel* do sistema Linux. O desenvolvimento do servidor NFS passou pela implementação dos procedimentos fornecidos pelo protocolo MOUNT e NFS (secção 2.4) usando essas *syscalls*.

Como já foi referido na secção 4.3.5, a interacção entre o servidor e o descodificador de *codewords* é totalmente síncrona, pelo que é necessário que o cliente force a replicação de pedidos caso o processo de descodificação demore demasiado tempo. Isto é conseguido através da opção de montagem *hard* no cliente. Esta opção permite montar o sistema de ficheiros remoto no cliente e assegurar que o servidor responda sempre aos pedidos do cliente (caso esteja a processar pedidos).

### 5.2.1 *FileInfoDB*

Como foi referido na secção 4.3.2, a *FileInfoDB* é uma base de dados que persiste informação sobre alguns atributos de um ficheiro. Esta base de dados deve o mais eficiente possível, já que é acedida sempre que o servidor NFS processa um pedido do cliente. Com este objectivo, a *FileInfoDB* foi implementada usando a biblioteca GDBM [3].

A biblioteca GDMB é uma implementação GNU da biblioteca DBM do Unix. É constituído por um conjunto de rotinas de base de dados que usa *hashing* para referenciar a informação armazenada. O GDBM armazena qualquer tipo de informação usando uma chave

única. Esta cache fica associada à informação e é necessária para a recuperação da informação associada. O GDMB usa técnicas de *hashing* para permitir o acesso rápido à informação através da chave associada. Podem ser consultadas mais informações sobre esta biblioteca no anexo B.

Na *FileInfoDB* a informação é armazenada (e recuperada) através do *File Handle* dos ficheiros. A informação armazenada refere-se a alguns atributos dos ficheiros no sistema NFS Fountain.

### 5.2.2 MRU

Como foi referido na secção 4.3.4, a *MRU* é uma base de dados que persiste informação sobre a data de acesso no modo de escrita a ficheiros em cache. Esta base de dados deve fornecer uma interface onde é possível ter conhecimento de todos os ficheiros que já não são acedidos à um tempo definido, pelo que o uso da biblioteca GDBM não resulta. A base de dados *MRU* é construída e acedida através da API da biblioteca SQLite [5].

O *SQLite* é uma biblioteca simples que proporciona um aceso a base de dados do tipo SQL transaccional que não requer o uso de um servidor SQL. A informação da base de dados é armazenada em ficheiros. Através destas e de outras características, o SQLite é ideal para aplicações que necessitem de uma base de dados SQL eficiente. No anexo B, podem ser consultadas mais informações sobre esta biblioteca.

A informação na MRU é referenciada através do *FileId* de um ficheiro. O atributo *FileId* é a chave-primária da base de dados MRU.

## 5.3 *Deamon* MRU

O *Deamon* MRU é um processo que corre em *background* e em paralelo ao servidor NFS. É responsável pela gestão dos ficheiros em cache. Sempre que um ficheiro não for acedido à um tempo predefinido, o *Deamon* MRU executa os procedimentos necessários à codificação e remoção do ficheiro em cache.

Como o *Deamon* MRU corre em paralelo com o servidor NFS é necessário um mecanismo de sincronização para o acesso à base de dados MRU e à cache. Este problema foi resolvido recorrendo ao uso de um *mutex* fornecido pela API do *POSIX Threads*.

Um *mutex* é um mecanismo de sincronização que impõe o princípio de exclusão mútua num recurso. O *mutex* é um dos principais meios para assegurar a sincronização entre várias *threads* e a protecção de informação partilhada quando ocorrem múltiplos acessos de escrita num recurso. O *mutex* funciona como uma fechadura que nega o acesso a um recurso partilhado. O conceito básico do *mutex* na implementação do *Pthreads* é a característica de apenas uma *thread* poder fechar a fechadura de cada vez. No processo de várias *threads* tentarem fechar a fechadura e adquirir o acesso ao recurso, apenas uma terá acesso imediato. As restantes *threads* ficam bloqueadas à espera pela sua vez. Quando a *thread* abrir a fechadura, é dado a oportunidade às *threds* bloqueadas de adquirirem o recurso.

## 5.4 CDC

O processo de codificação é relativamente simples pelo que não foi necessário fazer algo de significativo para o seu desenvolvimento. Por outro lado, a implementação do descodificador

é algo bastante complexo. Nesta secção, é abordado o algoritmo de descodificação através da arquitectura do descodificador apresentada na secção 4.5.

### 5.4.1 Descodificação

Como exemplo de descodificação, pode-se recorrer às *codewords* criadas anteriormente na tabela 2.1. Assume-se que as *codewords* são processadas por ordem de chegada, ou seja, pela ordem que são criadas. É também assumido que todas as *codewords* chegam ao descodificador e sem erros.

No início da descodificação a estrutura *SymbolStruct* é inicializada com o tamanho da mensagem original, ou seja, $k = 4$. Não existe mais nenhuma estrutura inicializada. Seguidamente, chega a *codeword* $cw1$. Esta tem grau 3 e depende dos símbolos $s1$, $s2$ e $s4$. Nos elementos respectivos na estrutura *SymbolStruct* é então adicionado um elemento às estruturas *CodewordDepStruct* que referenciam a *codeword* $cw1$. Como não é possível despoletar a descodificação, a *codeword* $cw1$ fica armazenada em memória. A figura 5.1 visualiza o processo de descodificação.

Passado algum tempo a *codeword* $cw2$ chega ao descodificador. Esta tem grau 2 e depende dos símbolos $s1$ e $s3$. Novamente, adiciona-se uma referência para a *codeword* $cw2$ nos símbolos $s1$ e $s3$. A *codeword* $cw1$ fica também armazenada em memória. A figura 5.2 visualiza o processo de descodificação.

De seguida chega a *codeword* $cw3$. Esta tem grau 1 e depende do símbolo $s2$, pelo que o símbolo $s2$ é descodificado. A *codeword* $cw3$ é eliminada. A figura 5.3 visualiza o processo de descodificação.

Com a descodificação do símbolo $s2$, é removida a dependência deste símbolo na *codeword* $cw1$. A *codeword* $cw1$ passa a ser de grau 2. Não é possível fazer mais nada. A figura 5.4 visualiza o processo de descodificação.

Ao chegar a *codeword* $cw4$ (figura 5.5), verifica-se que esta depende dos símbolos $s2$ e $s4$. Como o símbolo $s2$ já foi descodificado é removida a dependência na *codeword* $cw4$. Com a remoção da dependência a *codeword* $cw4$ passou a depender apenas do símbolo $s4$. O símbolo 4 é então descodificado. A figura 5.6 visualiza o processo de descodificação.

É removida a dependência do símbolo $s4$ na *codeword* $cw1$. Novamente, com a remoção da dependência na *codeword* $cw1$, esta passou só a depender do símbolo $s1$. O símbolo $s1$ é descodificado. A figura 5.7 visualiza o processo de descodificação.

O processo é repetido para o símbolo 1. É removida a dependência do símbolo $s1$ na *codeword* $cw2$. A *codeword* $cw1$ que ainda estava em memória é eliminada porque não contém informação relevante. A figura 5.8 visualiza o processo de descodificação.

A *codeword* $cw2$ passou só a depender do símbolo $s3$. O símbolo $s3$ é descodificado (figura 5.9). De seguida, é removida a dependência do símbolo $s3$. Como a *codeword* $cw2$ não tem informação relevante é eliminada (figura 5.10). Finalmente o processo de descodificação termina visto que todos os símbolos foram descodificados (figura 5.11).

## 5.5 Servidor de *Codewords*

O Servidor de *Codewords* (SC), foi desenvolvido através de duas abordagens. Inicialmente foi desenvolvido através de uma simples implementação de um servidor usando *sockets* sobre TCP/UDP e IP. Mais tarde, foi desenvolvido usando uma implementação que usa o protocolo

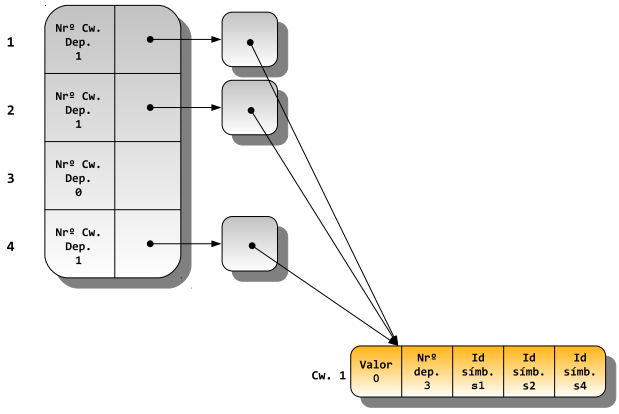


Figura 5.1: Chega a *codeword* 1 ao descodificador.

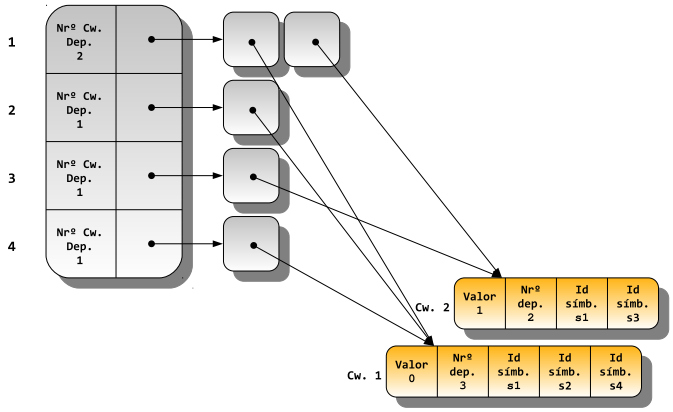


Figura 5.2: Chega a *codeword* 2 ao descodificador.

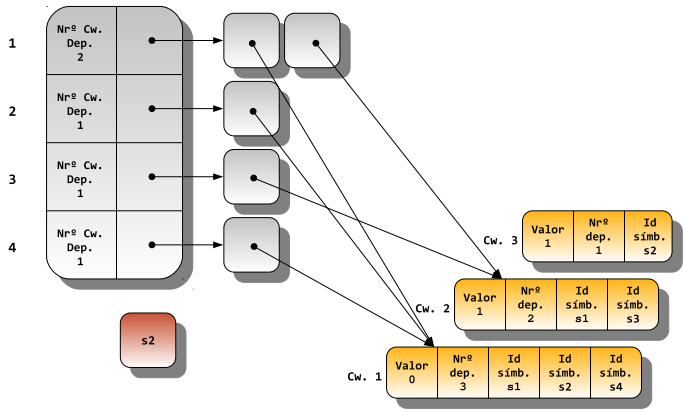


Figura 5.3: Chega a *codeword* 3 ao descodificador. O símbolo 2 é descodificado.

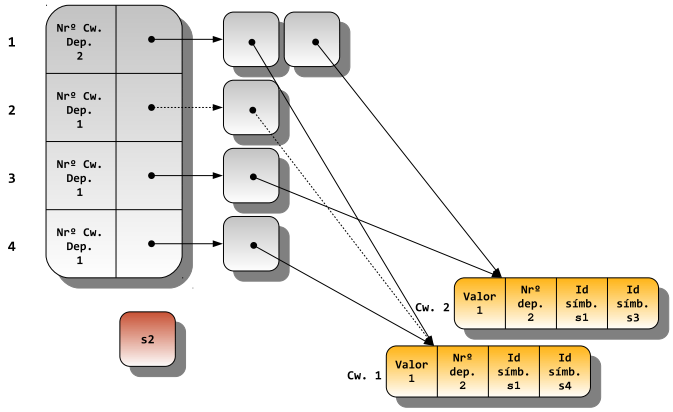


Figura 5.4: Remoção das dependências do símbolo 2 na *codeword* 1.

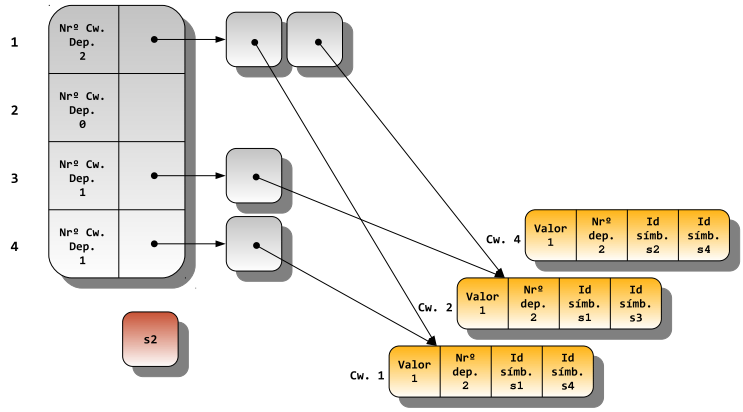


Figura 5.5: Chega a *codeword* 4 ao descodificador.

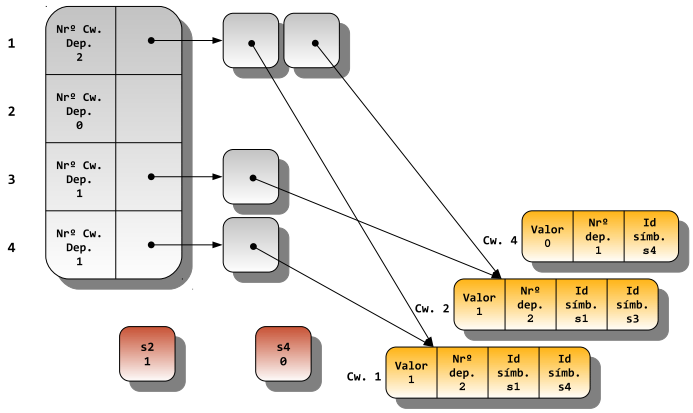


Figura 5.6: Remove-se a dependência do símbolo 2 na *codeword* 4. O símbolo 4 é descodificado.

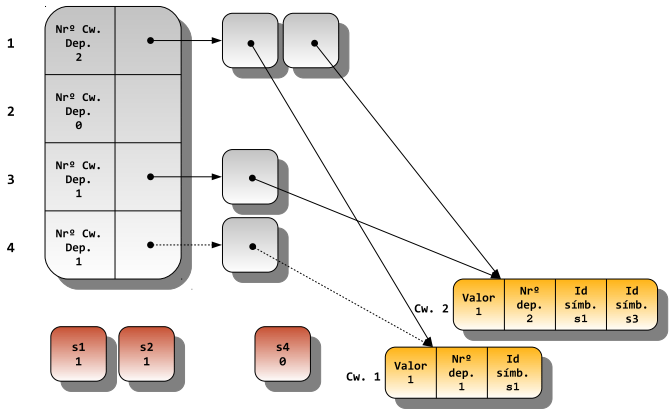


Figura 5.7: Remove-se a dependência do símbolo 4 na *codeword* 1. O símbolo 1 é descodificado.

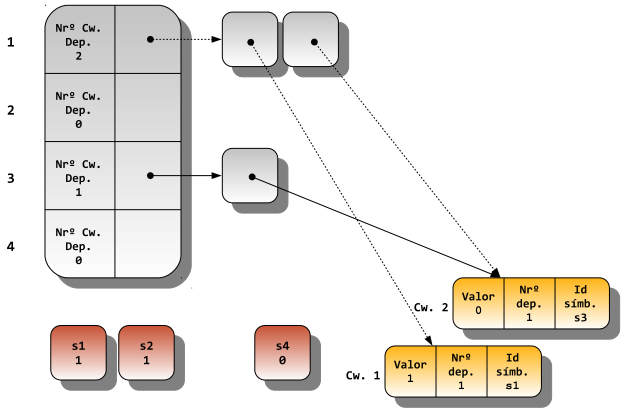


Figura 5.8: Remove-se a dependência do símbolo 1 na *codeword* 2. A *codeword* 1 é apagada.

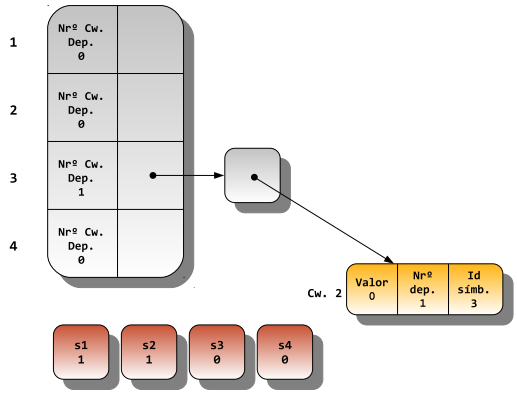


Figura 5.9: O símbolo 3 é descodificado.

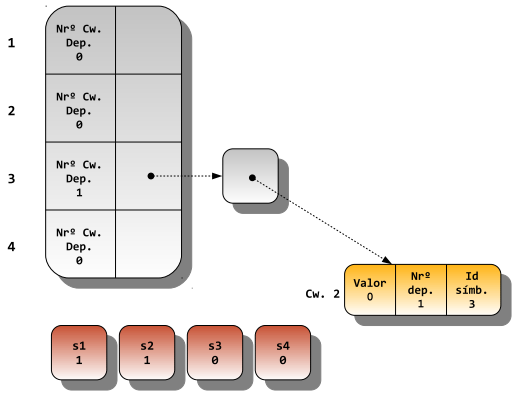


Figura 5.10: Remove-se as dependências do símbolo 3. A *codeword* 2 é apagada.

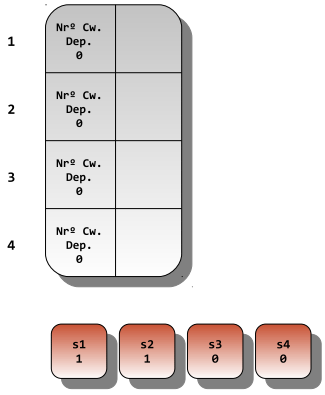


Figura 5.11: O processo de descodificação termina. Todos os símbolos foram descodificados.

RPC. No entanto, a implementação RPC sobre UDP limita o tamanho da *Codeword* a $8KB$, devido à limitação do UDP no protocolo RPC.

O servidor SC é responsável por processar dois tipos de pedidos enviados pelo servidor NFS: o armazenamento e a leitura de *codewords*. As *codewords* são armazenadas numa base de dados, *CodewordDB*. A *CodewordDB* é também construída e acedida pela biblioteca SQLite. Usando uma base de dados deste tipo pode-se por exemplo, saber facilmente que *codewords* pertencem a um ficheiro. É por esta razão que as *codewords* são armazenadas na base de dados e não directamente em ficheiros.

# Capítulo 6

# Avaliação

## 6.1 Sumário

Nesta secção são apresentados os resultados do desempenho e a tolerância a falhas do sistema NFS Fountain e seus componentes. Finalmente, o desempenho do sistema é avaliado tendo em conta as suas limitações. Os componentes do sistema NFS Fountain são avaliados quanto à eficiência de transmissão e processamento de informação.

## 6.2 Servidor NFS

Nesta secção são apresentados os resultados da eficiência do servidor NFS implementado para o sistema NFS Fountain, em termos da taxa média de transferência de ficheiros.

Os resultados apresentados na tabela 6.1, são relativos à transmissão de um ficheiro com 273,56 MB, usando uma LAN interligada com portas *ethernet* e cabos do tipo 100BASE-T que fornecem uma taxa de transferência até $100 Mbit/s$.

| Simulação | $T_x$ Máxima (MB/s) | $T_x$ Médio (MB/s) | $t$ Médio (s) |
|---|---|---|---|
| 1 | 9,61 | 4,86 | 56,29 |
| 2 | 11,3 | 5,71 | 47,91 |
| 3 | 9,44 | 5,4 | 50,66 |
| 4 | 9,77 | 4,48 | 61,06 |
| 5 | 10,03 | 5,71 | 47,91 |
| 6 | 9,26 | 5,56 | 49,2 |
| 7 | 9,54 | 5,71 | 47,91 |
| 8 | 10,36 | 5,03 | 54,39 |
| 9 | 9,47 | 5,12 | 53,43 |
| 10 | 9,94 | 5,48 | 49,92 |

Tabela 6.1: Taxa de Transmissão $T_x$ e Tempo de Transmissão $t$ Médios do Servidor NFS.

Com o objectivo de avaliar estes resultados, é apresentado na tabela 6.2, resultados obtidos para as mesmas condições anteriores através de uma implementação nativa de um servidor NFS.

| Simulação | $T_x$ Máxima (MB/s) | $T_x$ Médio (MB/s) | $t$ Médio (s) |
|---|---|---|---|
| 1 | 11,77 | 8,76 | 30,88 |
| 2 | 11,77 | 8,61 | 31,41 |
| 3 | 11,77 | 8,52 | 32,49 |
| 4 | 11,77 | 8,33 | 32,26 |
| 5 | 11,77 | 8,46 | 31,41 |
| 6 | 11,77 | 8,77 | 31,96 |
| 7 | 11.77 | 8,44 | 31,41 |
| 8 | 11,77 | 8,69 | 34,07 |
| 9 | 11,77 | 8,44 | 33,69 |
| 10 | 11,77 | 8,77 | 32,26 |

Tabela 6.2: Taxa de Transmissão $T_x$ e Tempo de Transmissão $t$ Médios do Servidor NFS.

Como seria de esperar, a implementação nativa é mais eficiênte em relação à implementação em *user level* do servidor NFS do sistema NFS Fountain.

## 6.3 CDC

Nesta secção, são apresentados os resultados obtidos em simulações de codificação de descodificação através o componente CDC do sistema NFS Fountain. Primeiro é feito uma comparação com os resultados obtidos no artigo [21], com o objectivo de verificar a correcta implementação do codificador de códigos LT.

De seguida, são mostrados os resultados de simulações para ficheiros de grande dimensão, que são um bom exemplo de caso de uso para o sistema NFS Fountain.

### 6.3.1 Comparação de Resultados

De modo a assegurar que a implementação do codificador e descodificador de códigos LT é eficiente, é interessante usá-la para tentar replicar os resultados obtidos em [21]. As características do código LT, apresentadas anteriormente na secção 2.2.3, também se devem verificar.

Os resultados que se apresentam de seguida, são referentes a um ficheiro com 40000 *bytes*. Todos os símbolos têm 4 *bytes* de tamanho, perfazendo no total, $k = 10000$ símbolos. Nas figuras 6.1, 6.2, 6.3 e 6.4 são apresentados histogramas que mostram o número de *codewords* necessárias (para além das 10000 *codewords* iniciais) à descodificação, para diversas simulações com $\delta = 0.5$ e vários valores de $c$. Pelos histogramas pode observar-se que ao ajustar os parâmetros da distribuição $c$ e $\delta$, consegue-se reconstruir a informação original apenas com um *overhead* de cerca de 5%. Estes resultados estão de acordo com os apresentados em [21]. Nota-se também que o valor de $c$ é inversamente proporcional ao número médio de codewords necessárias à descodificação.

Um aspecto importante a ter em conta é a evolução do número de símbolos descodificados ao longo da descodificação. Na figura 6.5, é apresentado o número de símbolos descodificados em função do número de *codewords* recebidas, para três codificações com parâmetros $c = 0.03$ e $delta = 0.5$. Observa-se que para estes valores de $c$ e $\delta$ a descodificação de símbolos só ocorre

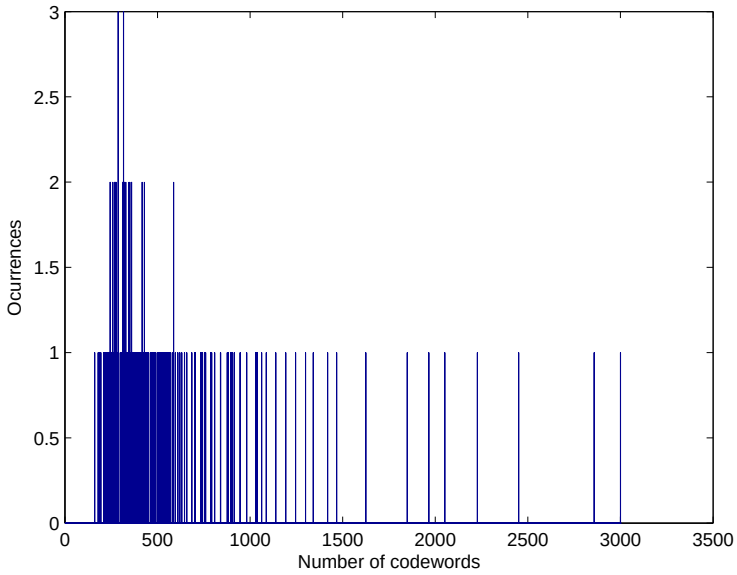


Figura 6.1: Histograma do número de codewords necessárias para recuperar um ficheiro com $k = 10000$ símbolos com parâmetros $c = 0.01$ e $\delta = 0.5$.

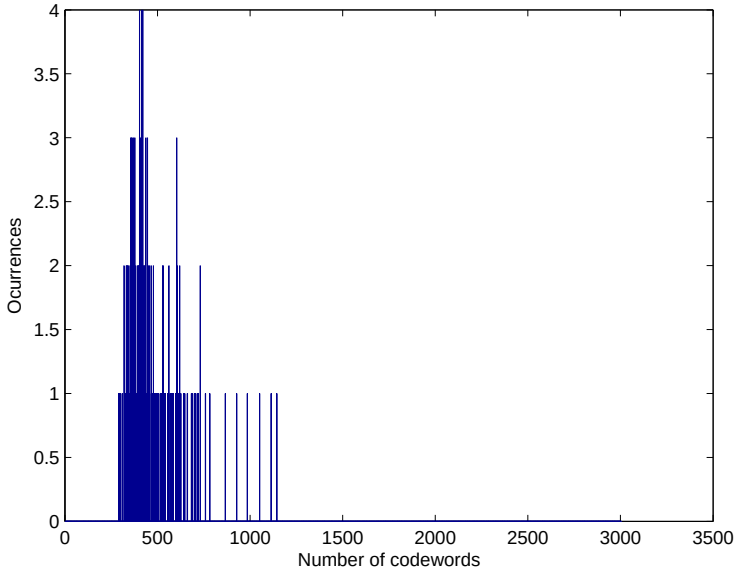


Figura 6.2: Histograma do número de codewords necessárias para recuperar um ficheiro com $k = 10000$ símbolos com parâmetros $c = 0.03$ e $\delta = 0.5$.

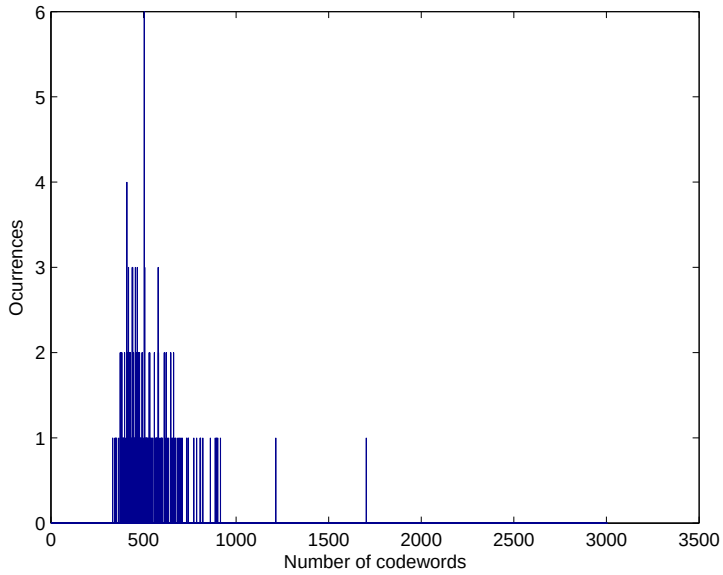


Figura 6.3: Histograma do número de codewords necessárias para recuperar um ficheiro com $k = 10000$ símbolos com parâmetros $c = 0.04$ e $\delta = 0.5$.

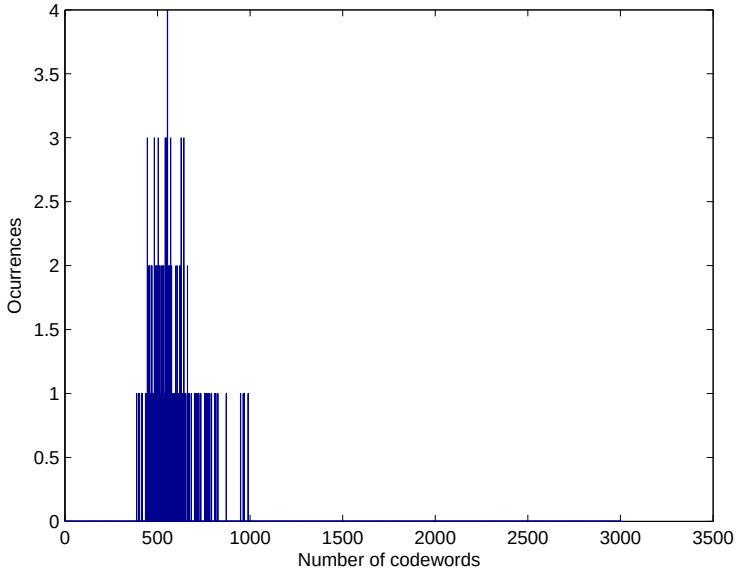


Figura 6.4: Histograma do número de codewords necessárias para recuperar um ficheiro com $k = 10000$ símbolos com parâmetros $c = 0.05$ e $\delta = 0.5$.

praticamente quando o descodificador recebe $k$ *codewords*, onde $k$ é o número de símbolos. Isto ocorre porque o número de *codewords* de grau 1 é baixo. Variando os parâmetros de $c$ ou *delta* é possível ajustar a curva de descodificação. No entanto, o número de *codewords* de grau 1 é directamente proporcional ao número de *codewords* necessárias à descodificação. Na figura 6.6 pode observar-se esse comportamento, quando se atribui ao parâmetro $c$ valores altos.

### 6.3.2 Resultados de Simulações

Nesta secção são apresentados os resultados de simulações de codificação e descodificação, ou seja, da codificação/descodificação de um ficheiro com dimensões adequadas a situações de caso de uso normais.

Para o codificador são apresentados valores médios para o tempo de codificação e para a percentagem de *codewords* geradas com grau 1. Quanto ao descodificador são apresentados valores médios para o tempo e para o pico máximo de memória atingido durante a descodificação. São também apresentados, valores médios para o número de *codewords* que foram necessárias à descodificação.

Os resultados que são apresentados nas tabelas seguintes são o resultado da codificação/descodificação de um ficheiro com $150MB$ de tamanho. No codificador foi definido um tamanho de símbolo $t_s = 10KB$, perfazendo no total $k = 15360$ símbolos, e foi configurado de modo a produzir no máximo $K = 2 * k$ *codewords*.

O sistema computacional usado para correr as simulações já tem alguma idade (5 anos), pelo que os resultados obtidos não serão os melhores. O sistema consiste num processador com um único *core* a correr à frequência de $1.7GHz$, com $2GB$ de memória RAM.

Foram feitas simulações para vários valores de $c$ e $\delta$, cujos resultados obtidos podem ser observados no anexo C.

A tabela 6.3 mostra valores médios para a percentagem de *codewords* geradas na codificação com grau 1, para vários valores de $c$ e $\delta$. Como já foi referido na secção 2.2, os parâmetros $c$ e $\delta$ influenciam o número de *codewords* com grau 1 geradas. Observa-se que uma ligeira variação no valor de $c$ tem um impacto significativo no número de *codewords* com grau 1, enquanto que uma ligeira variação no valor de $\delta$ tem um impacto muito menor.

A tabela 6.4 mostra valores médios para o tempo de codificação, para vários valores de $c$ e $\delta$. O tempo de codificação deveria diminuir consoante o aumento de *codewords* de grau 1, no entanto, olhando para a tabela verifica-se que se mantém praticamente inalterado. As razões para este facto devem prender-se com o método de gestão de memória e cache do sistema de operação.

A tabela 6.5 mostra valores médios para o número de *codewords* necessárias à descodificação, para vários valores de $c$ e $\delta$. Como se observa, e em confirmação aos resultados obtidos anteriormente em 6.3.1, consegue-se descodificar o ficheiro com um *overhead* de *codewords* baixo. Verifica-se também um resultado já esperado: quando o valor de $c$ aumenta, o número de *codewords* com grau 1 também aumenta, então o número de *codewords* necessárias à descodificação irá também aumentar. O mesmo acontece com $\delta$, a diminuição do seu valor implica um número maior de *codewords* necessárias à descodificação, embora num ritmo muito inferior.

A tabela 6.6 mostra valores médios para o tempo de descodificação, para vários valores de $c$ e $\delta$. O tempo de descodificação é praticamente semelhante nas simulações, visto que o número de *codewords* necessário à descodificação é também semelhante. No entanto, verifica-se que

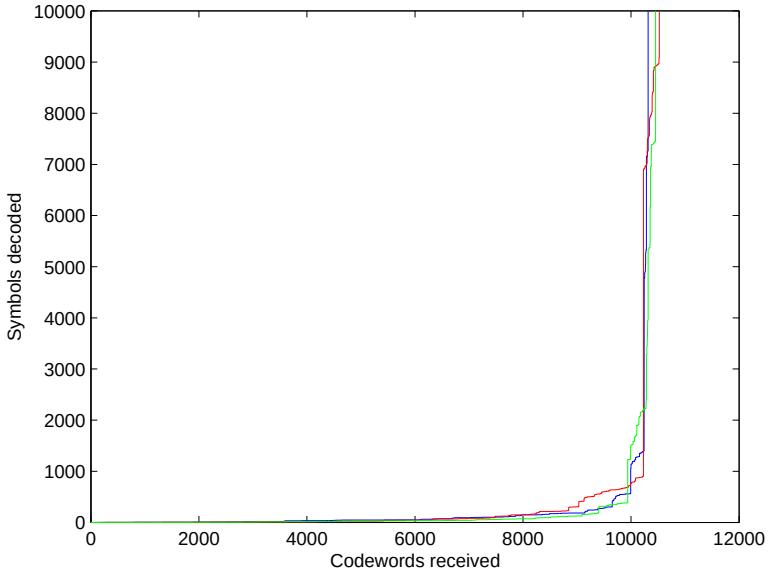


Figura 6.5: Símbolos descodificados em função do números de *codewords* recebidas pelo descodificador para três codificações com parâmetros $c = 0.03$ e $delta = 0.5$.

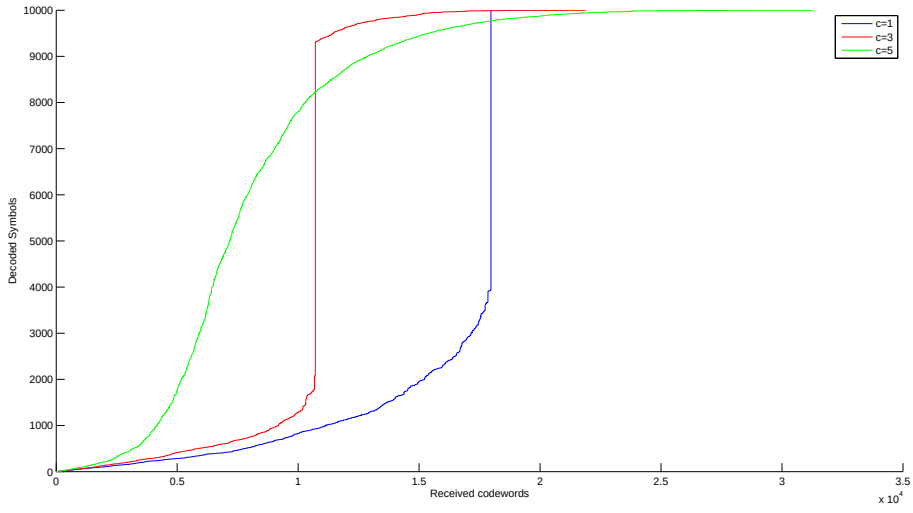


Figura 6.6: Símbolos descodificados em função do números de *codewords* recebidas pelo descodificador para três codificações com parâmetros $delta = 0.5$ e $c = 1..3$.

| c / $\delta$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 |
|---|---|---|---|---|---|
| 0.01 | 9.34 | 9.21 | 9.73 | 9.05 | 8.72 |
| 0.03 | 29.33 | 25.78 | 23.18 | 23.28 | 24.58 |
| 0.05 | 46.81 | 43.39 | 40.76 | 39.62 | 40.82 |
| 0.07 | 61.95 | 58.66 | 55.7 | 55.7 | 52.08 |
| 0.09 | 78.71 | 71.62 | 73.47 | 67.77 | 66.73 |

Tabela 6.3: Percentagem média de *codewords* com grau 1 geradas na codificação.

| c / $\delta$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 |
|---|---|---|---|---|---|
| 0.01 | 19.423 | 18.572 | 17.842 | 17.585 | 17.239 |
| 0.03 | 20.816 | 19.776 | 19.203 | 19.262 | 18.652 |
| 0.05 | 20.720 | 19.948 | 19.514 | 18.362 | 19.299 |
| 0.07 | 21.971 | 20.009 | 18.923 | 18.126 | 17.663 |
| 0.09 | 19.495 | 18.786 | 18.800 | 18.013 | 17.955 |

Tabela 6.4: Valores médios para o tempo de codificação.

| c / $\delta$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 |
|---|---|---|---|---|---|
| 0.01 | 16408.20 | 16211.50 | 16109.80 | 16081.60 | 16814.20 |
| 0.03 | 16295.10 | 16139.60 | 16189.90 | 16145.20 | 16083.00 |
| 0.05 | 16613.30 | 16456.00 | 16406.90 | 16327.80 | 16312.40 |
| 0.07 | 16899.70 | 16577.40 | 16566.20 | 16539.00 | 16449.90 |
| 0.09 | 17139.30 | 16857.20 | 16827.10 | 16638.40 | 16664.10 |

Tabela 6.5: Número médio de *codewords* necessárias à descodificação.

quanto maior for o número de *codewords* com grau 1, maior será o tempo de descodificação, pelo que o número de *codewords* necessárias à descodificação é maior.

A tabela 6.7 mostra valores médios para o pico de memória associado à descodificação, para vários valores de $c$ e $\delta$. O pico máximo de memória que ocorre durante a descodificação é também semelhante em todas as simulações. No entanto, este valor é bastante alto, na ordem dos 85% em relação ao tamanho do ficheiro. Isto torna-se um problema se o ficheiro tiver um tamanho na ordem dos $GB$. Este problema ocorre devido à natureza da descodificação dos códigos LT, onde a descodificação só ocorre praticamente quando o descodificador é alimentado com aproximadamente $k$ *codewords*. Na figura 6.7, pode observar-se esse facto, onde se visualiza a evolução da ocupação da memória durante uma descodificação. A taxa de descodificação de símbolos pode ser acelerada aumentando o número de *codewords* com grau 1, como é mostra a figura 6.6, no entanto o número de *codewords* necessárias à descodificação aumenta drasticamente. Para resolver este problema de forma eficiente é necessário implementar um mecanismo de cache de *codewords* em disco, de modo a apenas residirem em memória algumas *codewords*.

## 6.4 Servidor de *Codewords*

Nesta secção, são apresentados os resultados da eficiência do servidor de *Codewords* implementado para o sistema NFS Fountain, em termos da taxa média de transferência de *codewords*.

Os resultados apresentados na tabela 6.8, são relativos à transmissão de *codewords* com $10KB$ de tamanho, através de uma LAN que permite uma taxa de transferência até $100Mbit/s$.

Observou-se para ambas as implementações resultados identicos. A implementação do SC fornece uma boa taxa de transferência de *codewords* visto que o valor médio de transmissão é próximo da velocidade máxima da ligação.

### 6.4.1 Avaliação

Através dos resultados obtidos anteriormente, verifica-se que o desempenho do sistema NFS Fountain é satisfatório.

**Servidor NFS:** O servidor apresenta uma taxa de transmissão satisfatória. No entanto, existe uma perda de eficiência de cerca de $3MB/s$ em relação a uma implementação nativa em *kernel*. Como o objectivo do sistema NFS Fountain era a implementação de um protótipo de um sistema de ficheiros distribuído com uso de códigos Fountain, esta perda de eficiência não é de grande importância.

**CDC:** O CDC apresenta tempos de codificação e descodificação muito bons. No entanto, o consumo de memória no processo de descodificação é um problema. Existe um pico de memória associado à descodificação de cerca de 80% do tamanho do ficheiro. Para resolver este problema deve ser implementado um mecanismo de cache de *codewords*, de forma a reduzir o número de *codewords* em memória.

**Servidor de *Codewords*:** Pode-se dizer que a implementação do SC é boa, pelo que nos resultados obtidos obteve-se quase a taxa de transferência máxima imposta pela rede LAN.

| c / $\delta$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 |
|---|---|---|---|---|---|
| 0.01 | 7.120 | 6.610 | 6.120 | 5.465 | 5.766 |
| 0.03 | 7.086 | 6.757 | 6.549 | 6.390 | 5.904 |
| 0.05 | 6.928 | 6.616 | 6.897 | 6.496 | 6.173 |
| 0.07 | 7.125 | 6.770 | 6.471 | 6.331 | 6.191 |
| 0.09 | 7.375 | 6.801 | 6.553 | 6.260 | 6.578 |

Tabela 6.6: Valores médios para o tempo de descodificação.

| c / $\delta$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 |
|---|---|---|---|---|---|
| 0.01 | 142.653 | 145.999 | 139.207 | 141.523 | 152.284 |
| 0.03 | 128.800 | 131.755 | 132.954 | 131.520 | 131.500 |
| 0.05 | 128.740 | 128.091 | 127.519 | 129.221 | 129.089 |
| 0.07 | 125.309 | 127.213 | 123.559 | 124.837 | 132.132 |
| 0.09 | 128.080 | 126.866 | 125.677 | 125.430 | 125.086 |

Tabela 6.7: Valores médios para o pico de memória associado à descodificação.

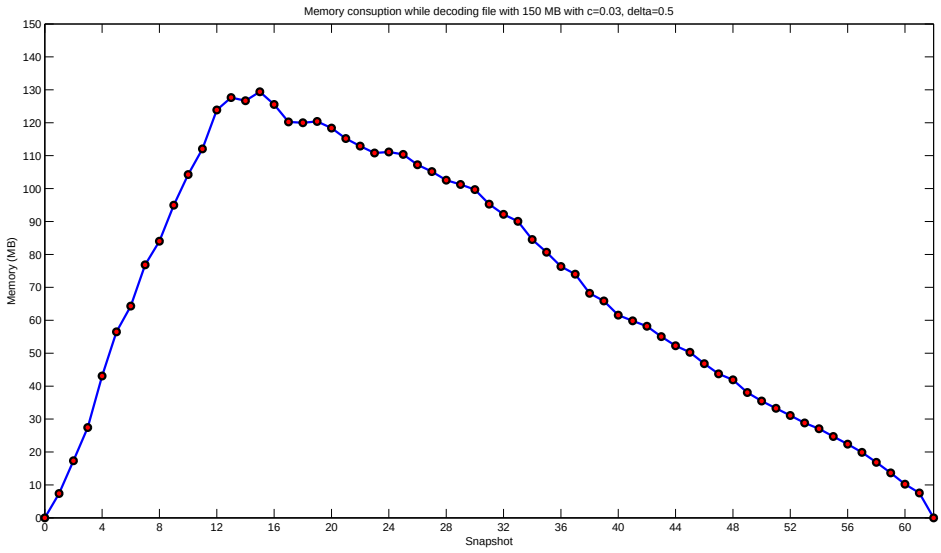


Figura 6.7: Evolução da memória durante a descodificação.

| Simulação | $T_x$ Máxima (MB/s) | $T_x$ Médio (MB/s) | $t$ Médio (s) |
|---|---|---|---|
| 1 | 11.77 | 8.46 | 117.54 |
| 2 | 11,77 | 8.36 | 118.91 |
| 3 | 11.77 | 8.36 | 117.67 |

Tabela 6.8: Taxa de Transmissão $T_x$ e Tempo de Transmissão $t$ Médios do Servidor de *Codewords*.

# Capítulo 7

# Conclusão e Trabalho Futuro

## 7.1 Sumário

Este capítulo completa a dissertação. É apresentado um breve resumo do trabalho realizado e a sua relevância nos sistemas de ficheiros. De seguida, é feita uma análise sobre o sistema desenvolvido, apresentando as suas principais vantagens e desvantagens, sendo apresentadas algumas ideias para trabalho futuro.

## 7.2 Conclusão

Com a forte necessidade a nível de armazenamento de informação no formato digital, existe cada vez mais, uma tendência para o uso de sistemas de ficheiros com capacidade de restauro de informação em caso de falhas.

Com o aparecimento dos códigos de correcção do tipo *Fountain*, surge um novo conceito que permite, com grande probabilidade de sucesso, a recuperação de informação através de um subconjunto qualquer de blocos codificados dessa informação, onde o somatório do tamanho dos blocos codificados, é ligeiramente superior ao tamanho da informação original.

Neste âmbito, é de grande interesse o desenvolvimento de novos sistemas para este novo conceito, o que no contexto desta dissertação, reflectiu-se no desenvolvimento de um sistema de ficheiros distribuído e na sua integração com um codificador e descodificador de um código Fountain. O objectivo é armazenar a informação em formato codificado com redundância de forma disitribuída em vários servidores. Assim, em caso de avaria de alguns servidores, é possível a recuperação da informação original com grande probabilidade de sucesso.

Na primeira parte deste trabalho, foi realizado um estudo sobre o código LT, que faz parte da família dos códigos Fountain. Através deste estudo, foi possível o desenvolvimento de um sistema codificador e descodificador eficiente para este tipo de códigos. Este sistema denominado CDC, permite a codificação e descodificação de qualquer ficheiro, independentemente do seu tamanho.

Na segunda parte deste trabalho, foi realizado um estudo sobre os sistemas de ficheiros distribuídos existentes actualmente. Foi decidido desenvolver o sistema de ficheiros distribuído através de uma implementação do protocolo NFS. A simplicidade e banalidade deste protocolo, contribuíram para a decisão do seu uso na implementação do protótipo do sistema. Foi desenvolvido um servidor NFS capaz de armazenar informação em cache. Na cache está apenas presente a informação acedida relativamente à pouco tempo. Existe associado ao servidor

NFS, um sistema MRU que regista o acesso à informação. Este sistema MRU é responsável por verificar se a informação em cache já não é acedida à um tempo previamente definido. Caso exista informação não acedida, é procedido à sua codificação usando o sistema CDC e enviada para servidores onde a informação é armazenada no formato codificado, denominados Servidores de *Codewords*. Caso essa informação seja requerida novamente, o sistema CDC do servidor NFS tenta descodificar a informação fazendo pedidos aos Servidores de *Codewords* pela informação pretendida, com o objectivo de a colocar novamente em cache.

Podemos então concluir que todos os objectivos propostos neste trabalho foram concluídos conseguindo-se desta forma demonstrar as capacidades de um protótipo simples de um sistema de ficheiros distribuído integrado com um codificador e descodificador de códigos LT.

## 7.3 Trabalho Futuro

### 7.3.1 Servidor NFS

O sistema NFS Fountain foi desenvolvido através da implementação de um servidor baseado no protocolo NFS. A sua implementação foi realizada em *user level*, pelo que a sua eficiência não será a melhor. Uma implementação a nível do *kernel* seria mais complicado, mas o servidor comunicava directamente com o *kernel*, pelo que a sua eficiência seria em muito favorecida. Seria também interessante o desenvolvimento de um cliente NFS. Deste modo, seria possível estender o protocolo NFS com mais procedimentos, podendo adicionar funcionalidade ao sistema.

Outro problema, é a interacção entre o servidor NFS e os restantes componentes do sistema NFS Fountain. Esta interacção é totalmente síncrona, o que leva a que o cliente replique os pedidos enviados ao servidor, de modo a que o servidor responda quando tiver a informação disponível, o que leva a um *overhead* na comunicação na rede. Para resolver este problema é necessário desenvolver uma forma de processar o pedido do cliente e comunicar-lhe que a informação ainda não está disponível. O cliente ao receber esta informação comunica periodicamente com o Servidor com intenção de saber se já tem a informação. Esta solução pode ser desenvolvida através de uma implementação do cliente NFS.

Por outro lado, a versão do protocolo usada (versão 2) é antiga e com pouca funcionalidade em termos de segurança e reabilidade dos procedimentos que fornece. Para resolver estes problemas seria necessário o uso de outra versão do protocolo NFS (por exemplo, versão 4), ou usar outro tipo de protocolos e/ou tecnologia (por exemplo, P2P). Existe por exemplo, o projecto Kosha [9], que tem objectivo extender o protocolo NFS para redes P2P.

### 7.3.2 CDC

Como foi observado na secção 6.3, a eficiência do sistema CDC é relativamente boa. No entanto, a memória necessária no processo de descodificação é um pouco elevada. Para resolver este problema é necessário implementar um sistema de cache de *codewords* de modo a limitar o número de *codewords* em memória. No entanto, deve ser efectuado um estudo sobre a percentagem *codewords* devem permanecer na memória em função do seu grau.

Por outro lado, existem actualmente códigos Fountain mais eficientes do que o código LT, em termos do número de *codewords* necessárias à descodificação da informação. Um exemplo é o código *Raptor*. Seria interessante adaptar o CDC para o uso deste tipo de códigos e avaliar a sua eficiência em relação aos códigos LT.

### 7.3.3 Servidor de *Codewords*

A forma de distribuição e requisição de *codewords* aos SCs não é eficiente. O ideal seria implementar um protocolo de *multicast*. No processo de codificação, as *codewords* geradas no codificador seriam transmitidas em *broadcast* usando UDP para todos os SCs. Da mesma forma, o descodificador podia fazer um pedido em *multicast* para os SCs transmitirem as *codewords* referentes a um ficheiro. Os SCs respondiam com as *codewords* caso as tivessem. No processo de transmissão podiam ser perdidas algumas *codewords*, podendo ser posto em prática o objectivo dos códigos LT: a recuperação da informação em caso de perda de informação.

Por outro lado, existe um problema nas *codewords* armazenadas nos SCs. Imagine-se que um SC recebe e armazena *codewords* relativas a um ficheiro. Esse SC, por motivos de avaria, é desligado. Por sua vez, o utilizador altera esse ficheiro, que passado algum tempo é codificado e armazenado nos SCs. Quando o SC avariado é colocado a trabalhar novamente, as *codewords* referentes ao ficheiro são então inválidas. Para resolver este problema deve-se implementar uma forma de identificar versões de ficheiros. Pode-se por exemplo, associar às *codewords* uma *hash* referente ao ficheiro a que pertencem.

# Apêndice A

# Visualizador do Descodificador de *Codewords*

Para melhor visualizar o processo de descodificação foi desenvolvida uma interface gráfica (UI) que mostra o estado da descodificação. A interface foi desenvolvida usando a linguagem de programação C recorrendo ao *toolkit* GTK+ [4], detalhado no anexo B.3. A comunicação entre o componente CDC e o visualizador é feita através de canais de comunicação, denominados no sistema Linux por *pipes*.

O Visualizador do Descodificador de *Codewords* (VDC) mostra várias informações sobre o processo de descodificação. Nomeadamente, mostra informações sobre o estado da descodificação do ficheiro, o número de *codewords* recebidas e informações sobre o estado dos Servidores de *Codewords*.

A figura A.1, mostra a interface gráfica do VDC. A interface está dividida em 4 secções. Na primeira secção, é apresentada informação sobre o ficheiro que está a ser descodificado, como o caminho (*path*) e o tamanho do ficheiro. Para além disso, é apresentado o número de *codewords* recebidas até ao momento.

Na segunda secção, é visualizado o estado dos símbolos que compõem o ficheiro. Os símbolos que compõem um ficheiro são representados usando blocos. Um bloco cinzento significa que esse símbolo ainda não foi descodificado, enquanto que um bloco verde indica que já foi descodificado.

Na terceira secção, é visualizado o estado dos Servidores de *Codewords*. Os servidores de *Codewords* na base de dados são representados usando um ícone com uma descrição que indica o IP e a porta do servidor. Os servidores que estão a responder a pedidos são sinalizados com um círculo verde, enquanto que os servidores que não respondem são sinalizados com um círculo vermelho. O servidor de *Codewords* que está a fornecer *codewords* no momento é seleccionado e o fundo do ícone torna-se diferente (na figura A.1 é azul).

Finalmente, na quarta secção é mostrado uma barra de progresso que representa o progresso de descodificação do ficheiro.

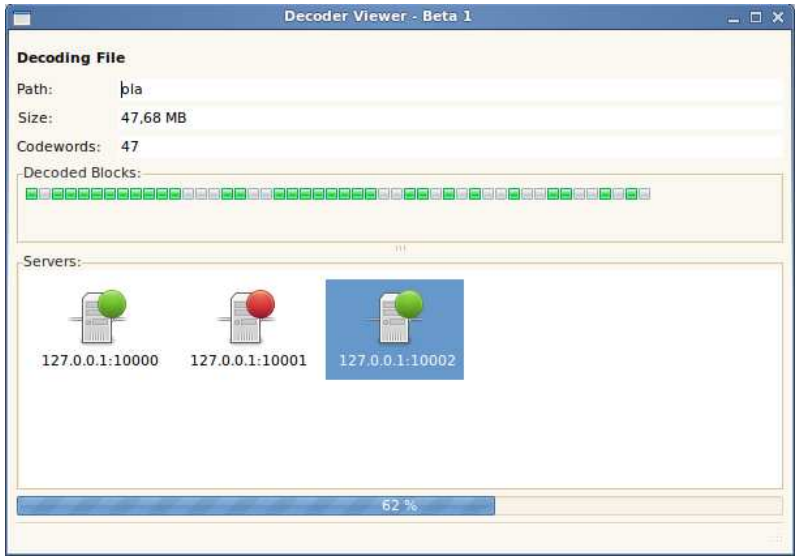


Figura A.1: Interface gráfica do VDC.

# Apêndice B

# Bibliotecas de Programação

Neste anexo são descritas brevemente algumas das bibliotecas de programação utilizadas no desenvolvimento do trabalho desta dissertação.

## B.1 GDBM

O GDBM é a implementação do projecto GNU da biblioteca DBM do sistema Unix. O GDBM implementa uma tabela de *hash* baseada em ficheiros no disco. É extremamente simples e eficiente em comparação às base de dados relacionais. É uma biblioteca *open source*. A sua utilidade é prática quando é necessário armazenar informação que pode ser recuperada rapidamente através de chaves.

## B.2 SQLite

O SQLite é uma biblioteca que implementa uma interface para aceder a base de dados através de SQL transaccional. É muito simples de usar, visto que é independente e não requer servidor, pelo que não é necessário proceder à sua configuração. É uma ferramenta *open source*.

A leitura e escrita de informação para a base de dados é efectuada directamente sobre ficheiros em disco. Apesar de não suportar todas as características que uma base dados mais complexa proporciona, é útil para a maior parte das aplicações que requerem um aceso rápido e eficiente a base de dados. Para além disso, o formato da base de dados SQLite é portável entre diferentes plataformas.

## B.3 GTK+

O GTK+ é um *toolkit* para a criação de interfaces gráficas. É multi plataforma e foi inicialmente desenvolvido como um conjunto de *widgets* para o *software* de manipulação de imagens GIMP (GNU Image Manipulation Program). Definem-se *widgets* como sendo controlos visuais que o utilizar pode interagir, como botões, *input boxes*, *combo boxes*, etc. Actualmente, é usado num grande número de aplicações, sendo principalmente usado no desenvolvimento do ambiente gráfico GNOME. O GTK+ é *software* livre e *open source*, pelo que faz parte do projecto GNU. Foi desenvolvido de modo a suportar várias linguagens. Entre elas destacam-se as linguagens C, C++, Perl, Python, Java e C#.

Existe uma ferramenta, Glade, que permite a construção de interfaces gráficas GTK+ visualmente. Com esta ferramenta o programador pode desenvolver a interface e o código do programa em separado, aumentando a produtividade. A figura B.1, mostra-se o ambiente de trabalho da ferramenta Glade. O ambiente de trabalho pode ser dividido em 4 partes essenciais como indica a figura, e que podem ser descritos da seguinte forma:

1. **Paleta de *Widgets*** Mostra todos os *widgets* que vêm por defeito e que podem ser usados numa interface gráfica GTK+.

2. **Secção de Desenho** Local onde se constrói a interface desejada. Os *widgets* podem ser colocados nesta secção.

3. **Hierarquia dos *Widgets*** Mostra a hierarquia dos *widgets* que compõem a interface gráfica.

4. **Propriedades dos *Widgets*** Mostra todos os atributos e sinais que um *widget* que o programador pode ajustar.

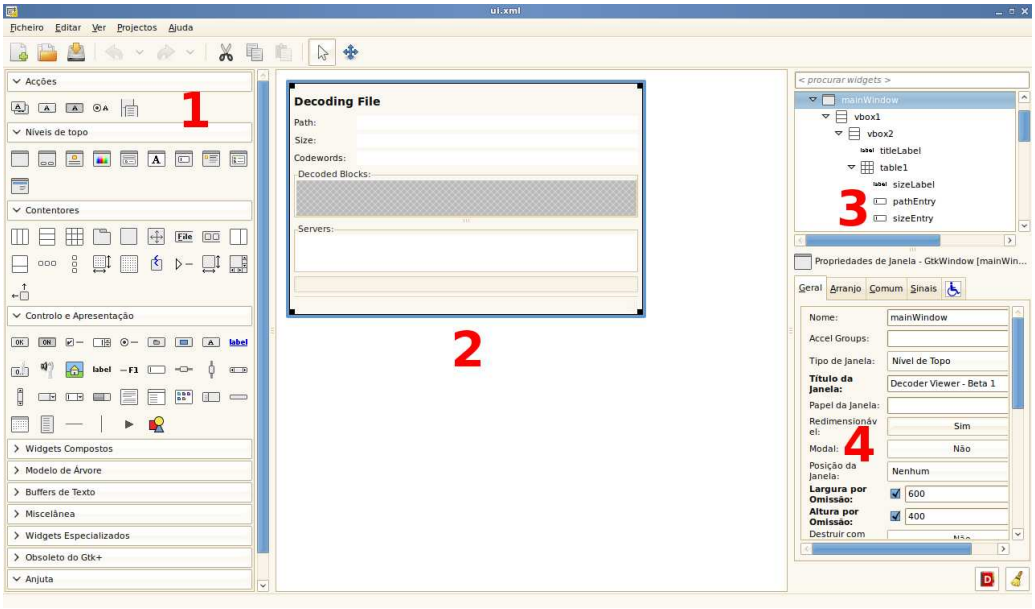


Figura B.1: Interface gráfica do Glade.

# Apêndice C

# Resultados das Simulações

## C.1 Tabela de Simulações de Codificação

Tabela C.1: Resultados das simulações de codificação.

| **Parâmetros** | **Tempo de Codificação (s)** | **% *Codewords* Grau 1** |
|---|---|---|
| $c = 0.01$ e $\delta = 0.1$ | 19.32 | 9.77 |
| | 19.43 | 7.81 |
| | 19.23 | 9.77 |
| | 20.18 | 9.11 |
| | 19.28 | 7.81 |
| | 19.21 | 10.74 |
| | 19.26 | 11.72 |
| | 19.11 | 9.77 |
| | 19.47 | 8.46 |
| | 19.74 | 8.46 |
| $c = 0.01$ e $\delta = 0.3$ | 18.84 | 6.84 |
| | 19.59 | 10.42 |
| | 18.81 | 11.07 |
| | 18.66 | 6.18 |
| | 20.29 | 9.44 |
| | 17.94 | 9.77 |
| | 19.85 | 6.84 |
| | 18.71 | 11.72 |
| | 17.47 | 8.79 |
| | 17.16 | 8.46 |
| $c = 0.01$ e $\delta = 0.5$ | 17.24 | 9.44 |
| | 18.75 | 10.74 |
| | 17.04 | 5.86 |
| | 17.48 | 8.14 |
| | 17.75 | 10.42 |
| | 18.61 | 7.81 |
| | 17.53 | 11.39 |

Continua na próxima página

**Tabela C.1 – continuação da página anterior**

| Parâmetros | Tempo de Codificação (s) | % *Codewords* Grau 1 |
|---|---|---|
| | 18.37 | 11.39 |
| | 16.97 | 11.07 |
| | 18.68 | 11.07 |
| $c = 0.01$ e $\delta = 0.7$ | 18.68 | 9.11 |
| | 16.84 | 9.11 |
| | 16.77 | 7.16 |
| | 17.59 | 7.81 |
| | 17.38 | 7.49 |
| | 17.90 | 12.37 |
| | 18.21 | 11.72 |
| | 16.86 | 7.16 |
| | 17.17 | 8.79 |
| | 18.45 | 9.77 |
| $c = 0.01$ e $\delta = 0.9$ | 16.57 | 9.11 |
| | 18.02 | 8.79 |
| | 16.43 | 11.72 |
| | 16.94 | 10.42 |
| | 15.20 | 9.44 |
| | 19.48 | 8.46 |
| | 17.00 | 4.23 |
| | 18.71 | 7.49 |
| | 17.31 | 9.77 |
| | 16.73 | 7.81 |
| $c = 0.03$ e $\delta = 0.1$ | 19.79 | 27.67 |
| | 21.54 | 28.32 |
| | 21.00 | 32.23 |
| | 21.31 | 29.62 |
| | 20.00 | 32.23 |
| | 20.03 | 27.02 |
| | 20.14 | 30.27 |
| | 21.36 | 27.99 |
| | 21.49 | 28.97 |
| | 21.50 | 28.97 |
| $c = 0.03$ e $\delta = 0.3$ | 19.99 | 27.67 |
| | 19.21 | 25.72 |
| | 19.67 | 24.09 |
| | 21.24 | 24.74 |
| | 19.85 | 26.37 |
| | 19.92 | 29.3 |
| | 21.00 | 26.37 |
| | 19.04 | 27.67 |
| | 18.94 | 21.16 |
| | 18.90 | 24.74 |

Continua na próxima página

**Tabela C.1 – continuação da página anterior**

| Parâmetros | Tempo de Codificação (s) | % *Codewords* Grau 1 |
|---|---|---|
| $c = 0.03$ e $\delta = 0.5$ | 19.08 | 26.37 |
| | 18.05 | 21.81 |
| | 18.41 | 23.11 |
| | 18.99 | 20.83 |
| | 18.77 | 20.18 |
| | 21.32 | 25.07 |
| | 18.70 | 22.79 |
| | 19.09 | 23.76 |
| | 19.15 | 21.81 |
| | 20.47 | 26.04 |
| $c = 0.03$ e $\delta = 0.7$ | 19.42 | 21.81 |
| | 19.47 | 28.65 |
| | 19.15 | 21.48 |
| | 18.75 | 23.44 |
| | 18.40 | 24.41 |
| | 19.29 | 18.23 |
| | 20.04 | 22.46 |
| | 19.68 | 23.44 |
| | 19.98 | 26.37 |
| | 18.44 | 22.46 |
| $c = 0.03$ e $\delta = 0.9$ | 18.82 | 23.76 |
| | 18.26 | 17.9 |
| | 18.76 | 27.34 |
| | 19.41 | 23.76 |
| | 17.14 | 28.65 |
| | 18.07 | 25.07 |
| | 18.22 | 25.07 |
| | 19.84 | 23.11 |
| | 19.18 | 27.34 |
| | 18.82 | 23.76 |
| $c = 0.05$ e $\delta = 0.1$ | 20.12 | 45.9 |
| | 20.67 | 42.32 |
| | 20.73 | 51.11 |
| | 21.05 | 43.29 |
| | 21.54 | 44.6 |
| | 20.40 | 49.48 |
| | 20.72 | 41.67 |
| | 20.48 | 47.2 |
| | 21.03 | 49.15 |
| | 20.46 | 53.39 |
| $c = 0.05$ e $\delta = 0.3$ | 19.09 | 39.39 |
| | 20.11 | 45.9 |
| | 19.97 | 39.71 |

Continua na próxima página

**Tabela C.1 – continuação da página anterior**

| Parâmetros | Tempo de Codificação (s) | % *Codewords* Grau 1 |
|---|---|---|
| | 19.50 | 47.53 |
| | 20.97 | 43.95 |
| | 20.49 | 42.97 |
| | 20.01 | 49.48 |
| | 19.91 | 44.27 |
| | 20.33 | 42.97 |
| | 19.10 | 37.76 |
| $c = 0.05$ e $\delta = 0.5$ | 19.85 | 38.09 |
| | 19.21 | 41.99 |
| | 19.63 | 46.88 |
| | 19.57 | 37.76 |
| | 19.32 | 35.81 |
| | 18.07 | 40.04 |
| | 20.23 | 45.57 |
| | 19.22 | 37.76 |
| | 20.89 | 41.02 |
| | 19.15 | 42.64 |
| $c = 0.05$ e $\delta = 0.7$ | 18.00 | 38.41 |
| | 18.22 | 43.29 |
| | 20.36 | 42.32 |
| | 17.59 | 45.25 |
| | 18.26 | 34.83 |
| | 18.01 | 42.32 |
| | 18.32 | 35.16 |
| | 20.09 | 41.02 |
| | 17.25 | 30.6 |
| | 17.52 | 42.97 |
| $c = 0.05$ e $\delta = 0.9$ | 19.61 | 41.67 |
| | 16.72 | 40.36 |
| | 18.31 | 40.36 |
| | 19.40 | 39.39 |
| | 18.59 | 40.69 |
| | 18.99 | 42.64 |
| | 20.43 | 38.41 |
| | 18.91 | 42.64 |
| | 22.36 | 42.97 |
| | 19.67 | 39.06 |
| $c = 0.07$ e $\delta = 0.1$ | 22.75 | 62.83 |
| | 21.99 | 69.99 |
| | 23.24 | 55.66 |
| | 22.61 | 65.43 |
| | 22.14 | 60.55 |
| | 21.30 | 60.55 |

Continua na próxima página

**Tabela C.1 – continuação da página anterior**

| Parâmetros | Tempo de Codificação (s) | % *Codewords* Grau 1 |
|---|---|---|
| | 20.47 | 60.87 |
| | 21.17 | 61.52 |
| | 21.96 | 60.22 |
| | 22.08 | 61.85 |
| $c = 0.07$ e $\delta = 0.3$ | 20.42 | 63.15 |
| | 20.52 | 61.85 |
| | 21.65 | 56.97 |
| | 21.53 | 58.59 |
| | 19.92 | 61.52 |
| | 19.72 | 56.32 |
| | 17.88 | 60.22 |
| | 17.84 | 62.17 |
| | 20.53 | 53.71 |
| | 20.08 | 52.08 |
| $c = 0.07$ e $\delta = 0.5$ | 18.44 | 54.36 |
| | 18.64 | 51.43 |
| | 19.14 | 49.15 |
| | 19.11 | 57.29 |
| | 20.14 | 61.52 |
| | 18.04 | 58.27 |
| | 20.10 | 56.97 |
| | 18.57 | 57.94 |
| | 17.81 | 50.78 |
| | 19.24 | 59.24 |
| $c = 0.07$ e $\delta = 0.7$ | 17.90 | 50.13 |
| | 17.80 | 61.2 |
| | 18.24 | 50.13 |
| | 18.80 | 51.43 |
| | 18.07 | 59.24 |
| | 17.74 | 60.22 |
| | 18.41 | 55.99 |
| | 17.61 | 56.32 |
| | 18.57 | 56.97 |
| | 18.12 | 55.34 |
| $c = 0.07$ e $\delta = 0.9$ | 18.68 | 50.78 |
| | 18.34 | 50.46 |
| | 16.92 | 54.04 |
| | 19.27 | 51.76 |
| | 17.16 | 52.41 |
| | 17.29 | 51.76 |
| | 16.81 | 52.08 |
| | 18.00 | 46.22 |
| | 17.23 | 57.62 |

Continua na próxima página

**Tabela C.1 – continuação da página anterior**

| Parâmetros | Tempo de Codificação (s) | % *Codewords* Grau 1 |
|---|---|---|
| | 16.93 | 53.71 |
| $c = 0.09$ e $\delta = 0.1$ | 19.19 | 69.66 |
| | 19.93 | 77.47 |
| | 18.57 | 81.05 |
| | 19.90 | 86.59 |
| | 19.66 | 81.05 |
| | 19.34 | 81.05 |
| | 20.02 | 72.27 |
| | 19.67 | 76.17 |
| | 19.88 | 82.36 |
| | 18.79 | 79.43 |
| $c = 0.09$ e $\delta = 0.3$ | 18.71 | 68.03 |
| | 19.23 | 72.59 |
| | 18.95 | 77.15 |
| | 18.95 | 81.38 |
| | 18.15 | 65.43 |
| | 17.91 | 67.06 |
| | 19.83 | 78.12 |
| | 18.51 | 65.76 |
| | 18.98 | 67.71 |
| | 18.64 | 72.92 |
| $c = 0.09$ e $\delta = 0.5$ | 19.96 | 69.99 |
| | 19.52 | 70.96 |
| | 17.87 | 74.22 |
| | 19.08 | 74.54 |
| | 18.10 | 73.57 |
| | 18.68 | 72.92 |
| | 18.67 | 72.27 |
| | 18.82 | 73.89 |
| | 18.65 | 73.57 |
| | 18.65 | 78.78 |
| $c = 0.09$ e $\delta = 0.7$ | 17.89 | 68.36 |
| | 16.85 | 68.03 |
| | 17.98 | 65.1 |
| | 18.23 | 70.64 |
| | 17.38 | 64.13 |
| | 17.16 | 70.96 |
| | 20.40 | 69.34 |
| | 19.39 | 65.76 |
| | 17.39 | 68.68 |
| | 17.46 | 66.73 |
| $c = 0.09$ e $\delta = 0.9$ | 19.03 | 68.68 |
| | 18.75 | 64.78 |

Continua na próxima página

**Tabela C.1 – continuação da página anterior**

| Parâmetros | Tempo de Codificação (s) | % *Codewords* Grau 1 |
|---|---|---|
| | 19.41 | 66.08 |
| | 16.59 | 68.68 |
| | 17.40 | 65.76 |
| | 17.63 | 62.17 |
| | 16.39 | 73.24 |
| | 17.44 | 67.71 |
| | 18.34 | 71.29 |
| | 18.57 | 58.92 |

## C.2 Tabela de Simulações de Descodificação

Tabela C.2: Resultados das simulações de descodificação.

| Parâmetros | *Codewords* Usadas | Tempo (s) | Memória (MB) |
|---|---|---|---|
| $c = 0.01$ e $\delta = 0.1$ | 16033 | 6.900 | 130.558 |
| | 19853 | 8.150 | 197.335 |
| | 16270 | 7.490 | 161.225 |
| | 15966 | 7.800 | 139.733 |
| | 15974 | 6.650 | 139.231 |
| | 15991 | 7.100 | 136.133 |
| | 15995 | 6.660 | 118.277 |
| | 16041 | 6.790 | 134.106 |
| | 15886 | 6.930 | 133.981 |
| | 16073 | 6.730 | 135.955 |
| $c = 0.01$ e $\delta = 0.3$ | 15888 | 5.990 | 129.154 |
| | 15966 | 6.330 | 153.888 |
| | 15980 | 6.310 | 150.521 |
| | 15886 | 6.030 | 151.824 |
| | 16072 | 6.580 | 145.214 |
| | 15967 | 6.280 | 151.954 |
| | 16049 | 6.990 | 131.740 |
| | 15959 | 6.650 | 118.357 |
| | 18433 | 6.920 | 182.442 |
| | 15844 | 6.550 | 133.159 |
| $c = 0.01$ e $\delta = 0.5$ | 15959 | 7.460 | 140.892 |
| | 16063 | 5.800 | 133.336 |
| | 17398 | 6.790 | 171.949 |
| | 15969 | 5.860 | 121.539 |
| | 16085 | 6.020 | 158.781 |
| | 16023 | 6.750 | 133.093 |
| | 15834 | 5.740 | 126.234 |
| | 15810 | 5.390 | 130.026 |
| | 15950 | 5.950 | 147.776 |

Continua na próxima página

**Tabela C.2 – continuação da página anterior**

| Parâmetros | *Codewords* Usadas | Tempo (s) | Memória (MB) |
|---|---|---|---|
| | 16007 | 5.440 | 128.443 |
| $c = 0.01$ e $\delta = 0.7$ | 15941 | 5.070 | 135.159 |
| | 16079 | 5.670 | 123.945 |
| | 16066 | 5.300 | 154.417 |
| | 15841 | 4.910 | 132.161 |
| | 15988 | 5.320 | 158.267 |
| | 16889 | 5.660 | 166.850 |
| | 16005 | 5.790 | 155.005 |
| | 16019 | 5.580 | 143.016 |
| | 15885 | 5.270 | 123.250 |
| | 16103 | 6.080 | 123.158 |
| $c = 0.01$ e $\delta = 0.9$ | 16618 | 5.520 | 164.502 |
| | 16261 | 5.630 | 161.662 |
| | 15918 | 5.560 | 133.926 |
| | 15965 | 4.940 | 136.506 |
| | 16898 | 6.010 | 166.404 |
| | 15884 | 5.190 | 125.371 |
| | 22879 | 7.740 | 227.048 |
| | 15932 | 5.910 | 136.320 |
| | 15944 | 5.700 | 133.595 |
| | 15843 | 5.460 | 147.508 |
| $c = 0.03$ e $\delta = 0.1$ | 16192 | 7.530 | 123.360 |
| | 16207 | 6.800 | 135.050 |
| | 16337 | 6.460 | 129.421 |
| | 16236 | 6.840 | 133.292 |
| | 16299 | 7.280 | 128.937 |
| | 16311 | 7.400 | 131.954 |
| | 16278 | 7.270 | 121.823 |
| | 16232 | 7.200 | 128.157 |
| | 16337 | 6.610 | 124.728 |
| | 16522 | 7.470 | 131.280 |
| $c = 0.03$ e $\delta = 0.3$ | 16197 | 6.720 | 121.144 |
| | 16043 | 6.230 | 136.403 |
| | 16194 | 6.530 | 130.926 |
| | 16185 | 6.510 | 130.939 |
| | 16168 | 6.520 | 129.486 |
| | 16141 | 7.130 | 124.623 |
| | 16006 | 6.930 | 147.347 |
| | 16170 | 7.150 | 125.181 |
| | 16181 | 6.660 | 127.774 |
| | 16111 | 7.190 | 143.725 |
| $c = 0.03$ e $\delta = 0.5$ | 16263 | 7.250 | 131.742 |
| | 16200 | 6.280 | 124.892 |

Continua na próxima página

**Tabela C.2 – continuação da página anterior**

| Parâmetros | *Codewords* Usadas | Tempo (s) | Memória (MB) |
|---|---|---|---|
| | 16221 | 6.310 | 132.652 |
| | 16015 | 6.250 | 121.446 |
| | 16436 | 6.820 | 163.356 |
| | 16329 | 7.110 | 129.892 |
| | 16288 | 6.430 | 123.855 |
| | 16122 | 6.110 | 129.416 |
| | 15994 | 6.540 | 144.234 |
| | 16031 | 6.390 | 128.058 |
| $c = 0.03$ e $\delta = 0.7$ | 16102 | 8.160 | 141.340 |
| | 16439 | 7.050 | 129.048 |
| | 16128 | 6.130 | 136.420 |
| | 16140 | 6.660 | 143.150 |
| | 16215 | 6.130 | 133.135 |
| | 16087 | 5.280 | 125.383 |
| | 16025 | 6.540 | 130.747 |
| | 16181 | 5.740 | 136.151 |
| | 16055 | 6.280 | 120.380 |
| | 16080 | 5.930 | 119.446 |
| $c = 0.03$ e $\delta = 0.9$ | 16104 | 6.010 | 130.684 |
| | 16059 | 5.530 | 139.165 |
| | 16061 | 6.070 | 129.857 |
| | 16061 | 6.250 | 135.827 |
| | 16044 | 5.580 | 131.629 |
| | 16027 | 5.730 | 129.903 |
| | 16216 | 5.950 | 118.718 |
| | 16010 | 6.120 | 134.557 |
| | 16241 | 5.930 | 134.229 |
| | 16007 | 5.870 | 130.429 |
| $c = 0.05$ e $\delta = 0.1$ | 16902 | 7.540 | 134.184 |
| | 16698 | 6.680 | 125.529 |
| | 16803 | 7.280 | 122.490 |
| | 16533 | 6.840 | 132.863 |
| | 16630 | 6.990 | 126.314 |
| | 16478 | 6.980 | 137.533 |
| | 16611 | 7.390 | 131.393 |
| | 16462 | 6.360 | 127.204 |
| | 16405 | 6.570 | 127.163 |
| | 16611 | 6.650 | 122.722 |
| $c = 0.05$ e $\delta = 0.3$ | 16419 | 6.710 | 129.325 |
| | 16448 | 6.380 | 126.286 |
| | 16422 | 7.310 | 124.410 |
| | 16398 | 7.310 | 127.332 |
| | 16432 | 6.310 | 130.570 |

Continua na próxima página

**Tabela C.2 – continuação da página anterior**

| Parâmetros | *Codewords* Usadas | Tempo (s) | Memória (MB) |
|---|---|---|---|
| | 16399 | 6.230 | 126.468 |
| | 16457 | 6.060 | 128.892 |
| | 16523 | 5.970 | 134.661 |
| | 16518 | 6.780 | 129.410 |
| | 16544 | 7.100 | 123.556 |
| $c = 0.05$ e $\delta = 0.5$ | 16335 | 7.070 | 128.279 |
| | 16349 | 7.050 | 126.637 |
| | 16535 | 9.170 | 130.298 |
| | 16448 | 6.520 | 133.237 |
| | 16335 | 6.730 | 126.697 |
| | 16330 | 6.350 | 126.440 |
| | 16315 | 6.970 | 133.193 |
| | 16481 | 6.410 | 126.082 |
| | 16480 | 6.110 | 116.289 |
| | 16461 | 6.590 | 128.036 |
| $c = 0.05$ e $\delta = 0.7$ | 16266 | 5.960 | 130.882 |
| | 16381 | 6.390 | 128.843 |
| | 16374 | 6.370 | 134.105 |
| | 16326 | 7.690 | 128.236 |
| | 16476 | 6.400 | 129.816 |
| | 16364 | 6.760 | 118.759 |
| | 16164 | 6.150 | 126.655 |
| | 16310 | 6.520 | 126.292 |
| | 16429 | 6.470 | 139.227 |
| | 16188 | 6.250 | 129.391 |
| $c = 0.05$ e $\delta = 0.9$ | 16236 | 6.380 | 132.125 |
| | 16268 | 6.240 | 129.890 |
| | 16195 | 5.810 | 127.089 |
| | 16199 | 5.730 | 130.606 |
| | 16434 | 6.020 | 128.187 |
| | 16217 | 6.070 | 134.983 |
| | 16233 | 6.330 | 133.495 |
| | 16463 | 6.380 | 131.346 |
| | 16473 | 6.610 | 119.219 |
| | 16406 | 6.160 | 123.950 |
| $c = 0.07$ e $\delta = 0.1$ | 16890 | 7.110 | 135.103 |
| | 16851 | 7.090 | 122.439 |
| | 16938 | 7.230 | 130.747 |
| | 17102 | 7.710 | 119.372 |
| | 17055 | 7.000 | 132.212 |
| | 16776 | 6.660 | 121.507 |
| | 16849 | 7.690 | 127.018 |
| | 16708 | 6.450 | 122.631 |

Continua na próxima página

**Tabela C.2 – continuação da página anterior**

| Parâmetros | *Codewords* Usadas | Tempo (s) | Memória (MB) |
|---|---|---|---|
| | 16943 | 6.920 | 120.672 |
| | 16885 | 7.390 | 121.385 |
| $c = 0.07$ e $\delta = 0.3$ | 16630 | 6.810 | 127.725 |
| | 16592 | 6.720 | 134.482 |
| | 16437 | 6.710 | 126.316 |
| | 16550 | 6.470 | 126.280 |
| | 16590 | 6.490 | 122.172 |
| | 16643 | 6.610 | 120.014 |
| | 16697 | 6.740 | 121.086 |
| | 16618 | 6.790 | 132.818 |
| | 16544 | 7.200 | 126.129 |
| | 16473 | 7.160 | 135.110 |
| $c = 0.07$ e $\delta = 0.5$ | 16493 | 6.440 | 121.455 |
| | 16543 | 6.080 | 115.726 |
| | 16744 | 6.870 | 121.623 |
| | 16410 | 5.730 | 123.743 |
| | 16442 | 6.630 | 127.575 |
| | 16714 | 6.360 | 121.947 |
| | 16574 | 6.370 | 122.692 |
| | 16755 | 6.630 | 124.672 |
| | 16555 | 6.690 | 129.343 |
| | 16432 | 6.910 | 126.809 |
| $c = 0.07$ e $\delta = 0.7$ | 16775 | 6.380 | 124.452 |
| | 16483 | 5.750 | 124.246 |
| | 16530 | 6.220 | 127.592 |
| | 16467 | 6.520 | 127.815 |
| | 16661 | 6.670 | 120.940 |
| | 16651 | 6.590 | 116.491 |
| | 16337 | 6.780 | 132.516 |
| | 16631 | 6.500 | 118.878 |
| | 16295 | 5.680 | 128.999 |
| | 16560 | 6.220 | 126.439 |
| $c = 0.07$ e $\delta = 0.9$ | 16367 | 6.260 | 131.509 |
| | 16391 | 6.100 | 136.014 |
| | 16391 | 6.360 | 141.383 |
| | 16671 | 6.230 | 123.487 |
| | 16303 | 5.780 | 129.822 |
| | 16569 | 6.650 | 124.396 |
| | 16406 | 6.380 | 133.267 |
| | 16308 | 5.850 | 146.742 |
| | 16601 | 6.390 | 130.016 |
| | 16492 | 5.910 | 124.681 |
| $c = 0.09$ e $\delta = 0.1$ | 17089 | 7.300 | 134.314 |

Continua na próxima página

**Tabela C.2 – continuação da página anterior**

| Parâmetros | *Codewords* Usadas | Tempo (s) | Memória (MB) |
|---|---|---|---|
| | 17157 | 7.380 | 126.782 |
| | 17167 | 6.900 | 133.151 |
| | 17304 | 7.210 | 129.902 |
| | 17242 | 7.490 | 128.775 |
| | 17237 | 7.140 | 126.236 |
| | 17106 | 7.870 | 125.621 |
| | 17062 | 7.210 | 126.747 |
| | 17011 | 8.150 | 121.013 |
| | 17018 | 7.100 | 128.264 |
| $c = 0.09$ e $\delta = 0.3$ | 16743 | 7.150 | 125.302 |
| | 16884 | 6.550 | 132.807 |
| | 16858 | 7.410 | 125.942 |
| | 17038 | 6.820 | 131.653 |
| | 16843 | 6.570 | 124.607 |
| | 16861 | 6.560 | 122.203 |
| | 16813 | 6.640 | 128.112 |
| | 16871 | 6.500 | 125.320 |
| | 16874 | 6.640 | 126.722 |
| | 16787 | 7.170 | 125.987 |
| $c = 0.09$ e $\delta = 0.5$ | 16887 | 6.770 | 127.582 |
| | 16639 | 6.620 | 126.998 |
| | 16768 | 6.480 | 131.237 |
| | 16817 | 6.620 | 125.254 |
| | 16926 | 6.710 | 118.018 |
| | 16712 | 6.450 | 120.400 |
| | 16789 | 6.730 | 125.448 |
| | 16902 | 6.460 | 125.537 |
| | 16974 | 6.430 | 126.897 |
| | 16857 | 6.260 | 129.402 |
| $c = 0.09$ e $\delta = 0.7$ | 16551 | 5.880 | 131.700 |
| | 16689 | 5.940 | 122.327 |
| | 16544 | 6.210 | 121.225 |
| | 16578 | 6.630 | 128.591 |
| | 16684 | 6.130 | 121.818 |
| | 16558 | 6.800 | 124.987 |
| | 16783 | 6.230 | 129.001 |
| | 16665 | 6.490 | 123.878 |
| | 16725 | 6.130 | 124.972 |
| | 16607 | 6.160 | 125.803 |
| $c = 0.09$ e $\delta = 0.9$ | 16750 | 6.780 | 126.260 |
| | 16612 | 6.480 | 122.672 |
| | 16724 | 7.200 | 126.860 |
| | 16670 | 6.530 | 129.111 |

Continua na próxima página

**Tabela C.2 – continuação da página anterior**

| Parâmetros | *Codewords* Usadas | Tempo (s) | Memória (MB) |
|---|---|---|---|
| | 16678 | 6.420 | 119.064 |
| | 16750 | 6.880 | 118.482 |
| | 16611 | 6.940 | 124.066 |
| | 16677 | 6.230 | 128.075 |
| | 16644 | 6.110 | 134.156 |
| | 16525 | 6.210 | 122.110 |

# Bibliografia


[1] http://oceanstore.cs.berkeley.edu/.

[2] http://valgrind.org/.

[3] http://www.gnu.org/software/gdbm/.

[4] http://www.gtk.org/.

[5] http://www.sqlite.org/.

[6] R. Ahlswede, Ning Cai, S. Y. R. Li, and R. W. Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, 2000.

[7] David Arthur and Rina Panigrahy. Analyzing bittorrent and related peer-to-peer networks. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 961–969, New York, NY, USA, 2006. ACM Press.

[8] Matt Blaze. A cryptographic file system for unix. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 9–16, New York, NY, USA, 1993. ACM.

[9] Ali Raza Butt, Troy A. Johnson, Yili Zheng, and Y. Charlie Hu. Kosha: A peer-to-peer enhancement for the network file system. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 51, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Dah M. Chiu, R. W. Yeung, Jiaqing Huang, and Bin Fan. Can network coding help in p2p networks? In *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks, 2006 4th International Symposium on*, pages 1–5, 2006.

[11] Christina Fragouli, Jean-Yves Le Boudec, and Jörg Widmer. Network coding: an instant primer. *SIGCOMM Comput. Commun. Rev.*, 36(1):63–68, January 2006.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, volume 37, pages 29–43, New York, NY, USA, 2003. ACM.

[13] C GKANTSIDIS, J MILLER, and P RODRIGUEZ. Anatomy of a p2p content distribution system with network coding. In *In Proceedings of the Second International Peer to Peer Symposium (IPTPS.*

[14] C. Gkantsidis and P. R. Rodriguez. Network coding for large scale content distribution. volume 4, pages 2235–2245 vol. 4, 2005.

[15] Christos Gkantsidis, John Miller, and Pablo Rodriguez. Comprehensive view of a live network coding p2p system. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 177–188, New York, NY, USA, 2006. ACM Press.

[16] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, page 19, Berkeley, CA, USA, 1994. USENIX Association.

[17] Ashish Khisti. Tornado codes and luby transform codes.

[18] S. Y. R. Li, R. W. Yeung, and Ning Cai. Linear network coding. *Information Theory, IEEE Transactions on*, 49(2):371–381, 2003.

[19] Andreas Ljungquist, Jonas Claesson, and Tobias Bondesson. Raid technology, 2003.

[20] M. Luby. Lt codes. pages 271–280, 2002.

[21] D.J.C. MacKay. Fountain codes. *Communications, IEE Proceedings-*, 152(6):1062–1068, Dec. 2005.

[22] M. Mitzenmacher. Digital fountains: a survey and look forward. In *Information Theory Workshop, 2004. IEEE*, pages 271–276, Oct. 2004.

[23] Tobias Oetiker, Irene Hyna, Hubert Partl, and Elisabeth Schlegl. The not so short introduction to latex2, 2008.

[24] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[25] Dongyu Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. *SIGCOMM Comput. Commun. Rev.*, 34(4):367–378, October 2004.

[26] Yousef Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, April 2003.

[27] S. Sarvotham, D. Baron, and R.G. Baraniuk. Sudocodes - fast measurement and reconstruction of sparse signals. In *Information Theory, 2006 IEEE International Symposium on*, pages 2804–2808, July 2006.

[28] Sun Microsystems. Xdr: External data representation standard, 1987.

[29] Sun Microsystems. Rpc: Remote procedure call protocol specification, 1988.

[30] Sun Microsystems. Nfs: Network file system protocol specification, 1989.