UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de São José do Rio Preto

André Francisco Morielo Caetano

Griddler: uma estratégia configurável para armazenamento distribuído de objetos peer-to-peer que combina replicação e erasure coding com sistema de cache

São José do Rio Preto
2017

André Francisco Morielo Caetano

Griddler: uma estratégia configurável para armazenamento distribuído de objetos peer-to-peer que combina replicação e erasure coding com sistema de cache

Dissertação apresentada como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação – Área de Concentração em Computação Aplicada, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Câmpus de São José do Rio Preto.

Financiadora: CAPES/DS

Orientador: Prof. Dr. Carlos Roberto Valêncio

São José do Rio Preto
2017

Caetano, André Francisco Morielo.
Griddler: uma estratégia configurável para armazenamento distribuído de objetos peer-to-peer que combina replicação e erasure coding com sistema de cache / André Francisco Morielo Caetano. -- São José do Rio Preto, 2017
83 f. : il., gráfs., tabs.

Orientador: Carlos Roberto Valêncio
Dissertação (mestrado) – Universidade Estadual Paulista “Júlio de Mesquita Filho”, Instituto de Biociências, Letras e Ciências Exatas

1. Computação. 2. Big data. 3. Banco de dados – Gerenciamento. 4. Banco de dados distribuído. 5. Arquitetura de computador. 6. Cache (Computadores) I. Universidade Estadual Paulista "Júlio de Mesquita Filho". Instituto de Biociências, Letras e Ciências Exatas. II. Título.

CDU – 681.3.025

Ficha catalográfica elaborada pela Biblioteca do IBILCE
UNESP - Câmpus de São José do Rio Preto

André Francisco Morielo Caetano

Griddler: uma estratégia configurável para armazenamento distribuído de objetos peer-to-peer que combina replicação e erasure coding com sistema de cache

Dissertação apresentada como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação – Área de Concentração em Computação Aplicada, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Financiadora: CAPES/DS

Comissão Examinadora

Prof. Dr. Carlos Roberto Valêncio
IBILCE/UNESP - São José do Rio Preto (SP)
Orientador

Prof. Dr. Geraldo Francisco Donegá Zafalon
IBILCE/UNESP - São José do Rio Preto (SP)

Prof. Dr. Pedro Luiz Pizzigatti Corrêa
POLI/USP - São Paulo (SP)

São José do Rio Preto
10 de Agosto de 2017

À meus pais

AGRADECIMENTOS

Agradeço muito ao Professor Carlos Roberto Valêncio pela oportunidade que meu deu no Grupo de Banco de Dados, aceitando-me como seu orientado logo no primeiro ano do curso de Graduação, ainda que eu tivesse à época passado por duas reprovações. Desde então evolui muito a nível pessoal e como estudante, o que certamente teria sido muito mais difícil sem a experiência adquirida no GBD. Hoje, com a conclusão do Mestrado, espero ter correspondido às expectativas depositadas em mim, reiterando sentimentos de gratidão pelos últimos 6 anos que passei no laboratório sob sua orientação.

Ainda entre os docentes dos quais fui aluno ao longo dos anos, faço aqui um agradecimento póstumo em especial ao Professor José Márcio Machado, o qual eu acompanhava em todas as disciplinas sempre que possível, mesmo as disciplinas optativas. Embora não seja do conhecimento de muitos, sempre admirei sua genialidade fora do comum, e a influência dele na Graduação foi determinante para que eu decidisse seguir o caminho da Pós-Graduação, como modelo de pesquisador, acadêmico e cientista.

Dos amigos, agradeço a Guilherme Priólli Daniel, que também trilhou o caminho do Mestrado quase lado a lado comigo, e indiretamente me ajudou até este dia. À Fábio Renato de Almeida, agradeço pelo conselhos e por servir como modelo em diversos sentidos para meu projeto de pesquisa. E também agradeço aos atuais membros da Equipe de Infraestrutura, Gabriel, Luis e Gustavo, que assumiram minhas tarefas no laboratório nesse último semestre para que eu pudesse terminar minha dissertação.

Agradeço muito a meus pais, pela compreensão e ajuda que me deram nos últimos anos. Poder morar com minha família me garantiu a estabilidade financeira e emocional necessária para desenvolver o projeto de Mestrado com muita tranquilidade. A eles, o reconhecimento merecido e todo o meu amor.

Por fim, agradeço a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa de estudos de Mestrado, obtida através do Programa de Pós-Graduação em Ciência da Computação da UNESP.

“I think it is possible for ordinary people to choose to be extraordinary.”
Elon Musk

RESUMO

Sistemas de gerenciamento de banco de dados, na sua essência, almejam garantir o armazenamento confiável da informação. Também é tarefa de um sistema de gerenciamento de banco de dados oferecer agilidade no acesso às informações. Nesse contexto, é de grande interesse considerar alguns fenômenos recentes: a progressiva geração de conteúdo não-estruturado, como imagens e vídeo, o decorrente aumento do volume de dados em formato digital nas mais diversas mídias e o grande número de requisições por parte de usuários cada vez mais exigentes. Esses fenômenos fazem parte de uma nova realidade, denominada Big Data, que impõe aos projetistas de bancos de dados um aumento nos requisitos de flexibilidade, escalabilidade, resiliência e velocidade dos seus sistemas. Para suportar dados não-estruturados foi preciso se desprender de algumas limitações dos bancos de dados convencionais e definir novas arquiteturas de armazenamento. Essas arquiteturas definem padrões para gerenciamento dos dados, mas um sistema de armazenamento deve ter suas especificidades ajustadas em cada nível de implementação. Em termos de escalabilidade, por exemplo, cabe a escolha entre sistemas com algum tipo de centralização ou totalmente descentralizados. Por outro lado, em termos de resiliência, algumas soluções utilizam um esquema de replicação para preservar a integridade dos dados por meio de cópias, enquanto outras técnicas visam a otimização do volume de dados armazenados. Por fim, ao mesmo tempo que são desenvolvidas novas tecnologias de rede e disco, pode-se pensar na utilização de *caching* para otimizar o acesso ao que está armazenado. Este trabalho explora e analisa os diferentes níveis no desenvolvimento de sistemas de armazenamento distribuído. O objetivo deste trabalho é apresentar uma arquitetura que combina diferentes técnicas de resiliência. A contribuição científica deste trabalho é, além de uma sugestão totalmente descentralizada de alocação dos dados, o uso de uma estrutura de *cache* de acesso nesse ambiente, com algoritmos adaptáveis.

Palavras-chave: big data, armazenamento, sistemas distribuídos, peer-to-peer, dados não-estruturados, armazenamento de objetos

ABSTRACT

Database management systems, in essence, aim to ensure the reliable storage of information. It is also the task of a database management system to provide agility in accessing information. In this context, it is of great interest to consider some recent phenomena: the progressive generation of unstructured content such as images and video, the consequent increase in the volume of data in digital format in the most diverse media and the large number of requests by users increasingly demanding. These phenomena are part of a new reality, named Big Data, that imposes on database designers an increase in the flexibility, scalability, resiliency, and speed requirements of their systems. To support unstructured data, it was necessary to get rid of some limitations of conventional databases and define new storage architectures. These architectures define standards for data management, but a storage system must have its specificities adjusted at each level of implementation. In terms of scalability, for example, it is up to the choice between systems with some type of centralization or totally decentralized. On the other hand, in terms of resiliency, some solutions utilize a replication scheme to preserve the integrity of the data through copies, while other techniques are aimed at optimizing the volume of stored data. Finally, at the same time that new network and disk technologies are being developed, one might think of using caching to optimize access to what is stored. This work explores and analyzes the different levels in the development of distributed storage systems. This work objective is to present an architecture that combines different resilience techniques. The scientific contribution of this work is, in addition to a totally decentralized suggestion of data allocation, the use of an access cache structure with adaptive algorithms in this environment.

Keywords: big data, storage, distributed systems, peer-to-peer, unstructured data, object storage

LISTA DE ILUSTRAÇÕES

Página

2.1 Crescimento no volume do universo de dados (GANTZ; REINSEL, 2012). 6
2.2 Cluster do Facebook: falhas em 1 mês (SATHIAMOORTHY et al., 2013). . 7
2.3 Esquema simplificado de representação de blocos de dados 9
2.4 Esquema simplificado de representação de estrutura de arquivos 10
2.5 Esquema simplificado de representação de um objeto de dados 12
2.6 Desenho de um arquitetura do tipo mestre-escravo 13
2.7 Desenho de um arquitetura do tipo *peer-to-peer* 14
2.8 Fluxo de restauração de um código MDS (5,3) 18
2.9 Exemplo de funcionamento de um código Regenerador 19
2.10 À esquerda código hierárquico (2,1) e, à direita, código hierárquico (4,3) . . 21
2.11 Comparação da taxa de acerto do ARC versus o LRU (adaptado do trabalho (MEGIDDO; MODHA, 2004) 24

3.1 Comparação do tempo de execução da cifgra SHA-1 e outros algoritmos, adaptado de (MAQABLEH, 2011). 29
3.2 Geração de GUID para cada um dos nós da rede 30
3.3 Geração da rede *peer-to-peer* em anel 31
3.4 Modelo em alto nível de um dos nós da rede Griddler 32
3.5 Rede com indicativos da tabela de roteamento do nó 0 34
3.6 Primeiro nó de uma rede na arquitetura Griddler 35
3.7 Representação das etapas a serem realizadas para inserção de um novo nó 36
3.8 Segundo nó de uma rede na arquitetura Griddler 37
3.9 Rede com alguns objetos inseridos 40
3.10 Representação do acesso ao cache em cada um dos nós do sistema distribuído 43

4.1 Gráfico da latência de acesso com e sem o uso de *cache* para dados replicados 46
4.2 Gráfico da latência de acesso com e sem o uso de *cache* para dados codificados 46
4.3 Tempo decorrido na codificação de diferentes conjuntos de dados 53
4.4 Tempo decorrido na decodificação de diferentes conjuntos de dados 54
4.5 Comparação de sobrecarga para ambas as técnicas de redundância 56

LISTA DE TABELAS

Página

2.1 Comparação entre os trabalhos correlatos 26

3.1 Tabela de roteamento do nó 0 do ambiente P2P 33

4.1 Latência média para diferentes objetos, replicação em 3x 45
4.2 Latência média para diferentes objetos, codificados 46
4.3 Observações de latência para diferentes objetos, replicação em 3x, sem cache 47
4.4 Observações de latência para diferentes objetos, replicação em 3x, com cache 48
4.5 Observações de latência para diferentes objetos, codificados, sem cache . 49
4.6 Observações de latência para diferentes objetos, codificados, com cache . 50
4.7 Tempos de codificação para diferentes volumes de dados 53
4.8 Tempos de decodificação para diferentes volumes de dados 54
4.9 Sobrecarga para redundância no armazenamento de dados binários 57

5.1 Comparação entre os trabalhos correlatos e o trabalho proposto 60

LISTA DE ABREVIATURAS E SIGLAS

3D	Três dimensões
AES	Advanced Encryption Standard
API	Application Programming Interface
ARC	Adaptive Replacement Cache
ATA	Advanced Technology Attachment
CIFS	Common Internet File System
CPU	Central Processing Unit
DHT	Distributed Hashing Table
E/S	Entrada e Saída (de dados)
FCP	Fibre Channel Protocol
GPGPU	General purpose GPU Computing
GPU	Graphics Processing Unit
GUID	Globally Unique Identifier
HDD	Hard Disk Drive
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
IDC	International Data Corporation
IP	Internet Protocol
iSCSI	Internet Protocol SCSI
LFU	Least Frequently Used
LRU	Least Recently Used
LVM	Logical Volume Manager
MBR	Minimum Bandwith Regenerating
MDS	Maximum Distance Separable
NAS	Network Attached Storage
NDSS	Network Distributed Storage Systems

OID	Object ID
P2P	Peer-to-Peer
PCI	Peripheral Component Interconnect
RAID	Redundant Array of Independent Disks
REST	Representational State Transfer
SAN	Storage Area Network
SAS	Serial Attached SCSI
SATA	Serial ATA
SCSI	Small Computer System Interface
SHA	Secure Hashing Algorithm
SMB	Server Message Block
SSD	Solid State Drive
TCP	Transmission Control Protocol
XFS	X File System

SUMÁRIO

Página

1 Introdução . . . 1
1.1 Considerações iniciais . . . 1
1.2 Motivação . . . 1
1.3 Objetivos . . . 3
1.4 Organização do Trabalho . . . 3

2 Fundamentação Teórica . . . 5
2.1 Considerações iniciais . . . 5
2.2 Desafios de armazenamento de dados . . . 5
2.3 Sistemas de armazenamento distribuídos em rede . . . 6
2.4 Paradigmas de armazenamento de dados . . . 8
2.4.1 Armazenamento de Blocos . . . 8
2.4.2 Armazenamento de Arquivos . . . 8
2.4.3 Armazenamento de Objetos . . . 9
2.5 Arquiteturas de sistemas distribuídos . . . 12
2.5.1 Arquitetura do tipo mestre-escravo . . . 12
2.5.2 Arquitetura do tipo peer-to-peer . . . 13
2.6 Modelos de tolerância a falhas . . . 14
2.6.1 Replicação de Dados e Códigos de Correção de Erros . . . 15
2.6.2 Códigos MDS (Reed-Solomon) . . . 16
2.6.3 Códigos Regeneradores (MBR) . . . 17
2.6.4 Códigos Localmente Reparáveis (Hierárquicos) . . . 19
2.7 Tecnologias de disco . . . 20
2.7.1 Discos magnéticos e Discos de estado sólido . . . 20
2.8 Tecnologias de processamento . . . 22
2.8.1 Processadores convencionais e Processamento gráfico (GPGPU) 22
2.9 Algoritmos de *Caching* . . . 22
2.10 Trabalhos Correlatos . . . 24
2.10.1 CAROM - Cache A Replica On Modification . . . 24
2.10.2 MICS - Mingling Chained Storage . . . 25
2.10.3 HRSPC - Hybrid Redundancy Scheme Plus Computing . . . 25
2.10.4 Robot - Big data storage system based on erasure coding . . . 25
2.10.5 HDFS-Xorbas - a module for erasure code in HDFS . . . 25
2.10.6 Análise dos trabalhos correlatos . . . 26
2.11 Considerações Parciais . . . 26

3 Armazenamento P2P com tolerância a falhas híbrida e sistema de cache 28
3.1 Considerações iniciais 28
3.2 Descrição e requisitos do ambiente distribuído 28
3.3 Tabela de roteamento 32
3.4 Inserção de um novo nó 34
3.4.1 Inserção quando não existem outros nós na rede 34
3.4.2 Inserção quando existem outros nós na rede 36
3.5 Remoção de um nó 37
3.5.1 Remoção prevista pelos usuários 37
3.5.2 Remoção devido a falhas e imprevistos 38
3.6 Operações básicas de interação com o sistema 38
3.6.1 Inserção de dados na forma de objetos 39
3.6.2 Busca de dados na forma de objetos 40
3.6.3 Remoção de dados na forma de objetos 41
3.6.4 Atualização de dados na forma de objetos 41
3.7 Mecanismo de *caching* distribuído com estratégia ARC 42
3.8 Considerações Parciais 43

4 Experimentos e Resultados 44
4.1 Considerações iniciais 44
4.2 Ambiente de testes e Conjunto de dados 44
4.3 Latência de acesso 45
4.4 Codificação e Decodificação 53
4.5 Sobrecarga de armazenamento 55
4.6 Discussão dos resultados e próximos passos 55
4.7 Considerações Parciais 58

5 Considerações Finais 59
5.1 Discussão sobre o trabalho desenvolvido 59
5.2 Contribuições do trabalho 60
5.3 Trabalhos futuros 60

Referências Bibliográficas 62

CAPÍTULO 1 – Introdução

1.1 Considerações iniciais

Atualmente, o armazenamento resiliente de grandes volumes de dados, ou *Big Data*, é um dos mais relevantes problemas a serem tratados em termos de infraestrutura de suporte na ciência da computação (ALNAFOOSI; STEINBACH, 2013) (HASHEM et al., 2015). Isso significa que todo armazenamento de dados deve ser feito de tal maneira que os mesmos nunca sejam perdidos, independentemente de falhas ou fatores externos comuns a qualquer ambiente computacional, como um dano a um disco rígido. Ao mesmo tempo, é essencial otimizar o tempo de resposta às requisições feitas sobre os dados, tendo em vista as limitações de velocidade no acesso das atuais mídias de armazenamento secundário e as exigências crescentes por parte de usuários e aplicações que necessitam de interações ágeis com o que está armazenado.

Muitas das tecnologias de vanguarda existentes em termos de tolerância a falhas utilizam uma abordagem de replicação, em que um certo grau de redundância é acrescentado aos dados, ao copiá-los e armazená-los em locais diferentes, muitas vezes distantes geograficamente (GONIZZI et al., 2015). Embora a técnica de replicação tenha se mostrado razoavelmente eficiente em diversos cenários e ainda seja utilizada em diferentes contextos, possui suas desvantagens. A mais evidente é o aumento da capacidade necessária em disco para armazenar um determinado conjunto de dados, o que implica em uma maior sobrecarga em cada atualização para manter as cópias idênticas, bem como incrementos nos custos de tempo e recursos de *hardware* (WEATHERSPOON; KUBIATOWICZ, 2002). Nesse sentido, novas técnicas têm sido progressivamente estudadas e introduzidas em ambientes distribuídos, com destaque para métodos que utilizem códigos de correção de erros, também conhecidos como *erasure codes* (KHAN et al., 2012). Quanto à otimização no acesso aos dados, a capacidade de explorar a latência reduzida de mídias como a memória principal torna o uso de técnicas de *caching* uma alternativa relevante. Contudo, também para estas técnicas cabe uma análise cuidadosa para sua aplicação em ambientes de armazenamento distribuído.

1.2 Motivação

A confiabilidade, velocidade no acesso e tolerância a falhas no armazenamento dos dados são altamente relevantes, pois busca-se a garantia de que os dados, frutos de significativos conjuntos de esforços ao longo das atividades dos empreendimentos ou pesquisas, possam ser utilizados a fim de proporcionar vantagens competitivas ou resultados de relevância aos estudos e pesquisas acadêmicas. Este trabalho encontra

motivação nos desafios de pesquisa na área, que hoje é alvo de estudo não somente de iniciativas da academia, mas também de empresas de maior porte.

Em análises com relação às tecnologias de vanguarda mais utilizadas para armazenamento de dados, se destaca o alto grau de adoção, impacto de negócio e maturidade dos códigos de correção de erros (RINNEN, 2016), que compõem algoritmos de dispersão da informação em sistemas distribuídos, o que tem estimulado cada vez mais novas iniciativas nesta área. Em paralelo a isto, a aplicabilidade da codificação dos dados para armazenamento tem sido, há algum tempo, estudada em diversas situações reais. Na área da saúde, por exemplo (TZOVARAS et al., 1998), imagens em três dimensões, 3D, são uma realidade consolidada, tanto no planejamento de cirurgias quanto em simulações de radioterapia. Tais imagens devem ser armazenadas confiavelmente por longos períodos de forma a preservar os diagnósticos relacionados, sem perdas. É nos trabalhos mais recentes, no entanto, que se encontra uma maior dedicação a esse tema, pois percebeu-se que os esquemas de resiliência utilizados para armazenamento atualmente tendem a não suportar o volume incremental de dados existentes e gerados a todo momento. Com isto foram desenvolvidas novas bibliotecas de *software* (PLANK; GREENAN, 2014) (CURRY et al., 2011) (TIAN, 2014), arquiteturas de armazenamento (YIN et al., 2013) (TANG et al., 2015) (MA et al., 2013) (LI et al., 2016) e mesmo sistemas de arquivos distribuídos (BIAN; SEKER, 2013), todos baseados em códigos de correção de erros e algoritmos de dispersão da informação, com bons resultados. Porém exaustivas comparações recentes ainda comprovam a oportunidade de espaço para melhorias em diversos sentidos (DENG et al., 2014), como em termos de uso da rede. Conforme será discutido mais adiante, nem todos os códigos são otimizados para consumo de banda de rede, e os que o são acabam tendo alguma outra desvantagem. Em diferentes corporações, como o Facebook, isso é um problema, visto que inviabiliza a plena utilização de codificação desse nível, pois comprovou-se por simulações que isto provocaria uma saturação total nos *links* de rede devido à grande quantidade de acessos aos dados armazenados (SATHIAMOORTHY et al., 2013). Então este é um ramo ainda em aberto para pesquisa. Outras empresas como o Google têm passado a utilizar códigos de correção de erros, especialmente os códigos ditos MDS, do inglês *Maximum Distance Separable*, a serem vistos mais adiante neste documento (FIKES, 2010), porém ainda não se chegou a um consenso quanto à melhor forma de implantar este tipo de tecnologia, devido às diversas nuances inerentes aos diversos métodos existentes. Como tanto a técnica de replicação como a técnica de codificação de dados tem seus pontos positivos e negativos, há ainda trabalhos relativamente recentes que sugerem que o ideal é combinar ambas para obter o melhor em termos de tolerância a falhas com

eficiência no armazenamento (GRIBAUDO; IACONO; MANINI, 2016).

Para contornar a sobrecarga adicional que algoritmos associados a códigos de correção de erros impõem sobre o tráfego na rede de um sistema de armazenamento distribuído, uma alternativa possível é utilizar algum tipo de *cache* com acesso otimizado. Esse tipo de estratégia permite que o acesso aos dados, mesmo quando estes são codificados, não passe pelo acesso a diversos nós na rede para, ao invés disso, direcionar o fluxo das requisições primariamente ao *cache*. Por esse motivo essa estrutura de acesso precisa ser construída sobre mídias com acesso mais rápido do que os dispositivos de armazenamento secundário. Em geral a memória principal, ou mais recentemente os discos de estado sólido, cumprem esse papel e são o principal alvo de estudo de trabalhos recentes na área (AGGARWAL et al., 2016) (ABHIJITH et al., 2016) (LIN et al., 2016), embora ainda haja espaço para novas pesquisas e melhorias, especialmente no que tange a integrar essas técnicas com arquiteturas de armazenamento distribuído.

1.3 Objetivos

Diante de tal motivação, o objetivo deste documento é descrever os resultados obtidos no desenvolvimento do trabalho de pesquisa associado, de modo a corroborar o esforço investido no estudo dos desafios mencionados anteriormente para apresentar uma solução inovadora. O objetivo geral desse trabalho é a análise de diferentes técnicas para armazenamento de dados voltadas à tolerância a falhas e velocidade no acesso ao que está armazenado. Os objetivos específicos desse trabalho consistem no desenvolvimento de uma estratégia aprimorada para dispersão e armazenamento confiável de dados por meio de tolerância a falhas mista, e com foco no aprimoramento do acesso através de técnicas de *caching* em um ambiente totalmente descentralizado. Toma-se por hipótese que a introdução de técnicas de codificação de dados em sistemas descentralizados de armazenamento distribuído contribui para garantia de tolerância a falhas, em especial quando associada à replicação, e é possível escolher o melhor método para cada conjunto de dados que se deseja armazenar. Essa hipótese é reforçada pela escolha do algoritmo adequado para *caching* dos dados, o que pode melhorar consideravalmente o acesso aos dados armazenados, em especial quando estes se encontram codificados.

1.4 Organização do Trabalho

A seguir é apresentada a estrutura deste trabalho:

- Capítulo 2 – Fundamentação teórica: apresentação dos principais conceitos de sistemas distribuídos de armazenamento de dados, tecnologias afins, replicação, códigos de correção de erros e algoritmos de *caching*. Descrição dos problemas associados e apresentação de trabalhos correlatos recentes da área.
- Capítulo 3 – Arquitetura proposta: apresentação do esquema de armazenamento sugerido e todas as suas particularidades.
- Capítulo 4 – Experimentos e resultados: descrição de testes realizados e dos resultados obtidos. Análise e discussão dos resultados obtidos em comparação com os trabalhos correlatos.
- Capítulo 5 – Conclusão: considerações finais referentes ao trabalho apresentado, bem como avaliação de possíveis melhorias e trabalhos futuros.

CAPÍTULO 2 – Fundamentação Teórica

2.1 Considerações iniciais

Neste capítulo são apresentados aspectos teóricos relacionados com as seguintes áreas de estudo em ciência da computação: sistemas de armazenamento distribuído, paradigmas de armazenamento, técnicas de replicação e técnicas de codificação de dados. Estas duas técnicas são utilizadas para acrescentar alguma redundância e tolerância a falhas controlada, de modo que, quando for necessária a recuperação, os dados estejam sempre disponíveis. Também são apresentadas e discutidas as técnicas de *caching* mais comuns. Do ponto de vista prático, o estudo destes tópicos encontra aplicações em algumas áreas importantes da ciência da computação, conforme será visto a seguir.

2.2 Desafios de armazenamento de dados

Como resultado de processos de negócio, monitoramento de atividades, sensores, dentre outros fatores, há uma tendência não só nas organizações, mas em todos os contextos, de aumento nos volumes de dados gerados e armazenados diariamente. Redes sociais e *websites* permitem ainda que usuários criem registros completos de suas vidas ao postar diariamente suas atividades, lugares visitados, fotos exibidas e preferências pessoais. Essa quantidade expressiva de informações é frequentemente referenciada como *Big Data*, um termo que busca destacar os desafios existentes no tratamento destes dados em termos de armazenamento, interoperabilidade, governança e análise (GANDOMI; HAIDER, 2015) (ASSUNÇÃO et al., 2015). Duas características recorrentes em diversos desses conjuntos de dados são o volume e variedade, o que indica um comportamento de crescimento constante e a não-restrição aos dados tidos como convencionais, encontrados em formato texto em tabelas e bancos de dados relacionais (ASSUNÇÃO et al., 2015) (JIN et al., 2015). Dessa forma, dois desafios importantes estão em como tratar tipos não-estruturados em conjuntos maiores de dados e como otimizar o uso de disco de modo a garantir a disponibilidade frente a falhas comuns ou mesmo desastres de infraestrutura em larga escala. Em termos de armazenamento, estes são alguns dos fatores principais a serem considerados, e o uso de replicação, códigos de correção de erros e algoritmos de dispersão se mostra uma alternativa válida a ser considerada nesse sentido. Na Figura 2.1 é apresentado um indicativo do crescimento do universo digital, previsto pela IDC - *International Data Corporation* - há alguns anos, do qual boa parte é composta de dados não-estruturados. Em paralelo, é possível observar na Figura 2.2 um exemplo da quantidade de falhas que acontecem em um ambiente real de armazenamento distribuído, em que foi avaliado

um cluster do Facebook com 3000 computadores. Por meio da observação de ambas as imagens é possível perceber o contraste entre a importância do que tem sido criado em termos de dados e o que uma infraestrutura computacional de alta disponibilidade necessita em termos de tolerância a falhas. É, portanto, essencial entender que tipos de arquiteturas e sistemas computacionais são necessários para trabalhar com esse nível de desafios.

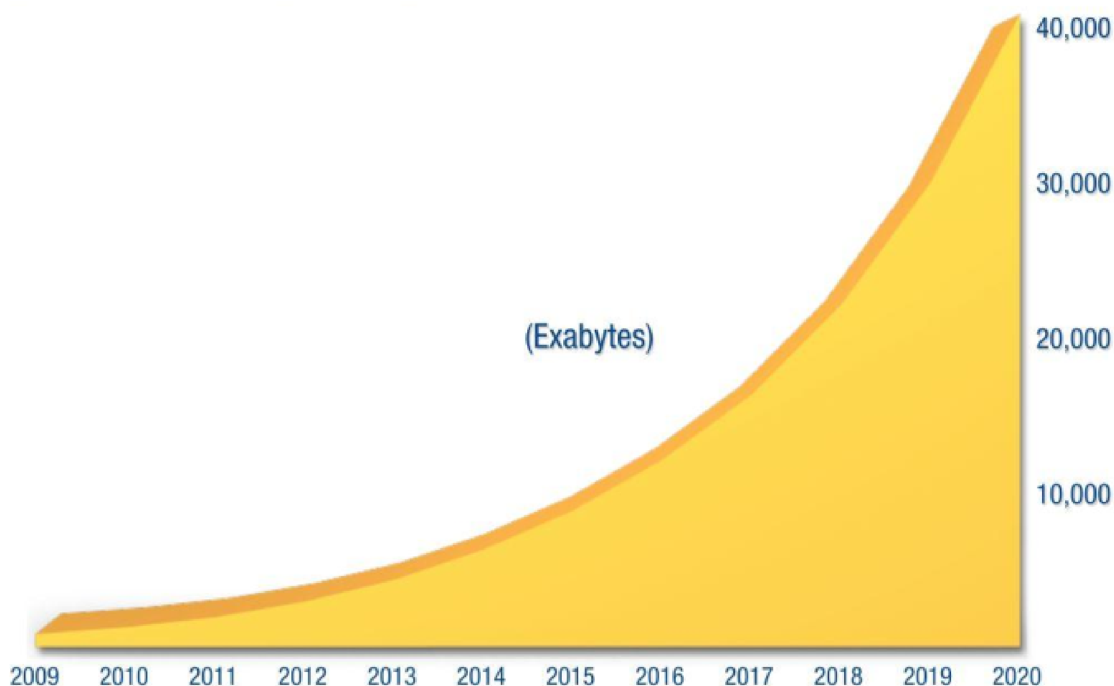


Figura 2.1: Crescimento no volume do universo de dados (GANTZ; REINSEL, 2012).

2.3 Sistemas de armazenamento distribuídos em rede

Embora as tecnologias de tolerância a falhas e disponibilidade, como os arrays de discos independentes (em inglês, *Redundant Array of Independent Disks* - RAID), tenham se desenvolvido amplamente no contexto dos sistemas de armazenamento mais atuais, em paralelo novas tecnologias surgiram, as quais utilizam diversas unidades lógicas de armazenamento – ou simplesmente nós de armazenamento – que atuam em conjunto para aumentar a capacidade de provisionamento do sistema como um todo. Isso ocorre pelo fato de haver um volume significativo de dados gerados a cada dia, o que torna difícil e custoso construir um único dispositivo computacional com capacidade de armazenamento e entrada/saída, E/S, suficientes para suportar essa carga (DATTA; OGGIER, 2013). Ao tratar de sistemas distribuídos em rede entenda-se esse tipo de sistema de armazenamento, que agrupa recursos de diferentes nós conectados

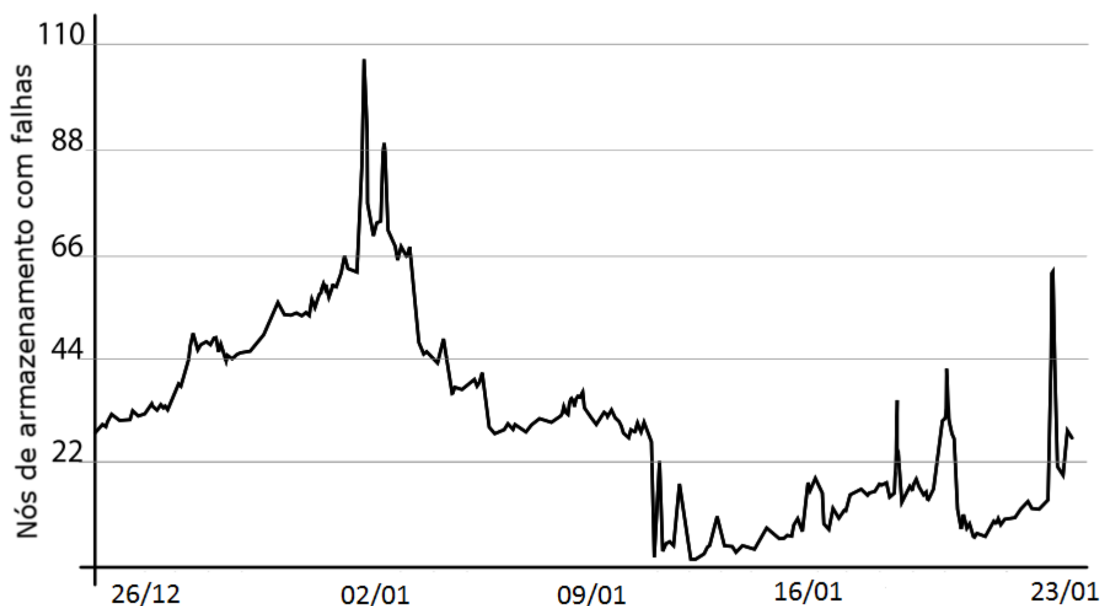


Figura 2.2: Cluster do Facebook: falhas em 1 mês (SATHIAMOORTHY et al., 2013).

entre si, os quais individualmente podem ou não utilizar tecnologias específicas como RAID. Os dados são distribuídos por meio de diversas unidades de armazenamento interconectadas, daí o nome sistemas de armazenamento distribuídos em rede – do inglês *networked distributed storage systems* (NDSS) (OGGIER; DATTA, 2013).

Os NDSS podem caracterizar diversos tipos de sistemas e arquiteturas, tais como *datacenters* e sistemas de armazenamento em Nuvem ou *backup peer-to-peer* (P2P), cada um com suas particularidades, mas que compartilham características em comum. Dado que estes sistemas geralmente tomam proporções significativas, e são compostos muitas vezes por centenas ou milhares de nós, a falha de um nó individual ou mesmo de outros componentes da rede acaba se tornando uma norma, e não uma exceção (DATTA; OGGIER, 2013). Por esse motivo, e com o objetivo de oferecer uma alta disponibilidade geral para os serviços fornecidos, é primordial garantir tolerância tanto a interrupções temporárias quanto a falhas permanentes de componentes individuais do sistema. A tolerância a falhas é obtida por meio de redundância, e a resiliência a longo prazo é obtida por meio da restauração da redundância perdida ao longo do tempo por qualquer falha. Nesse sentido os códigos de correção de erros, do inglês *erasure codes*, se tornaram bastante populares pois garantem a resiliência de um sistema e implicam em uma sobrecarga de armazenamento relativamente baixa. Em trabalhos mais recentes é recorrente a pesquisa em termos de modelos de códigos de correção de erros feitos sob medida para atender as necessidades de sistemas de armazenamento distribuídos, particularmente com destaque para melhorias em termos de reparabilidade do sistema (PAMIES-JUAREZ; OGGIER; DATTA, 2013). A seguir

são apresentados e explicados alguns outros conceitos relacionados a tecnologias de armazenamento de dados que devem ser levados em consideração na concepção de sistemas de armazenamento distribuídos em rede.

2.4 Paradigmas de armazenamento de dados

Nas próximas seções são descritos três dos principais paradigmas de armazenamento de dados existentes, cujas características permitem justificar sua aplicação em diferentes contextos. No texto é dado maior destaque à tecnologia de armazenamento de objetos, pois ela tem sido o foco de diversas pesquisas recentes.

2.4.1 Armazenamento de Blocos

Nesse tipo de estratégia são criadas e gerenciadas sequências de tamanho fixo de *bits*, chamadas de *blocos*, do dispositivo de armazenamento. Essas sequências podem ter apenas alguns *bytes* ou mesmo ocupar algumas dezenas de *megabytes*, como é o caso de algumas tecnologias mais recentes (SHVACHKO et al., 2010). Para armazenamento em blocos, o sistema operacional obrigatoriamente se conecta aos dispositivos de armazenamento instalados no computador. Os dados ficam então disponíveis por meio de uma série de interfaces e clientes, tais como: Canais de Fibra (ou *Fibre Channel Protocol*, FCP), SCSI (*Small Computer System Interface*) e iSCSI (*Internet Protocol SCSI*), SAS (*Serial Attached SCSI*), ATA (*Advanced Technology Attachment*) e SATA (*Serial ATA*).
Algumas dessas tecnologias são mais comumente utilizadas para acesso a dispositivos de armazenamento alocados fora do computador, como em uma SAN, rede de área de armazenamento, do inglês *Storage Area Network*. Contudo discos SAS e ATA são mais utilizados da forma convencional, conectados diretamente ao computador, sem equipamentos intermediários (EDITION, 2014). Na Figura 2.3 é possível observar um esquema de armazenamento em blocos.

2.4.2 Armazenamento de Arquivos

A estratégia de armazenamento em arquivos tem como diferencial a utilização de uma estrutura pré-definida de diretórios. Sistemas conectados à rede que armazenam dados na forma de arquivos são conhecidos como NAS - *Network Attached Storage*. Normalmente estes dispositivos atuam da mesma forma que um servidor computacional comum, e tem os seus próprios processadores. Para acessar os dados, são utilizados os protocolos padrão TCP/IP. Alguns dos protocolos mais comuns são: SMB (*Server Message Block*) ou CIFS (*Common Internet File System*), que é comumente usado em redes baseadas no Windows, NFS (*Network File System*), que é comum em redes

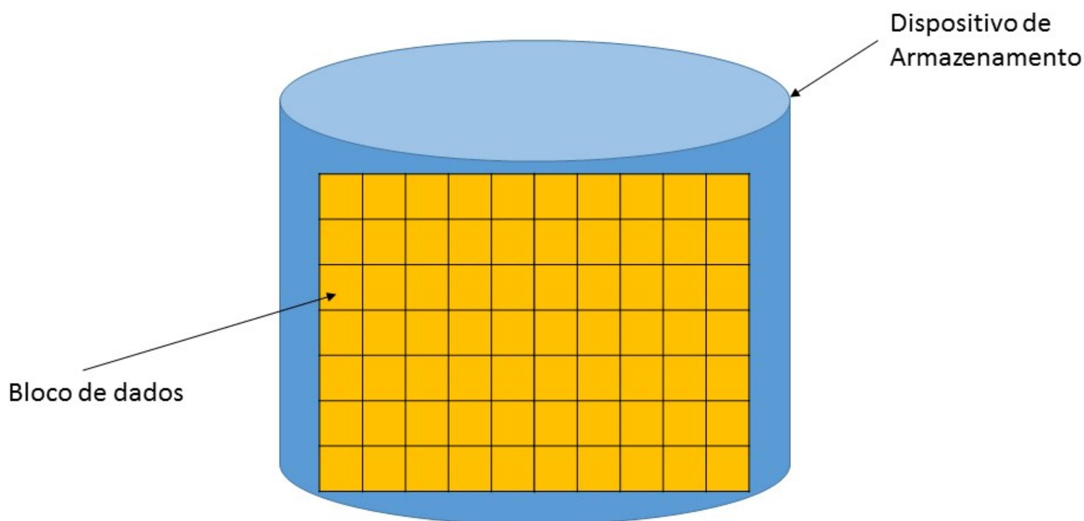


Figura 2.3: Esquema simplificado de representação de blocos de dados

baseadas em Unix/Linux, e HTTP (*Hypertext Transfer Protocol*), o protocolo padrão para acesso via navegadores *web* (EDITION, 2014).
Esses tipos de sistemas de armazenamento são fáceis de implantar e o acesso do cliente é simples, pois é feito por meio de um ou mais dos protocolos mencionados anteriormente. Independente do sistema operacional, dado que todos os dispositivos na rede estejam conectados entre si de forma compartilhada, os dados de sistemas do tipo NAS poderão ser acessados, e por esse motivo ainda são utilizados em diversos contextos. No entanto, sistemas desse tipo possuem algumas desvantagens significativas que devem ser levadas em conta. Eles são normalmente mais lentos que sistemas de armazenamento baseados em acesso direto a blocos, visto que necessitam de processamento adicional. Ao mesmo tempo, dispositivos NAS também têm escalabilidade limitada. Quando um dispositivo NAS esgota seus recursos de disco, é possível adicionar outro dispositivo em paralelo. Porém, como estes dispositivos a princípio não interagem entre si, ocorre o fenômeno das *ilhas de armazenamento*, que são ineficientes para se gerir (MA et al., 2014). Na Figura 2.4 é possível observar uma parte da estrutura de arquivos de um sistema operacional Linux.

2.4.3 Armazenamento de Objetos

Sistemas de armazenamento baseados em objetos usam uma estrutura nova, chamada de *container*, para armazenar dados na forma de objetos em um espaço de endereço plano ao invés de utilizar os sistemas de arquivos hierárquicos, baseados em diretório, que são comuns em sistemas de armazenamento baseados em blocos e

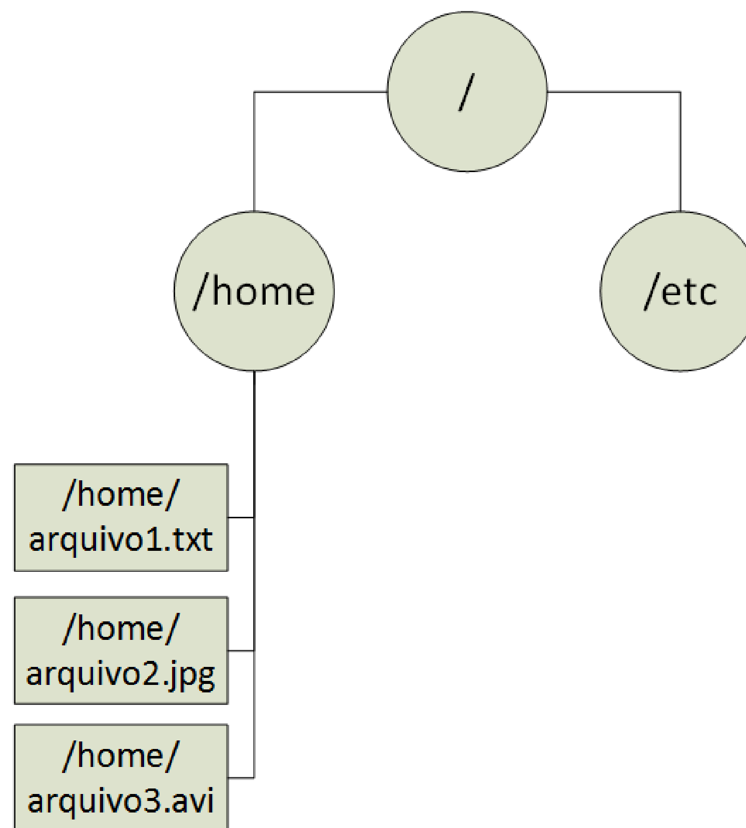


Figura 2.4: Esquema simplificado de representação de estrutura de arquivos

arquivos. Um *container* armazena os dados reais (por exemplo, uma imagem ou vídeo), os metadados (por exemplo, data, tamanho, tipo), e um OID, *Object ID*, único (FACTOR et al., 2005). Cada OID é armazenado em um banco de dados ou aplicativo à parte e é usado para fazer referência a objetos em um ou mais *containers*. Os dados são acessados via protocolo HTTP por meio de um navegador *web* ou diretamente por meio de uma API como REST, Transferência de Estado Representacional, do inglês *Representational State Transfer*. Essa API implementa funções importantes como PUT, GET e DELETE para interagir com os objetos armazenados. O espaço de endereço plano em um sistema de armazenamento permite tratar o disco e a memória como um único espaço contíguo e ignorar situações como fragmentação e paginação. Por esse motivo, o armazenamento baseado em objetos permite simplicidade e escalabilidade massiva, mas os dados nesses sistemas normalmente não podem ser modificados, e devem ser completamente eliminados e uma versão inteiramente nova escrita em seu lugar (MESNIER; GANGER; RIEDEL, 2003). Porém essa é uma opção interessante pois torna possível guardar versões de um mesmo objeto em diferentes estados e com diferentes modificações. Armazenamento com base em objetos é comumente utilizado para serviços na Nuvem por fornecedores, como IBM SoftLayer, Amazon S3, Google

e Facebook, e devido a isso existem tecnologias de mercado que fornecem soluções bastante completas nesse sentido (KAPADIA; RAJANA; VARMA, 2015).

Armazenamento de objetos é diferente de armazenamento em blocos e arquivos, pois virtualiza totalmente a implementação física da apresentação lógica. É semelhante ao *check-in* de bagagem em aeroportos, em que a bagagem é colocada no sistema de esteiras sem que saiba onde será depositada ou qual caminho percorrerá. Há apenas a garantia de retirada da bagagem no destino final da viagem. Se você usa uma bagagem de mão, você tem que saber exatamente o lugar em que ela está em todos os momentos, o que analogamente para armazenamento de dados pode ser custoso. O foco de armazenamento de objetos é, portanto, o *scale-out*, ou seja, o uso de sistemas distribuídos em larga escala (EDITION, 2014). Cada nó, desse sistema de armazenamento maior pode utilizar localmente um sistema de arquivos, mas a ideia de arquiteturas de armazenamento objeto é permitir a utilização de *hardware* não-especializado, ou *hardware commodity*, ao contrário de equipamentos caros e difíceis de lidar utilizados em sistemas de armazenamento tradicionais. As tarefas mais importantes de um sistema de armazenamento de objetos são as seguintes:

- Alocação de dados (*placement*)
- Automatização de tarefas de gerenciamento, inclusive a garantia de durabilidade e disponibilidade

Normalmente, um usuário envia seu pedido GET, PUT ou DELETE para qualquer nó de armazenamento do sistema distribuído, e o pedido é traduzido para os dispositivos de armazenamento por parte do *software* de gerenciamento de objetos. O *software* também cuida do modelo de durabilidade ao fazer uso de técnicas como replicação e códigos de correção de erros. O modelo de durabilidade em geral não é RAID devido às dificuldades de escalabilidade dessa tecnologia quando o volume dos dados atinge a ordem de centenas de *terabytes*. Ao mesmo tempo é preciso ter algum mecanismo automatizado para tratar de tarefas críticas de gestão, tais como verificações periódicas do estado dos nós, auto-correções, e migração de dados. A administração também é facilitada pela abstração do endereçamento plano, o que significa que um administrador pode gerenciar todo o sistema distribuído como se fosse uma única entidade (EDITION, 2014). Na Figura 2.5 é ilustrada uma representação de um objeto, no qual há espaço dedicado tanto para o armazenamento dos dados quanto para o armazenamento dos metadados, além de uma sequência exclusiva pela qual é possível identificar um único objeto dentre todos os que estiverem armazenados.

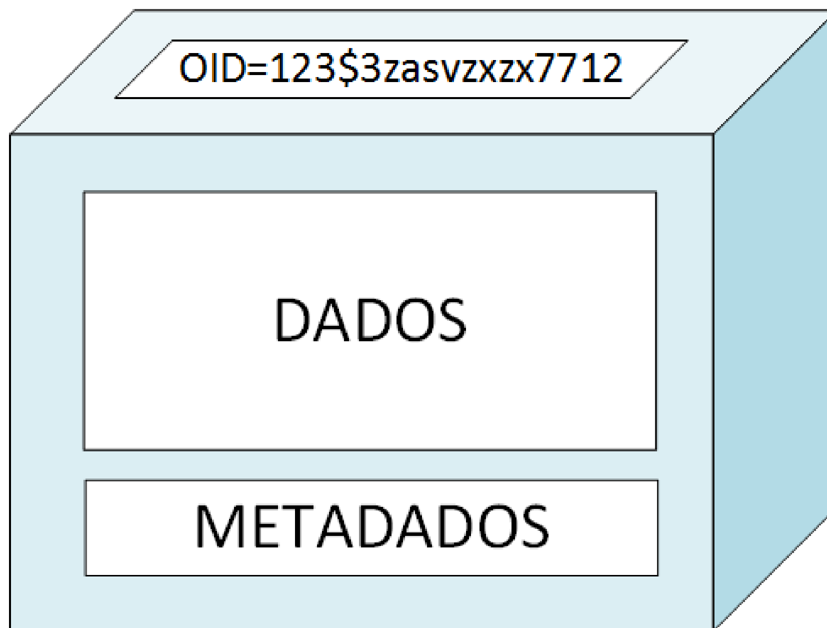


Figura 2.5: Esquema simplificado de representação de um objeto de dados

2.5 Arquiteturas de sistemas distribuídos

A seguir são descritas duas das arquiteturas mais comuns de sistemas distribuídos, as quais apresentam características, até certo ponto, antagônicas. Por esse motivo, é necessário detalhar os motivos que levam à escolha da arquitetura mais adequada para um determinado trabalho.

2.5.1 Arquitetura do tipo mestre-escravo

Talvez o paradigma mais tradicional para sistemas distribuídos, e um padrão extensamente adotado para *clusters* de computadores, a arquitetura do tipo mestre-escravo divide em dois grupos os recursos computacionais disponíveis. Computadores com nível *mestre* são responsáveis pelo pré-processamento das tarefas recebidas pelo sistema. Ao mesmo tempo, um computador *mestre* atribui as tarefas aos computadores *escravos*, que por sua vez são responsáveis pela real execução das ordens recebidas. Em geral, para se obter o melhor desempenho de um sistema desse tipo, o ideal é que o número de tarefas nunca ultrapasse o número de processadores disponíveis nos escravos. Aplicações deste paradigma incluem principalmente computação paralela (SAHNI; VAIRAKTARAKIS, 1996). Em programação paralela é possível desenvolver um único programa que permita o uso de desvios, ou *forks*, para lançar múltiplas linhas, ou *threads*, de execução. A operação de desvio envolve a passagem de diferentes quantidades de dados para os computadores *escravos*. Esses computadores, por sua vez, retornam os resultados para a linha de execução principal do programa, controlada

pelo *mestre*.

Existem de fato alguns trabalhos recentes que utilizam essa arquitetura para sistemas de armazenamento (QIN et al., 2015), dentre os quais diversos se baseiam no HDFS - *Hadoop Distributed File System* - um sistema de arquivos distribuído de uso geral (SATHIAMOORTHY et al., 2013) (KO; ZAW, 2014) (RASHMI et al., 2014). Porém outros trabalhos também apontaram o problema com essa arquitetura, que cria efeitos de afunilamento na rede, em que um *mestre* se torna um ponto crítico de falha, pois se este falhar, ainda que os demais computadores funcionem, o sistema fica indisponível (CAIWEI; LEI; LIANSHENG, 2012). Na Figura 2.6 é ilustrado um exemplo de uma arquitetura do tipo mestre-escravo em que um único nó mestre é responsável por atender e distribuir solicitações para três nós escravos, que ficam encarregados de todo o processamento adicional.

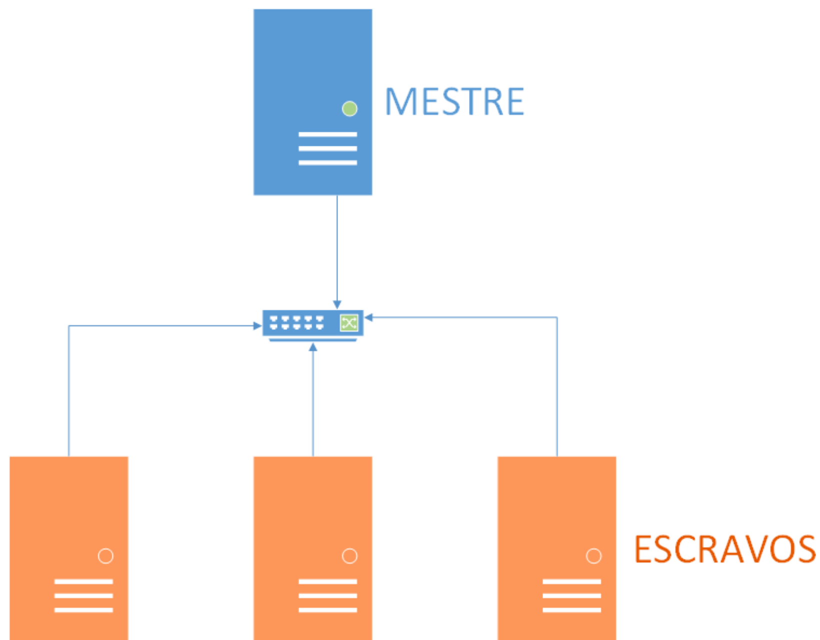


Figura 2.6: Desenho de um arquitetura do tipo mestre-escravo

2.5.2 Arquitetura do tipo peer-to-peer

Sistemas *peer-to-peer* (P2P) surgiram inicialmente como um fenômeno social, de modo a ser uma arquitetura para compartilhamento de recursos computacionais, como ciclos de CPU (ANDERSON, 2004) ou para troca de arquivos (RIPEANU, 2001). Em um modelo P2P não existe o conceito de computadores servidores e especializados, que tenham uma função central de gerência do sistema distribuído como um todo. Ao invés disso essa função é atribuída a todos os membros da rede, que podem ser computadores com *hardware commodity*. O conceito principal a ser levado em conta

quando se trata de sistemas de armazenamento P2P é que os dados são distribuídos entre cada um dos *peers* para que um nível alto de confiabilidade e tolerância a falhas seja obtido, com um custo geral reduzido. Dessa forma há uma liderança compartilhada, e não há um ponto único de falha (RIPEANU, 2001).

Por todos esses motivos tem sido dada muita atenção a esse tipo de arquitetura em diversos trabalhos recentes em armazenamento de dados, haja vista que algumas de suas características superam limitações da arquitetura do tipo mestre-escravo (PARK; SONG, 2016) (DELL'AMICO et al., 2015) (CARON et al., 2014) (ESINER; DATTA, 2016) (MARTALÒ et al., 2014). Na Figura 2.7 é ilustrado um exemplo de uma arquitetura do tipo *peer-to-peer*, para a qual não há o papel de um nó gerenciador principal. Ao contrário, todos os nós são igualmente responsáveis por atender e processar solicitações e cada *peer* interage com os demais conforme o necessário.

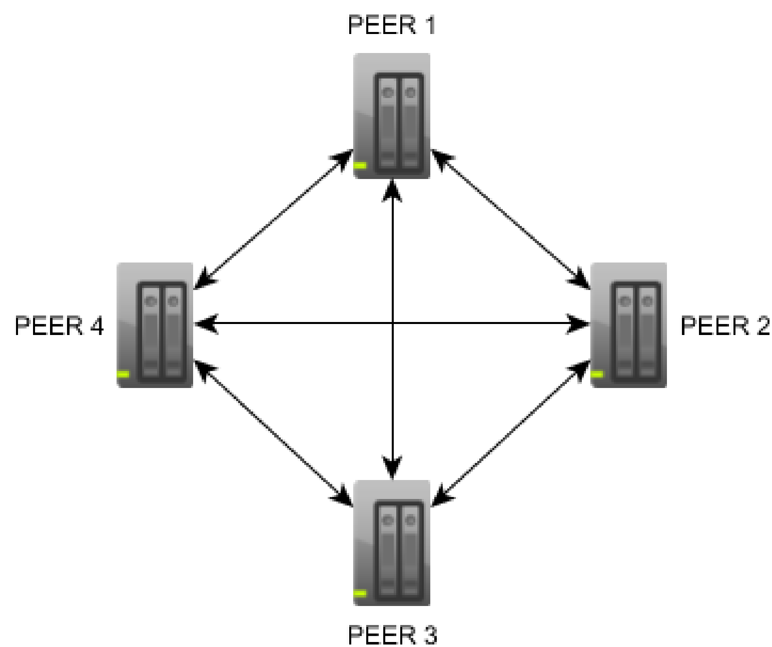


Figura 2.7: Desenho de um arquitetura do tipo *peer-to-peer*

2.6 Modelos de tolerância a falhas

Nesta seção são realizadas algumas explicações sobre os modelos de tolerância a falhas mais utilizados em diversos trabalhos da literatura, quais sejam: replicação e códigos de correção de erros. Espera-se com isso mostrar os pontos positivos e negativos de cada um deles, no sentido de fornecer a base teórica para trabalhos cujo objetivo seja garantir a tolerâncias a falhas em sistemas de armazenamento distribuído.

2.6.1 Replicação de Dados e Códigos de Correção de Erros

Em computadores e nas telecomunicações digitais, os dados são representados na forma binária, isto é, uma sequência de bits que tem o valor 0 ou 1. Essa sequência pode se dividir naturalmente em unidades como octetos ou *bytes* que representam caracteres específicos (LINT, 2012). Toda essa representação é feita de forma bastante direta, de modo que o maior problema não está na representação binária dos dados, mas na sua transmissão por canais de comunicação ou no seu armazenamento. Isto ocorre porque estas funções podem acarretar em interferências que podem causar erros nos dados, por meio da adição, remoção e alteração de um ou mais *bits* (PURSER; HOUSE, 1995).

A forma mais comum de garantir a confiabilidade no armazenamento dos dados é a replicação. Essa técnica consiste em apenas criar uma ou mais cópias inteiras de um determinado conjunto de dados, geralmente em computadores ou discos rígidos diferentes daquele em que foram gravados originalmente. Essa estratégia, apesar de eficiente em termos de acesso aos dados armazenados, incorre em uma sobrecarga extra de uso dos dispositivos de armazenamento, haja vista a necessidade de reservar espaço para as cópias. Técnicas de correção de erros, por outro lado, introduzem redundância controlada nos dados, a fim de mitigar possíveis problemas, de modo que mensagens corrompidas durante sua transmissão ou armazenamento possam ser corrigidas antes de qualquer processamento adicional (PURSER; HOUSE, 1995). Ou seja, apenas uma parte dos *bits* transmitidos ou armazenados são dados válidos. Esse tipo de técnica, os códigos de correção de erros, possuem ampla e complexa herança matemática originada em um ramo de estudo conhecido como teoria de códigos (LINT, 2012). Além de benefícios claros para a transmissão de dados, observou-se que alguns subconjuntos dentre todos os tipos de códigos estudados nessa área do conhecimento podem ser utilizados em aplicações de armazenamento de dados, notoriamente os códigos MDS – *Maximum Distance Separable*. Embora este tipo de código seja o foco de diferentes trabalhos atuais, também existem outras variantes de codificação que tem sido estudadas em menor escala, porém a estas também são dadas as devidas explicações nas próximas seções. É importante destacar, no entanto, alguns compromissos importantes que a técnica de codificação, do inglês *erasure coding*, impõe, a saber (CARPENTIER, 2013):

- **Maiores perdas de dados do que a replicação quando o número de discos com falha cresce** - no caso da replicação, enquanto ainda houver uma cópia intacta a perda de dados é minimizada. No caso dos códigos, cada falha resulta em uma perda parcial do que está armazenado, porém a vantagem neste caso é

a fácil restauração;

- **Os melhores resultados só são realmente obtidos em sistemas distribuídos de grande porte** - é estatisticamente vantajoso possuir um sistema distribuído grande o suficiente, em termos da quantidade de nós computacionais que atuam em conjunto, para tolerar falhas individuais de forma eficiente. A depender da construção do código, isto acaba por se tornar uma necessidade;
- **Os melhores resultados só são realmente obtidos ao tratar arquivos realmente grandes** - para arquivos relativamente pequenos, por exemplo um arquivo de 100KB, o ganho em armazenamento é irrelevante se comparado ao custo de processamento e transmissão dos dados. Tais características certamente os tornam mais úteis em contextos como o de *Big Data*, e embora a perda de dados seja de fato maior do que com o uso da simples replicação, a capacidade de restaurar os dados de maneira eficiente é um fator a se considerar e neste caso os códigos possuem qualidades mais favoráveis.

Nas seções seguintes se encontram descrições dos principais códigos trabalhados na literatura da área, os quais se dividem em três classes principais: MDS, Regeneradores e Localmente Reparáveis. É importante mencionar que para cada uma das classes de códigos mencionadas podem existir diversas derivações de códigos em desenvolvimento, e por esse motivo em cada seção foi destacado apenas um exemplo para ser descrito em maiores detalhes. Porém é necessário assumir que todos os demais códigos existentes em cada classe respeitam as devidas regras e características de sua categoria.

2.6.2 Códigos MDS (Reed-Solomon)

No contexto de *erasure coding*, códigos ditos MDS – acrônimo inglês para *Maximum Distance Separable* – são aqueles que fornecem eficiência ótima para armazenamento. Destes, o tipo mais conhecido que tem sido utilizado e estudado ao longo dos anos é o dos códigos de *Reed-Solomon* (REED; SOLOMON, 1960). Para entender tais códigos, no entanto, é preciso previamente compreender que seu uso está voltado principalmente a dados na forma de objetos, que em última instância são tratados como arquivos. Cada um desses objetos pode ser armazenado em n discos rígidos. Dado um número k arbitrário, em que $k < n$, códigos $MDS(n,k)$ fornecem a garantia de tolerar até um máximo de $n-k$ falhas de discos, o que implica que k discos são suficientes para acessar quaisquer bits dos dados originais. Especificamente, o objeto de dados é codificado em n blocos por meio de métodos algébricos ou operações lógicas, e esses blocos devem ser uniformemente disseminados em n discos

rígidos (SUH; RAMCHANDRAN, 2010). No caso de códigos de *Reed-Solomon*, a codificação cria símbolos de um campo finito Fq, de tamanho q, e cada símbolo é armazenado em um nó diferente, ou seja, cada símbolo gerado contém parte dos dados originais. Suponha que o tamanho total do objeto de dados a ser armazenado seja de M bits. Então o volume armazenado em cada nó é equivalente a M/k bits, se os metadados associados a esse objeto não forem considerados. Nesse sentido, a eficiência de armazenamento de códigos MDS é na melhor das hipóteses k/n (LI; LI, 2013).

Ao se comparar com a replicação em 3 vias é possível implementar um código $MDS(5,3)$, conforme ilustrado na Figura 2.8, que ainda tolere no máximo duas falhas de discos rígidos, enquanto que ao mesmo tempo melhore a eficiência de armazenamento em 80%. Para acessar o objeto de dados o sistema precisa acessar k blocos codificados diferentes de k discos diferentes e recuperar os dados originais por meio de um algoritmo de decodificação, que varia de acordo com o tipo de código MDS utilizado. Para *Reed-Solomon*, a decodificação usa um método algébrico com aritmética de campos finitos. Entretanto esse algoritmo de decodificação inerentemente acarreta em aumento na latência de acesso dos discos. Em vários casos, é razoável buscar recuperar todo o arquivo, com o objetivo de garantir o acesso aos dados. Entretanto, do ponto de vista do sistema de armazenamento em si, é desnecessário recuperar um objeto completo se somente é necessário reparar, eventualmente, um pequeno bloco codificado que foi danificado, o que corresponde a apenas uma fração do objeto de dados original. Essa característica é presente em todos os códigos MDS, e sofre alterações apenas em outra categoria de códigos chamados Regeneradores, que são descritos adiante. Porém no caso MDS, é preciso acessar pelo menos k discos para reparar apenas um único disco. Como os dados em cada nó são armazenados em símbolos de tamanho M/k, esse acesso implica em transferir não menos que M bits pela rede, que é o tamanho do arquivo original. É válido lembrar que no caso da replicação, para reparar uma réplica é preciso acessar apenas uma única dentre as demais réplicas. Esse requisito pode aumentar dramaticamente tanto a E/S dos discos quanto gerar uma sobrecarga de utilização de banda de rede em um *datacenter*, e isto afeta significativamente a performance tanto do sistema de armazenamento quanto das demais aplicações hospedadas na mesma Nuvem computacional (LI; LI, 2013).

2.6.3 Códigos Regeneradores (MBR)

No contexto de *erasure coding*, códigos ditos Regeneradores, do inglês *Regenerating Codes*, são aqueles que fornecem eficiência ótima para banda de rede. Destes, um exemplo são os códigos MBR – acrônimo inglês para *Minimum Bandwith*

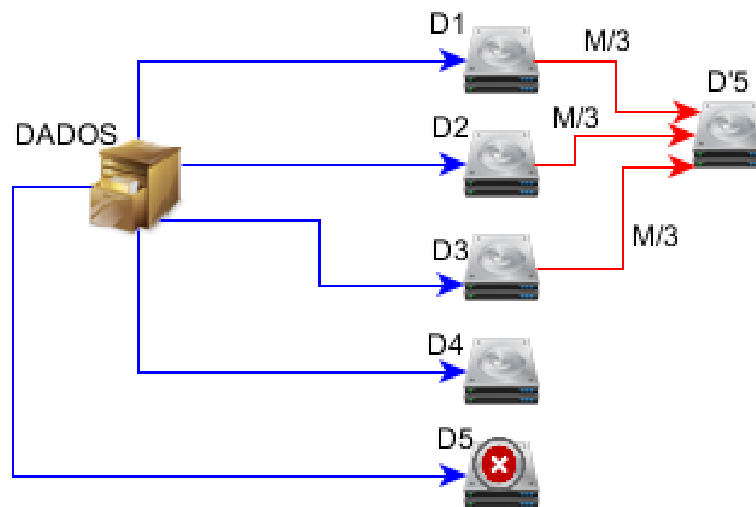


Figura 2.8: Fluxo de restauração de um código MDS (5,3)

Regenerating. Esta classe de código se utiliza do compromisso de armazenar em cada nó uma quantidade extra de dados redundantes para se beneficiar de reparos mais eficientes, em que a quantidade de *bits* transferidos seja exatamente a necessária para restaurar um conjunto de dados, sem os excessos de códigos MDS (RASHMI et al., 2009).

A diferença desse tipo de código para os códigos MDS começa pela forma com que as informações são armazenadas em cada nó individualmente. Ao invés de tratar as informações em cada nó como apenas um símbolo pertencente a um campo finito, trabalha-se com o armazenamento de vetores de símbolos em cada nó. Ou seja, cada nó armazena α símbolos dentro de Fq, em que $\alpha > 1$. Nessa configuração fica claro que é possível para qualquer nó individual transferir apenas uma parcela dos dados que armazena (LI; LI, 2013).

Fora este novo parâmetro α, dois outros parâmetros d e β, são associados com códigos regeneradores. Por definição códigos regeneradores permitem que um nó com falhas se conecte a um conjunto arbitrário de $d \geq k$ nós dos $(n–1)$ nós restantes, e transfira $\beta \leq \alpha$ símbolos de cada nó. Veja que a constante k ainda é presente nestes códigos, assim como nos códigos MDS. Enquanto nestes compõe a definição do total armazenado em cada nó como um valor M/k, no caso de códigos Regeneradores MBR esse valor chega a $2Md/k(2d-k+1)$. O total de dados transferidos para fins de reparo, seja $d\beta$, é denominado de *repair bandwidth*. Em códigos regeneradores típicos o valor médio de $d\beta$ para *repair bandwidth* é pequeno se comparado ao tamanho original M do arquivo armazenado, o que é um ganho se comparado a códigos do tipo MDS (SHAH

et al., 2012). Na Figura 2.9 é apresentado um modelo simples de funcionamento de um código Regenerador, em que durante uma falha do nó 1 ocorre o acesso a um número maior de nós computacionais, porém com uma quantidade menor de dados transferidos pela rede quando comparado a um código do tipo MDS.

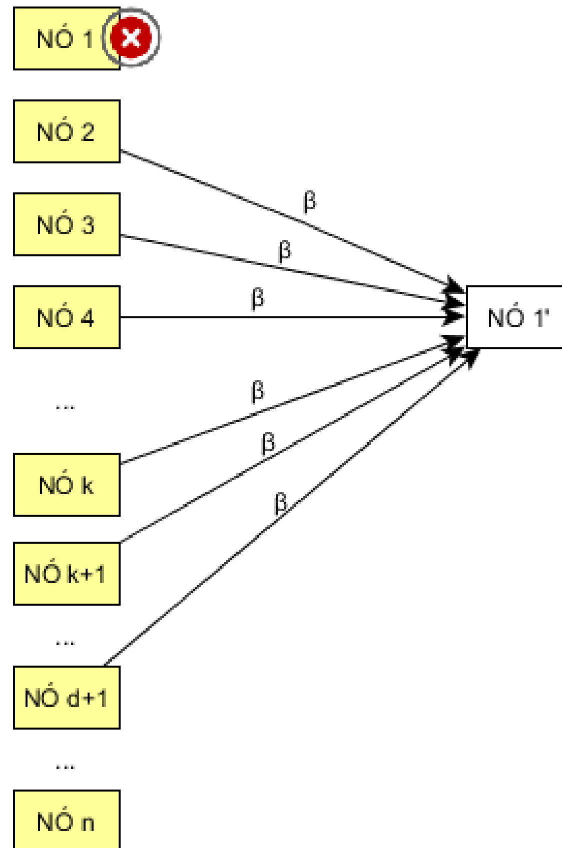


Figura 2.9: Exemplo de funcionamento de um código Regenerador

2.6.4 Códigos Localmente Reparáveis (Hierárquicos)

No contexto de *erasure coding*, códigos ditos Localmente Reparáveis são aqueles que buscam fornecer eficiência ótima para acesso aos discos. A ideia desses códigos é minimizar o número de nós de armazenamento envolvidos nas situações de reparo dos dados (PAPAILIOPOULOS; DIMAKIS, 2014). O exemplo mais comum desse tipo de código são os códigos Hierárquicos.

Códigos localmente reparáveis, inclusive os códigos hierárquicos, garantem que seja possível obter um valor de nós acessados d tal que $d < k$. Ou seja, é possível contatar um número menor de nós do que o mínimo que é necessário normalmente para um código MDS durante a restauração dos dados. Essa propriedade dos códigos

MDS permite, no entanto, que qualquer conjunto de k nós seja suficiente para restaurar um nó danificado, e tal propriedade não é obtida em códigos Localmente Reparáveis. Isso significa que nem todos os grupos de falhas possíveis podem ser toleradas (PAPAILIOPOULOS; DIMAKIS, 2014) (DUMINUCO; BIERSACK, 2008).

Particularmente em relação aos códigos hierárquicos, como o próprio nome sugere, tais códigos são construídos de uma maneira organizada em hierarquias. Na Figura 2.10 é ilustrado um exemplo de construção hierárquica de códigos. Na primeira parte da imagem é possível observar uma instância de códigos hierárquicos (2,1), que produz dois nós de armazenamento com blocos de dados codificados e um terceiro que é utilizado como paridade. Dados F1 e F2 como blocos de dados originais a serem armazenados, são criados blocos codificados B1, B2, e B3, em que somente B3, de grau 2, é o bloco de paridade. Qualquer combinação de dois dentre B1, B2, e B3 possui arestas que indicam os blocos F1 e F2, e sugere que quaisquer dois dentre eles podem ser utilizados para reparar os blocos de dados originais (LI; LI, 2013). Novamente, assim como nos demais códigos, as técnicas envolvidas tanto para codificação quanto decodificação envolvem processos algébricos de aritmética de campos finitos, e em alguns casos o uso de cálculos de ou-exclusivo (XOR). Na continuação da imagem é apresentado um código hierárquico (4,3), que nada mais é que uma extensão do código (2,1). É importante notar que mesmo com o aumento no número de nós, o número necessário em caso de necessidade de restauração dos dados originais permanece 2. Como era de se esperar, estes códigos também implicam em armazenar um volume maior de informações em cada nó, quando comparados por exemplo com códigos MDS. Este tipo de códigos foi concebido inicialmente para sistemas de armazenamento P2P (DUMINUCO; BIERSACK, 2008), porém isto não é uma norma, e sim uma decisão de projeto.

2.7 Tecnologias de disco

Nesta seção são descritas as duas principais tecnologias de disco e dispositivos existentes, os quais funcionam como base para qualquer sistema de armazenamento, distribuído ou não. O objetivo dessa seção é comparar ambas as tecnologias no sentido de justificar suas aplicações em diferentes situações.

2.7.1 Discos magnéticos e Discos de estado sólido

A unidade de disco rígido (HDD) é um dispositivo de armazenamento utilizado para armazenar e recuperar dados digitais por meio da rotação rápida de discos revestidos com material magnético. Um HDD não é volátil, ou seja, mantém seus dados mesmo quando na ausência de energia. Os dados armazenados podem ser lidos

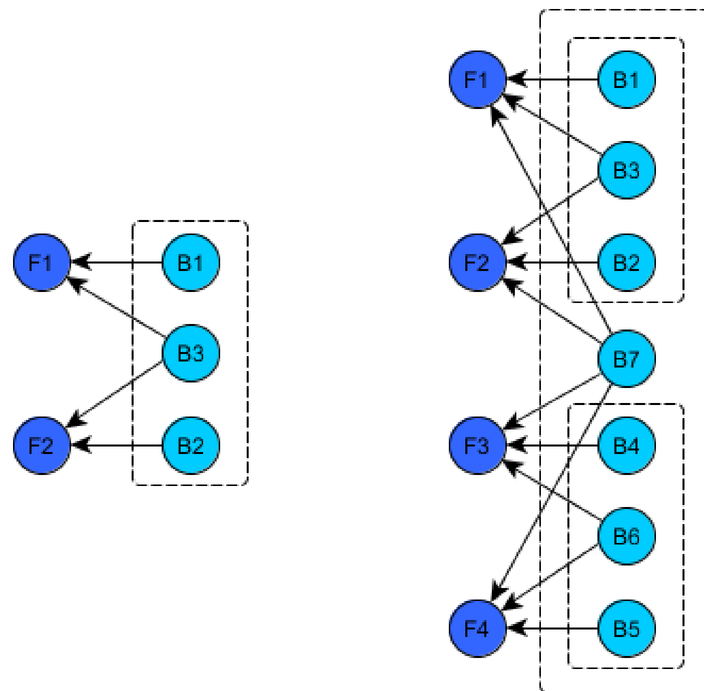


Figura 2.10: À esquerda código hierárquico (2,1) e, à direita, código hierárquico (4,3)

de um modo de acesso aleatório, o que significa que os blocos de dados podem ser armazenados ou recuperados em qualquer ordem. Um disco rígido contém um ou vários discos rotativos, rigidamente fixados, com cabeças magnéticas dispostas sobre um braço atuador que pode mudar de posição para ler e gravar dados nas superfícies metálicas (KANG et al., 2013).

A unidade de estado sólido (SSD), também conhecida como um disco de estado sólido, é um dispositivo para armazenamento de dados por meio de conjuntos de circuitos integrados como memória para armazenar dados de forma consideravelmente eficiente. Um SSD utiliza componentes eletrônicos que obedecem os padrões de entrada/saída, E/S, convencionais dos HDDs, o que permite assim ser um substituto mais fácil em aplicações comuns. SSDs utilizam a memória de armazenamento *flash*, que tem a capacidade de reter dados sem energia, das mesma forma que discos rígidos convencionais (KANG et al., 2013).

Embora os SSDs possuam diversos benefícios, como serem mais duráveis, mais rápidos e mais silenciosos (SAXENA; KUMAR, 2014) (LEE et al., 2011), há um custo ainda financeiramente alto para implantar esta tecnologia em larga escala e sua capacidade de armazenamento ainda não chega à capacidade dos HDDs. Há alguns trabalhos que sugerem que para sistemas que dependem de um número maior de dispositivos de armazenamento ainda faz mais sentido usar discos rígidos convencionais (RIZVI;

CHUNG, 2010), muito embora isso seja uma decisão de projeto que deve ser baseada nos recursos disponíveis para implantação do ambiente de armazenamento.

2.8 Tecnologias de processamento

Nas seções a seguir são descritas duas das principais tecnologias de processamento de dados existentes para computadores atualmente, bem como o que deve ser levado em consideração para escolha do uso de uma em detrimento da outra. Isso porquê ambas possuem seus pontos positivos e negativos.

2.8.1 Processadores convencionais e Processamento gráfico (GPGPU)

O paradigma de programação de uso geral, GPGPU, do inglês *General purpose GPU Computing* é claramente um expoente em termos de processamento de dados em pesquisas recentes. Isso porque esses dispositivos, que normalmente são utilizados para processamento gráfico, possuem centenas ou milhares de núcleos de processamento, principalmente em dispositivos mais recentes. Comparativamente às unidades de processamento convencionais, as CPUs, para alguns casos o ganho de desempenho chega a ser de 1000x com a utilização de uma GPU (GREGG; HAZELWOOD, 2011) (ROSENBAND; ROSENBAND, 2009). É claro que isso varia de aplicação para aplicação, e não há uma regra geral. Por exemplo, para compressão de vídeos, alguns trabalhos recentes apresentam bons resultados com o uso de GPU (KATSIGIANNIS; DIMITSAS; MAROULIS, 2015).
Em termos de armazenamento de dados não há muitos trabalhos recentes que utilizem GPU (AL-KISWANY et al., 2008) (ZHAO et al., 2016) (SOBE, 2012), porém os mesmos apresentam alguns resultados interessantes. No entanto, há algumas outras argumentações que levam em consideração limitações das interfaces PCI - *Interconector de Componentes Periféricos*, do inglês *Peripheral Component Interconnect*, de tal forma a mostrar que quando o problema a ser tratado leva em consideração uma grande quantidade de dados e uma quantidade não tão significativa de processamento, nem sempre uma GPU é a melhor opção (GREGG; HAZELWOOD, 2011). Da mesma forma que para as tecnologias de disco, a escolha entre CPU é GPU é dependente do projeto, e não há um consenso geral para o uso exclusivo de uma das duas tecnologias.

2.9 Algoritmos de *Caching*

Técnicas de *caching* aplicadas a armazenamento de dados permitem uma melhora na disponibilidade do que está armazenado, por meio da criação de cópias locais em dispositivos de armazenamento próximos. Com a estratégia do redirecionamento

das requisições para suas cópias mais próximas é possível reduzir o tempo de resposta e mesmo o consumo da banda de rede. Por esse motivo essas técnicas tem sido frequentemente utilizadas, principalmente para sistemas *peer-to-peer*, que são totalmente descentralizados. Uma das aplicações mais comuns é em serviços de transmissão como o *streaming* de vídeos (LAKSHMI; KUMAR; VENKATACHALAM, 2015).

Dentre os algoritmos de *caching* existentes, é bastante comum o uso do LRU, do inglês *Least Recently Used*, possivelmente uma das técnicas de *caching* mais fáceis de implementar, que cria uma lista dos itens solicitados e remove progressivamente da lista aqueles dados que não foram requisitados dentro de um certo período de tempo. Os dados da lista geralmente costumam ficar salvos em uma mídia de acesso mais rápido, como a memória principal. Outra técnica também bastante comum é a LFU, do inglês *Least Frequently Used*, que ao longo do tempo substitui o que está armazenado na área de acesso mais rápido com base na frequência em que os dados são acessados (LI et al., 2014). Existe, no entanto, uma terceira técnica cujo desempenho é superior a esses dois algoritmos, porém de implementação significativamente mais complexa, denominada de ARC - *Adaptive replacement cache*.

A política ARC usa o histórico do conteúdo recentemente removido do *cache* para mudar de forma dinâmica suas características de recência ou frequência. Este algoritmo é, portanto, uma combinação das estratégias LRU e LFU. Em mais detalhes, a política ARC divide o *cache* em duas partes, T_1 e T_2. T_1 armazena os dados que só foram acessados pela primeira vez, e T_2 armazena em *cache* os dados que foram acessados muitas vezes. Assim, T_1 representa os dados recentemente acessados e T_2 os frequentemente acessados. Além disso, também são mantidas outras duas listas, B_1 e B_2, que servem apenas para armazenar os meta-dados referentes às remoções mais recentes feitas em T_1 e T_2 , respectivamente. As características do *cache* podem então ser ajustadas com base nos históricos obtidos nas listas B_i para modificar os parâmetros de remoção das listas T_i dinamicamente. Dessa forma é possível detectar padrões de acesso ou mesmo de transferência de dados na rede que possam ser revertidos em melhora na política de *cache* geral (RAIGOZA; SUN, 2014). Trabalhos anteriores da literatura demonstraram as melhoras na taxa de acerto do algoritmo ARC em comparação com o LRU (MEGIDDO; MODHA, 2004), conforme é possível observar na Fig. 2.11, na qual a taxa de acerto do ARC fica próxima ao LRU apenas quando a área de *cache* é definida com tamanhos maiores. O gráfico mostra que com um número menor de páginas a taxa de acerto no ARC é sempre superior, então ele permite aos projetistas variar mais a quantidade e o tamanho das páginas do *cache* para se adaptar a cada ambiente de armazenamento.

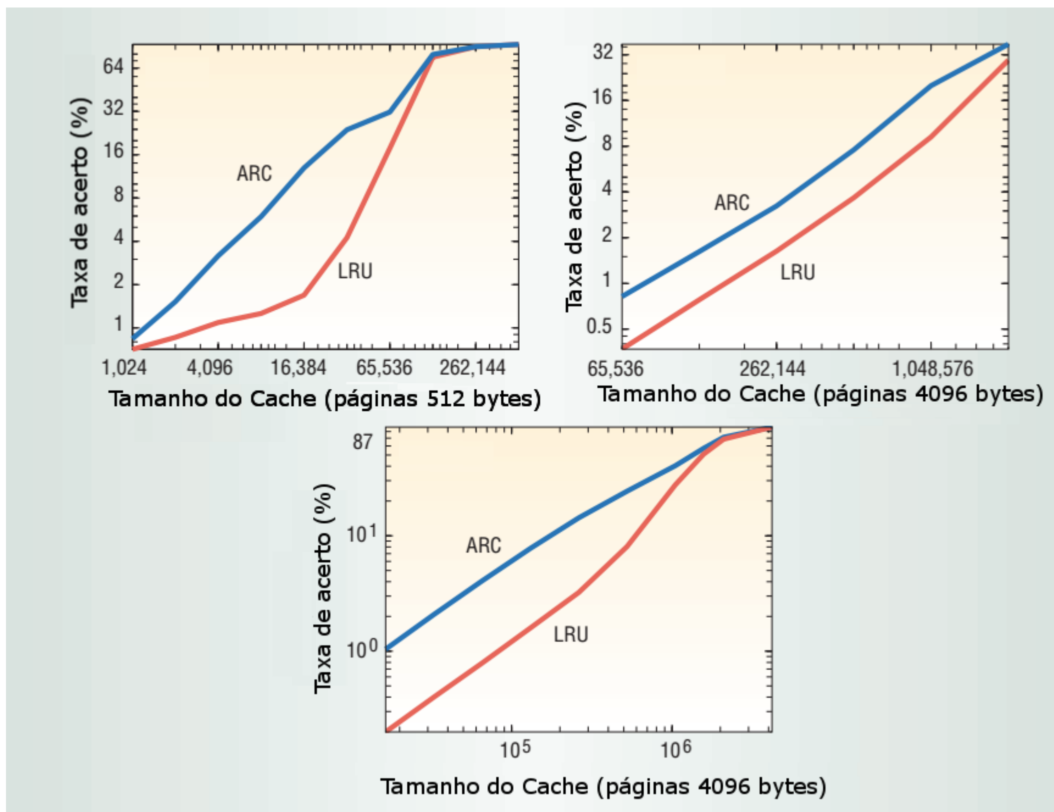


Figura 2.11: Comparação da taxa de acerto do ARC versus o LRU (adaptado do trabalho (MEGIDDO; MODHA, 2004)

2.10 Trabalhos Correlatos

Nesta seção, são apresentados alguns dos principais trabalhos encontrados na literatura que utilizam o método de replicação, o método de códigos de correção de erros, ou ambos, para propor arquiteturas de armazenamento distribuído de dados.

2.10.1 CAROM - Cache A Replica On Modification

O trabalho que propõe a arquitetura CAROM (MA et al., 2013) foi o primeiro da literatura a sugerir o uso combinado de replicação e de códigos de correção de erros. O contexto do trabalho foca em ambientes em Nuvem, e os testes foram realizados a nível de *datacenter*. Há uma estratégia de *caching* implementada, a qual utiliza o algoritmo LRU. Essa estratégia de *caching* é feita também a nível geral, por *datacenter*, e utiliza a memória RAM como forma de acesso mais rápido. A arquitetura utilizada é centralizada, assim como a forma que o *cache* fica disponível.

2.10.2 MICS - Mingling Chained Storage

O trabalho que propõe a arquitetura MICS (TANG et al., 2015) é mais recente, e se baseia em parte no trabalho anterior da arquitetura CAROM, porém com algumas diferenças notáveis. Utiliza um modelo de gerenciamento com múltiplos mestres e propõe o armazenamento na forma de objetos, além de possuir como uma das principais contribuições a criação de uma função de UPDATE para os objetos armazenados. Normalmente essa função depende de remover o objeto e recriá-lo novamente, pois não existe atualização direta. O armazenamento dos dados é realizado inteiramente em discos rígidos magnéticos, e a codificação dos dados é feita utilizando processadores convencionais.

2.10.3 HRSPC - Hybrid Redundancy Scheme Plus Computing

O trabalho que propõe a arquitetura HRSPC (LI et al., 2016) é mais focado em melhorar diretamente alguns as aspectos de códigos de correção de erros de modo a mesclá-los num algoritmo misto ao invés de usar as duas técnicas separadamente como outros trabalhos. No entanto, isso torna o trabalho muito mais teórico do que aplicado. Não são dados detalhes específicos da arquitetura e, embora obtenha bons resultados assim como outros trabalhos, não utiliza uma arquitetura *peer-to-peer* explicitamente, mas sugere alguns conceitos nesse sentido. Também utiliza discos rígidos magnéticos e processadores convencionais para armazenamento e codificação dos dados, respectivamente.

2.10.4 Robot - Big data storage system based on erasure coding

O trabalho que propõe a arquitetura Robot (YIN et al., 2013) é focado apenas no uso de códigos de correção de erros para armazenamento de dados, e ignora o uso de replicação, o que segundo estudos recentes pode ser um erro (GRIBAUDO; IACONO; MANINI, 2016). Contudo ainda apresenta bons resultados e propões uma mescla de arquiteturas, pois em uma visão geral há claramente a figura dos computadores mestres, que são aqueles responsáveis por codificar e decodificar os dados armazenados, além de controlarem os metadados. Contudo numa segundo observação há também uma rede *peer-to-peer* em anel de computadores que funcionam exclusivamente para armazenar dados e não realizam nenhum processamento adicional.

2.10.5 HDFS-Xorbas - a module for erasure code in HDFS

O trabalho que propõe a arquitetura HDFS-Xorbas (SATHIAMOORTHY et al., 2013) é baseado no sistema de arquivos distribuído HDFS - *Hadoop Distributed File*

System (BORTHAKUR, 2008). Por esse motivo, a arquitetura é semelhante a desse sistema de arquivos, que é do tipo mestre-escravo. A principal contribuição do trabalho é fornecer um esquema de códigos de correção de erros para o HDFS, que a princípio usa apenas replicação em três vias. Com isso, propõem um novo tipo de código e o implementam de forma integrada a essa tecnologia previamente existente, com bons resultados, porém ainda força o uso de códigos ou replicação, e não ambos em conjunto.

2.10.6 *Análise dos trabalhos correlatos*

Uma representação comparativa entre os principais trabalhos encontrados na literatura em armazenamento de dados foi realizada por meio do destaque dos principais aspectos arquiteturais de cada um deles. Na Tabela 2.1 é exposta essa representação comparativa. É válido destacar, com relação aos seis aspectos arquiteturais analisados em cada trabalho, a ausência de qualquer mecanismo de *cache* na maioria deles. Ao mesmo tempo, há um padrão seguido por todos em termos de metodologias e tecnologias de estruturação, armazenamento e processamento dos dados. Adicionalmente, todos os trabalhos se mostram dependentes de alguma forma a centralizações em suas arquiteturas, como esperado em *designs* do tipo mestre-escravo.

Tabela 2.1: Comparação entre os trabalhos correlatos

Nome do trabalho / Aspectos arquiteturais	CAROM	MICS	HRSPC	Robot	HDFS-Xorbas
Redundância	R/EC	R/EC	R/EC	EC	EC
Estruturação	OS	OS	OS	OS	OS
Design	ME	ME	ME/P2P	ME/P2P	ME
Armazenamento	HDD	HDD	HDD	HDD	HDD
Processamento	CPU	CPU	CPU	CPU	CPU
Cache	LRU	-	-	-	-

R = replicação; EC = erasure coding; OS = object storage; ME = mestre-escravo.

2.11 Considerações Parciais

Neste capítulo foram apresentados os conceitos mais importantes que envolvem a área de armazenamento de dados e, ao final, foi feita uma apresentação dos principais trabalhos do estado-da-arte. O foco dos trabalhos estudados foi a tolerância a falhas e a garantia de redundância dos dados armazenados. Para cada tecnologia ou metodologia apresentada foram descritas as suas principais características, em que também se

exibem as vantagens e desvantagens que devem ser levadas em consideração de acordo com os compromissos que cada projeto deseja alcançar. No encerramento deste capítulo foi possível apresentar uma análise comparativa entre os trabalhos correlatos, com destaque para seis dos principais aspectos de interesse para uma arquitetura de armazenamento distribuído.

CAPÍTULO 3 – Armazenamento P2P com tolerância a falhas híbrida e sistema de cache

3.1 Considerações iniciais

Neste capítulo está descrita a arquitetura proposta, que recebe o nome de Griddler. O nome advém de um quebra-cabeças matemático de reconstrução de imagens, em que algumas sequências de números são utilizadas como base para redefinir desenhos simples a partir de um espaço inicialmente vazio. A ideia desse quebra-cabeça, e o nome utilizado, lembra que esta é uma arquitetura como foco principalmente em tolerância a falhas, ou seja: recuperar informações a partir de dados redundantes. A arquitetura proposta difere das demais, principalmente por ser *peer-to-peer* e utilizar um mecanismo diferenciado de *cache* distribuído. A implementação foi realizada inteiramente nas linguagens C e C++ e conta hoje com mais de 3000 linhas de código.

3.2 Descrição e requisitos do ambiente distribuído

Inicialmente supõe-se para a arquitetura proposta um ambiente com quaisquer *n* computadores em rede. É necessário que cada um destes computadores possua um endereço IP único e consiga se comunicar com os demais, muito embora num primeiro momento não tenham qualquer ligação lógica e atuem de forma independente. Não há restrições quanto a *hardware* especializado necessário em cada computador para o funcionamento do software da arquitetura Griddler, mas se recomenda o uso de processadores recentes quando possível, o que será explicado em seções posteriores desse capítulo. Dado esse ambiente inicial, é preciso inicializar então os mecanismos que possibilitam que os *n* computadores se configurem em nós de um ambiente de armazenamento integrado.

A primeira tarefa a ser realizada é utilizar alguma estratégia para gerar um GUID - *Globally Unique Identifier*, identificador global único, para cada um dos nós da rede. Isso é necessário pois permite aos clientes de *software* acessarem os dados armazenados ao buscá-los exatamente em cada um dos nós em que estiverem armazenados, sem possibilidade de conflitos. No caso da Griddler, o algoritmo utiliza a cifra SHA-1 (EASTLAKE; JONES, 2012), uma técnica de *hashing* que é relativamente simples, possui diversas implementações, e produz um valor de dispersão de 160 bits com 40 caracteres, o que é um valor razoável pois permite que coexistam 2^{160} nós com GUID diferentes na rede simultaneamente. Além disto, o algoritmo SHA-1 possui tempo de execução reduzido comparativamente a outros algoritmos, conforme é possível observar na Figura 3.1. No caso da arquitetura proposta esse valor é gerado com

base no endereço IP de cada nó, que por definição também é único a cada computador.

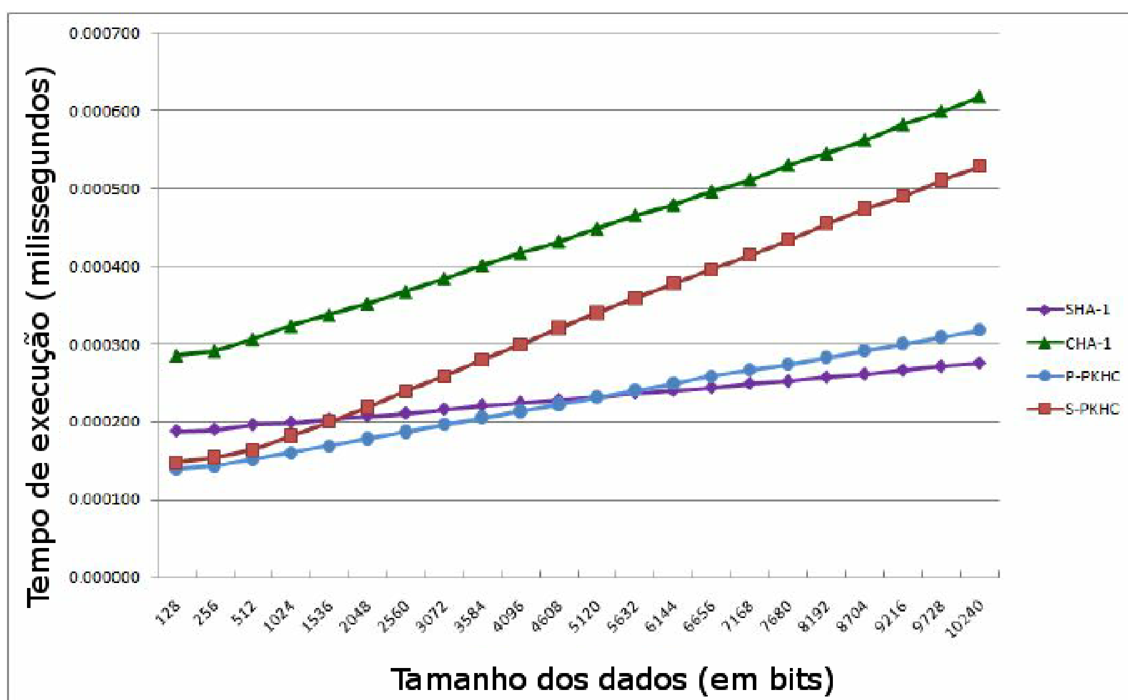
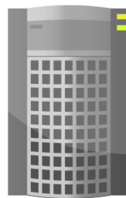


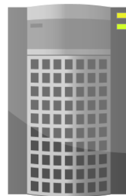
Figura 3.1: Comparação do tempo de execução da cifgra SHA-1 e outros algoritmos, adaptado de (MAQABLEH, 2011).

Mesmo com identificadores, os nós inicialmente não tem ligação nenhuma uns com os outros. O primeiro fator considerado na arquitetura Griddler segue um padrão de referência para qualquer sistema P2P, e trata da existência de mecanismos que permitam organizar uma rede de computadores que inicialmente não tem essa ligação lógica. Por ser um ambiente totalmente descentralizado, não existe a figura de um controlador central, de modo que o gerenciamento é distribuído entre todos os membros da rede. A Griddler implementa uma estrutura baseada em um dos protocolos existentes para gerenciamento de ambientes P2P, o protocolo Chord (STOICA et al., 2003), que apesar de ter sido proposto há algum tempo ainda é muito utilizado em trabalhos recentes (LI; GUO; FRANZINELLI, 2015) (JEDDA; MOUFTAH, 2015). Para facilitar o gerenciamento e as buscas, esse protocolo realiza suas operações por meio de uma estrutura chamada de tabela de *hashing* distribuído, do inglês *distributed hashing table*, DHT, que será detalhada nas próximas seções.

Por definição do protocolo Chord, após devidamente identificados, os nós devem num segundo momento ser abstraídos logicamente para uma estrutura em rede do tipo anel, ou seja, tratados como se estivessem em um círculo e ordenados pelo seu GUID. Como os valores gerados pela função de *hashing*, a princípio, não podem ser comparados

diretamente para determinar qual é maior, foi necessário implementar uma função de conversão para um valor decimal (mod 2^{160}) de modo que o anel possa ser gerado. Essa é uma etapa preliminar essencial para a arquitetura proposta, e está representada na Figura 3.2. Para conversão de uma cifra para um valor inteiro, é feito um cálculo baseado nos números inteiros existentes na sequência SHA-1 e por meio da atribuição de valores inteiros aos demais caracteres. De forma simplificada, após ter efetivamente construído essa ligação lógica, a rede tem a aparência semelhante a da Figura 3.3. A ligação é dita lógica pois não existe fisicamente, mas é uma topologia definida em *software* com base nos GUID de cada nó, cuja ordenação influencia posteriormente em todas as operações do sistema distribuído.





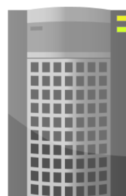


Figura 3.2: Geração de GUID para cada um dos nós da rede

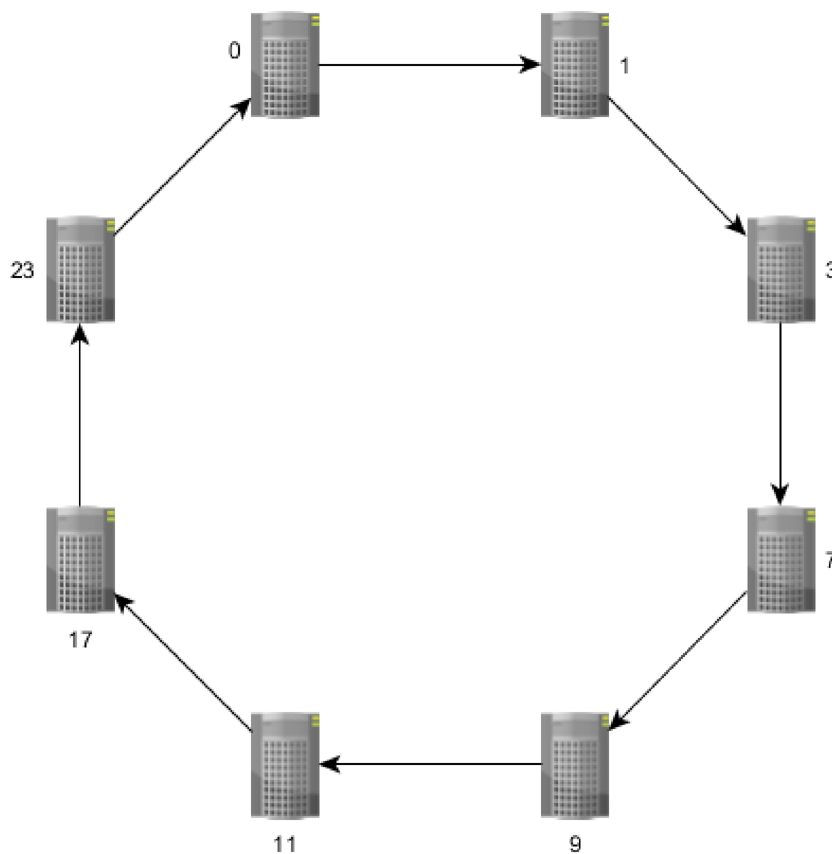


Figura 3.3: Geração da rede *peer-to-peer* em anel

Adicionalmente, foi desenvolvida uma representação dos componentes em alto nível do ambiente proposto. Este modelo é ilustrado na Figura 3.4, e retrata os módulos existentes em cada um dos nós de uma rede na arquitetura desenvolvida. É possível observar que o usuário consegue interagir com o sistema de armazenamento através de um cliente de linha de comando simples ou através do protocolo HTTP em um navegador *web* comum. O módulo de *hashing* é usado tanto na leitura quanto na escrita de dados, e por isso a área de *cache* e o armazenamento local são ligados a ele. Adicionalmente, existem módulos auxiliares que são utilizados para monitoramento do status dos nós na rede, para codificação dos dados antes de serem armazenados e para gerir a comunicação com os demais nós da rede. Cada *peer* da rede Griddler possui essas mesmas funções e interage com os demais na rede em anel conforme o necessário para solicitar ou inserir dados e seus metadados. A comunicação entre os nós é totalmente realizada via protocolo HTTP, por meio do mesmo servidor *web* disponível a usuários externos.

Nas seções seguintes são apresentadas as operações mais comuns e importantes, que são: inserção, o equivalente a função PUT, busca, o equivalente a função

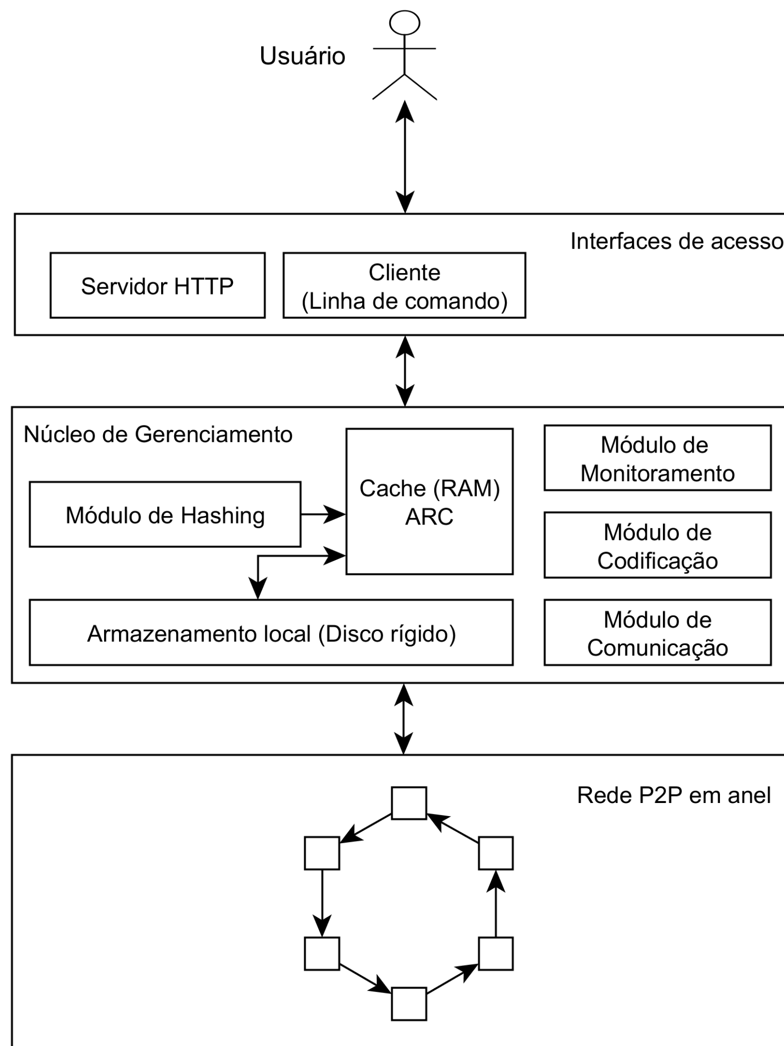


Figura 3.4: Modelo em alto nível de um dos nós da rede Griddler

GET, remoção, o equivalente a função DELETE, e atualização, o equivalente à função UPDATE na arquitetura proposta. Também será detalhado o funcionamento básico de inserção e remoção de nós na Griddler, em particular em termos de resposta a falhas.

3.3 Tabela de roteamento

Essa estrutura tem função central na Griddler, bem como no protocolo Chord para gerenciamento dos dados em ambiente *peer-to-peer*. Trata-se de uma tabela extra que é armazenada em memória RAM. O tamanho dessa tabela é uma decisão de projeto, e depende da implementação de cada arquitetura. Para a Griddler é utilizado o valor máximo de 9, mas esse tamanho pode ser customizado se for necessário. Ele indica uma certa quantidade de rotas, ou endereços IP e GUIDs de nós de armazena-

mento, que ficam registrados em cada nó *n*. O primeiro item dessa tabela é sempre o endereço do nó imediatamente na sequência do nó *n* na rede em anel. Para cada item *i* adicional, a tabela armazena o endereço do nó $(n+2^{i-1}(mod 2^{160}))$ na sequência do nó *n*. A operação em módulo apenas indica que os cálculos devem ser realizadas para números no formato gerado pelo algoritmo SHA-1. Ou seja, para 4 entradas, a tabela do nó 0 ficaria conforme é apresentado na Tabela 3.1.

Tabela 3.1: Tabela de roteamento do nó 0 do ambiente P2P

Finger table	
1	Endereço IP do nó 1 $(0+2^{1-1}(mod 2^{160}))$
2	Endereço IP do nó 2 $(0+2^{2-1}(mod 2^{160}))$
3	Endereço IP do nó 4 $(0+2^{3-1}(mod 2^{160}))$
4	Endereço IP do nó 8 $(0+2^{4-1}(mod 2^{160}))$

Ter esse conhecimento localmente é importante pois quando um nó recebe uma requisição de busca com base em um GUID de objeto, ainda que o próprio nó não tenha esse objeto salvo, conseguirá determinar qual nó da sua tabela de roteamento contém ou está mais próximo do local em que o objeto está de fato armazenado. Isto porque tanto os identificadores dos nós quanto dos próprios objetos armazenados são ambos gerados a partir da mesma cifra, e o protocolo Chord busca distribuir os objetos armazenados de forma a aproximar nós e objetos com identificadores próximos. Por exemplo, caso existam um nó com identificador 1 e outro com identificador 8 na rede, um objeto com identificador 7 tem maior chance de ser salvo no nó 8. Com a utilização da tabela de roteamento a busca se torna mais eficiente, visto que foi determinado que, com alta probabilidade, o número máximo de buscas por meio dessa estratégia em um rede com *n* nós se encontra na ordem de O(log*n*) (STOICA et al., 2003). Essa mesma estratégia pode ser utilizada na inserção de dados para chegar ao nó de armazenamento adequado para um determinado GUID de objeto. Na Figura 3.5 é possível observar como ficam definidas as ligações pela tabela de rotas, de modo que o nó 0 conhece as rotas para os nós 1, 2, 4 e 8. Dessa forma, uma requisição ao nó 0 de um objeto que se encontra em um desses outros nós será facilmente redirecionada. Caso nenhum desses nós armazene o objeto solicitado, ainda assim pelo menos um deles saberá qual nó armazena o objeto ou qual outro nó estaria mais próximo dos dados requisitados. No melhor dos casos o nó que recebeu a requisição inicial será coincidentemente o responsável por armazenar o objeto solicitado e a resposta será imediata, mas no pior caso deverá passar por todos os nós da rota até que seja recebido algum retorno para a requisição.

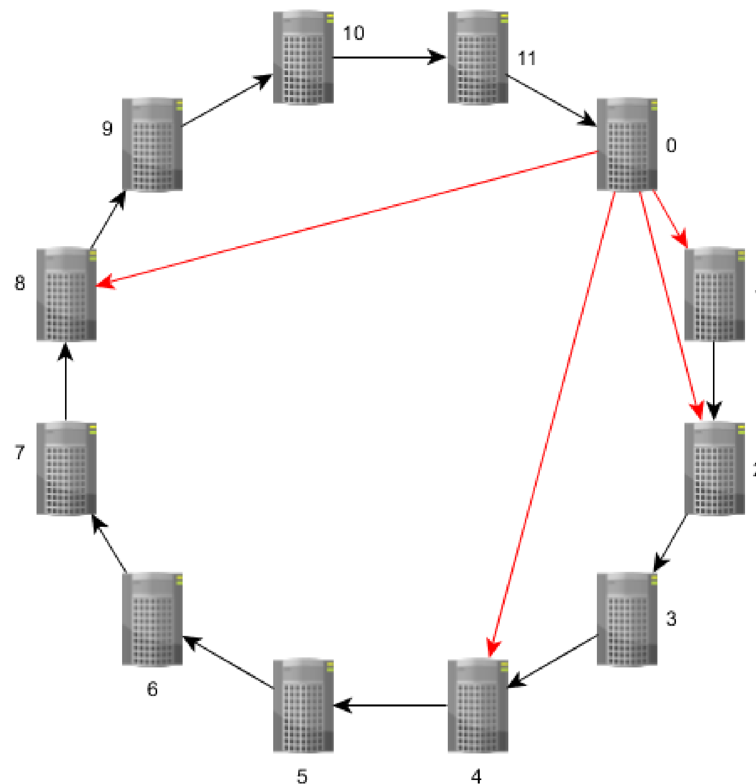


Figura 3.5: Rede com indicativos da tabela de roteamento do nó 0

3.4 Inserção de um novo nó

Nesta seção são apresentados os métodos de inserção de um nó na rede em cada um dos casos possíveis. Neste momento ainda se trata mais de organização do *hardware*, de modo que as operações a seguir não influenciam nos dados previamente armazenados no sistema.

3.4.1 Inserção quando não existem outros nós na rede

Quando não existem outros nós na rede Griddler, significa que o sistema precisa ser inicializado. A inserção de um nó depende da execução em plano de fundo do *software* implementado no nó que se deseja inserir na rede em anel. Para todos os efeitos, os exemplos a seguir consideram inicializações feitas para nós da rede baseados no sistema operacional Linux. Na Figura 3.6 segue o exemplo de uma inicialização do *software* cliente no primeiro nó da rede. Os parâmetros da execução são, em ordem, o endereço IP da interface de rede a ser associada com o serviço, a porta a ser associada com o serviço, um diretório local para ser utilizado para responder a requisições HTTP e o algoritmo de *cache* a ser utilizado. Esse último existe pois a

Griddler possui implementados tanto o algoritmo de *caching* adaptável, ARC, quanto o algoritmo LRU, para fins de comparação.

```
root$ ./griddler 192.168.56.56 8000 $(pwd)/.webserver arc
--------------------------------------------------
Servico esta executando em http://192.168.56.56:8000
--------------------------------------------------
+--------------------------------------------------------+
| Bem-vindo!                                             |
|                                                        |
| Lista de operacoes:               \__    __/           |
|                                   /_/ /\ \_\           |
| 0) STATUS                        __ \ \/ / __          |
| 1) PUT                           \_\_\/\/_/_/          |
| 2) GET                       __/\___\_\/_/___/\__      |
| 3) DELETE                      \/ __/_/\_\__ \/        |
| 4) SAIR                          /_/ /\/\ \_\          |
|                                   __/ /\ \__           |
|                                   \_\ \/ /_/           |
|                                   /        \           |
|                                    GRIDDLER            |
|                                       1.0              |
+--------------------------------------------------------+
---> 0
#######################################################

GRIDDLER em 192.168.56.56:8000
<NO: 478, PREDECESSOR: 478, SUCESSOR: 478>
Fingers Table: [478, 478, 478, 478, 478, 478, 478, 478, 478]

#######################################################
```

Figura 3.6: Primeiro nó de uma rede na arquitetura Griddler

É interessante notar que cada nó armazena informações sobre seu predecessor e sucessor, que numa rede com um único nó inicialmente correspondem a esse mesmo nó. O mesmo é válido para a tabela de roteamento, que apresenta repetições num primeiro momento pois não existem nós suficientes na rede para preencher seus índices adequadamente. O diretório passado por parâmetro para responder requisições HTTP é utilizado pois não é necessário interagir com a Griddler apenas por meio desse cliente simplificado. Foi implementado um servidor *web* integrado que permite que aplicações se conectem por meio do protocolo HTTP e que usuários interajam com a arquitetura em um navegador *web* comum.

3.4.2 Inserção quando existem outros nós na rede

Neste caso a operação de inserção acontece quando um nó é inserido na rede, mas existem outros nós conectados em anel. Inicialmente os demais *peers* não tem como tomar conhecimento da existência desse novo nó, de modo que cabe a esse novo nó explicitamente solicitar a entrada na rede P2P. Essa solicitação pode ser feita a qualquer um dos *peers* da rede.

O *peer* que recebeu a solicitação auxilia em seguida na criação de um GUID a ser atribuído ao novo nó, com base no endereço IP do mesmo. Também com base no GUID, o *peer* da rede que recebeu a solicitação envia um aviso ao *peer* mais adequado para preceder o novo nó, de modo a manter a rede ordenada com base nos identificadores inteiros. O *peer* que agora precederá o novo nó compartilha uma cópia de sua própria tabela de roteamento local, que servirá para que o novo nó crie uma nova tabela de roteamento. Por fim, os *peers* da rede recalculam suas rotas conforme o necessário.

Dessa forma a rede é reconfigurada. Espera-se que o novo nó entre na rede com uma quantidade significativa de espaço de armazenamento livre. Por esse motivo, foi implementado um mecanismo de migração automática de dados, transferindo para o novo nó alguns objetos que estavam armazenados em outros pontos da rede. Na Figura 3.7 é possível perceber de forma mais visual essas etapas, as quais estão devidamente implementadas no ambiente desenvolvido.

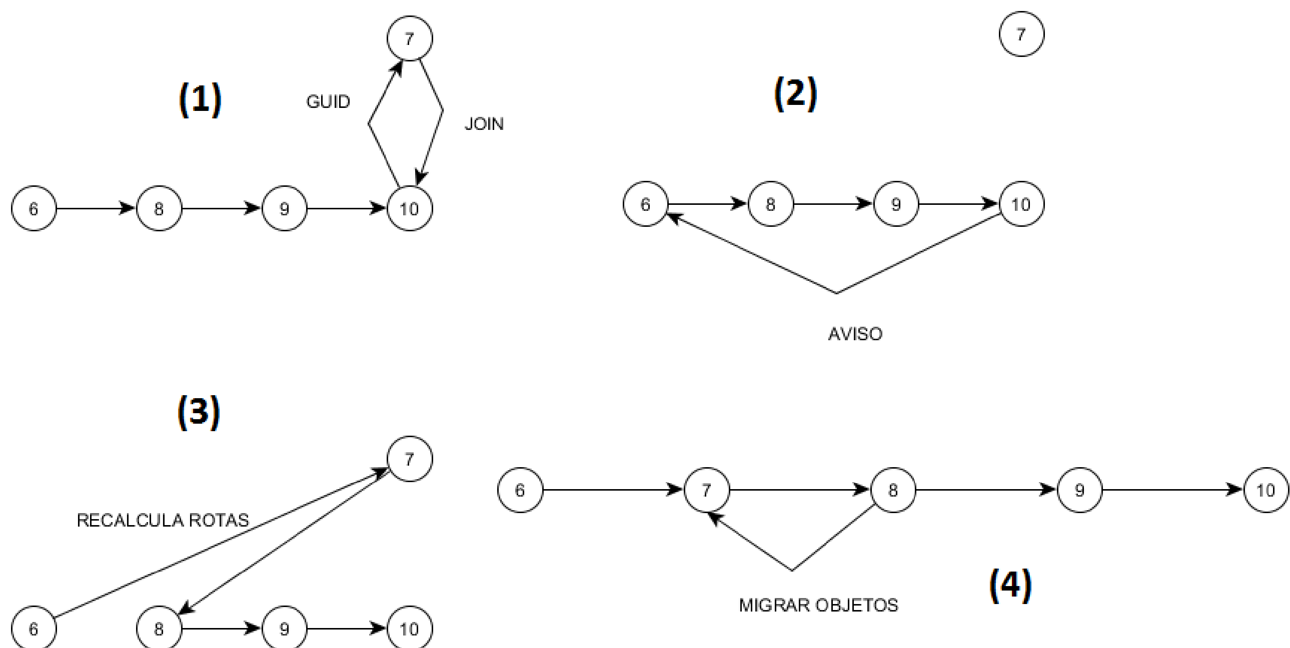


Figura 3.7: Representação das etapas a serem realizadas para inserção de um novo nó

Em contraste à inicialização com apenas um nó, o *software* cliente neste caso deve ser inicializado com um parâmetro a mais, o parâmetro *join*. Em tempo de compilação é definida uma variável condicional para uma referência ao *backbone*, que é o nó utilizado como ponte para a rede principal da Griddler, e quando executado com o parâmetro *join* o novo nó buscará na rede esse nó referência da rede principal e solicitará sua inclusão. Em comparação com o exemplo anterior, em que foi adicionado um único nó, o próximo nó da rede tem configurações semelhantes ao exemplo da Figura 3.8.

```
root$ ./griddler 192.168.56.2 8000 $(pwd)/.webserver arc --join

...

...

...

######################################################

GRIDDLER em 192.168.56.2:8000
<NO: 178, PREDECESSOR: 478, SUCESSOR: 478>
Fingers Table: [478, 478, 478, 478, 478, 478, 478, 478, 478]


######################################################
```

Figura 3.8: Segundo nó de uma rede na arquitetura Griddler

3.5 Remoção de um nó

Nesta seção são apresentados os métodos de remoção de um nó na rede em cada um dos casos posssíveis. Neste momento ainda se trata mais de organização do *hardware*, de modo que as operações a seguir não influenciam nos dados anteriormente armazenados no sistema.

3.5.1 Remoção prevista pelos usuários

É possível que em algum momento seja necessário remover um nó da rede de forma planejada. Por exemplo, para fazer alguma atualização em um equipamento, ou mesmo manutenção de qualquer tipo. Nestes casos, é possível remover o nó da rede acionando a função de saída do *software* cliente.

De forma semelhante ao processo de entrada de um novo nó, o que acontece nestes casos de encerramento, por meio dessa função, é que o nó que está de saída redistribui todos os objetos armazenados a partir de solicitações aos demais nós da rede. Antes disso envia alertas a seu sucessor e predecessor diretos e, após redistribuir todos os objetos de dados o *software* cliente é encerrado e o vínculo lógico é desfeito. Ao mesmo tempo os antigos sucessor e predecessor do nó que deixou a rede cuidam de atualizar suas próprias tabelas de roteamento e avisam os demais nós restantes da saída que ocorreu.

3.5.2 Remoção devido a falhas e imprevistos

O processo de falha imprevista é um pouco mais complicado de tratar do que uma remoção planejada. Na Griddler é proposto o uso de uma estratégia de *probing*, ou verificação, para esse tipo de situação. Isso significa que circulam na rede diversas mensagens, enviadas constantemente e utilizadas como indicativo de estado dos nós. Cada *peer* fica responsável por monitorar seu sucessor imediato na rede em anel. Ou seja, o *peer* 6 monitoraria o *peer* 7 e o *peer* 7 monitoraria o *peer* 8, e assim sucessivamente. Dessa forma, quando qualquer nó falhar, seu predecessor imediato é que toma conhecimento da ausência do nó e fica responsável por iniciar os processos de reconstrução.
A reconstrução dos dados não precisa ser imediata, mesmo com a inserção de um novo nó na rede no lugar daquele que ficou com falha, dado que as técnicas de replicação e códigos de correção de erros por si só são suficientes para garantir o acesso aos dados mesmo diante de algumas indisponibilidades. Por esse motivo, no momento a Griddler não conta com um mecanismo automatizado de reconstrução, de modo que um nó apenas avisa os demais quando percebe a falha de outro e o que é feito é tão somente a atualização das tabelas de roteamento. Ao mesmo tempo, o que ocorre na Griddler é que na próxima operação de atualização de um determinado objeto de dados o sistema automaticamente recria as cópias perdidas, por replicação ou codificação, decorrentes de possíveis falhas dos nós. Dessa forma, os dados são eventualmente recriados e se evita sobrecargas desnecessárias de processamento e rede no sistema, principalmente quando as falhas são frequentes.

3.6 Operações básicas de interação com o sistema

Nesta seção, são apresentados os métodos de interação com os dados disponíveis na arquitetura desenvolvida. As operações implementadas e descritas a seguir representam o essencial para sistemas de armazenamento de dados: escrita, leitura, remoção e atualização das informações.

3.6.1 Inserção de dados na forma de objetos

A inserção de dados na arquitetura proposta segue os mesmos princípios do algoritmo Chord para sistemas P2P, o qual foi mencionado anteriormente, com uma alteração que é exclusiva deste projeto. A operação proposta tem a estrutura de uma tripla com o seguinte formato:

PUT(chave, valor, r)

No qual a chave é também um GUID, ou OID, gerado a partir do arquivos com base na mesma técnica de hashing, SHA-1. Ou seja, da mesma forma, é possível haver até 2^{160} objetos armazenados no sistema ao mesmo tempo. O parâmetro *valor* representa os dados que se deseja armazenar. O único diferencial acrescentado pela arquitetura Griddler que não faz parte da definição original do algoritmo Chord é o parâmetro *r*, que é um valor inteiro. O parâmetro *r* apenas define o tipo de redundância desejado, que no momento pode ser de três tipos:

- valor 0, para nenhuma redundância
- valor 1, para redundância por meio de replicação em 3 vias
- valor 2, para redundância por meio de um código MDS do tipo Liberation com parâmetros (6,2)

Na verdade ainda se pensa em trabalhar com maiores variações nesses parâmetros, mas as funções básicas atualmente são essas três. A escolha dos parâmetros do código utilizado se baseia na garantia da mesma redundância que a replicação em 3 vias, ou seja, até duas falhas simultâneas de nós de armazenamento. Os parâmetros de codificação, no entanto, podem ser alterados a qualquer momento conforme o necessário. A inserção na verdade é a operação mais simples quando não há nenhuma redundância, pois é a forma como foi prevista no algoritmo original. O que ocorre é que, após o cálculo do GUID do objeto de dados que se deseja armazenar, esse objeto é diretamente mapeado ao nó cujo GUID seja imediamente superior a esse valor. Ou seja, se convertidos em uma base decimal para facilitar a compreensão, o desenho da rede com alguns objetos inseridos se assemelha com o da Figura 3.9, em que se destaca a aproximação de nós e objetos com identificadores próximos.

A única diferença quando se acrescenta alguma outra técnica de redundância é que os dados redundantes também são salvos na forma de objetos em outros nós da rede. Por exemplo, no caso da replicação em três vias, o objeto original é salvo em um dos nós de acordo com seu GUID, e duas outras cópias são salvas em outros nós quaisquer da rede da rede em anel, de acordo com o GUID gerado para eles. É

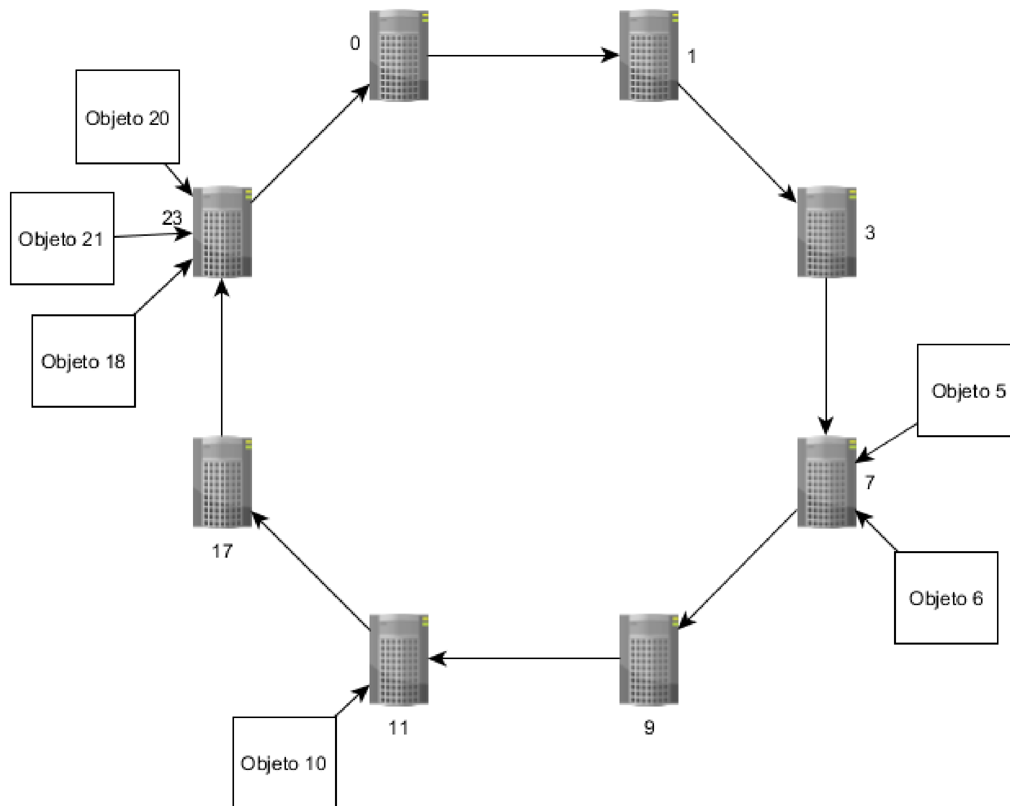


Figura 3.9: Rede com alguns objetos inseridos

importante ressaltar, no entanto, que esses dados redundantes não carregam consigo o GUID do arquivo original, independente se o tipo de redundância for por meio de replicação ou de códigos de correção de erros, visto que se tratam de chaves diferentes. Por exemplo, para replicação, supondo um objeto original com o nome *imagem1*, os objetos redundantes são *imagem1_c1*, *imagem1_c2* e *imagem1_c3*. Conforme será visto para a busca, quando um programa cliente solicita o acesso a um determinado objeto o sistema sempre é direcionado ao local no qual a primeira cópia desses dados se encontra, que é a parte mais difícil da busca. Somente se essa cópia não estiver disponível é que o acesso parte para as informações redundantes, porém é sabido que estas informações estão em um ou mais dos nós da sequência na rede em anel.

3.6.2 Busca de dados na forma de objetos

A busca dados na arquitetura proposta segue os mesmos princípios do algoritmo Chord para sistemas P2P, o qual foi mencionado anteriormente, com algumas alterações exclusivas desse projeto. A operação proposta tem o seguinte formato:

GET(chave)

A solicitação de busca, assim como a de inserção, pode partir de qualquer um dos nós. Portanto, a solução trivial é percorrer sequencialmente todos os nós do anel até encontrar aquele que armazena o objeto com o GUID buscado. Contudo, à medida em que o número de nós cresce, esse tipo de busca se torna mais e mais ineficiente. Assim, no trabalho proposto, a busca de dados utiliza uma estrutura auxiliar em cada um dos nós, denominada de *finger table*, ou *routing table*, que é uma tabela indicativa do roteamento necessário para se chegar a um determinado objeto.

3.6.3 Remoção de dados na forma de objetos

A remoção de dados na arquitetura desenvolvida busca remover sequencialmente todos as possíveis versões, redundantes ou não, do objeto que esteja armazenado no sistema distribuído. A operação implementada tem o seguinte formato:

DELETE(chave)

Como não há nenhuma forma de gerenciamento que informe se um determinado objeto está armazenado com única cópia, várias cópias ou mesmo codificado, é preciso verificar todos esses casos para removê-lo efetivamente. Dessa forma a função de remoção implementada na arquitetura desenvolvida solicita aos demais *peers* a remoção de quaisquer versões do objetos possam existir. Essas requisições se espalham pelo sistema distribuído, e ao final o usuário tem como resposta que todas as versões do objeto foram devidamente removidas.

3.6.4 Atualização de dados na forma de objetos

A atualização dos dados é uma operação semelhante à de inserção, salvo que consiste em inserir uma nova versão de um objeto previamente inserido. Na arquitetura Griddler essa operação é feita como uma combinação das demais operações. É possível remover o objeto antigo e reinseri-lo, inclusive com outro nível de redundância. Porém, a operação de inserção por padrão sobrescreve objetos existentes para uma determinada chave caso existam.

Dessa forma, dado que exista um objeto codificado no sistema, por exemplo um vídeo com a chave de identificação “vídeo1”, inserir um novo “vídeo1” por meio de codificação vai sobrescrever a versão antiga. Contudo, se o vídeo estivesse replicado, seria necessário removê-lo para somente após essa etapa inseri-lo novamente com codificação. Da mesma forma, inserir uma nova versão com replicação de um objeto anteriormente inserido com replicação irá sobrescrever a versão antiga automaticamente. É ne-

cessário remover a versão antiga somente quando se alterna o tipo de redundância desejada.

3.7 Mecanismo de *caching* distribuído com estratégia ARC

A Griddler implementa um mecanismo de *caching* distribuído por meio da utilização do algoritmo adaptável ARC, *Adaptive Replacement Cache*, em detrimento dos algoritmos mais comuns LRU e LFU. Nos estudos encontrados na literatura não houveram outros trabalhos que utilizem esse algoritmo em uma arquitetura P2P com tolerância a falhas híbrida. Contudo, alguns trabalhos sugeriram que uma estratégia de *cache* pode trazer benefícios consideráveis a um sistema de armazenamento distribuído (MA et al., 2013) (TANG et al., 2015).

Cada um dos nós do sistema distribuído mantém localmente as tabelas descritas na seção 2.9, que no caso do ARC são duas, uma para itens recentemente acessados e outra para itens frequentemente acessados. Então do ponto de vista da Griddler o que acontece é que cada nó, quando recebe uma requisição, primeiro tenta localizar o objeto em seu sistema de *cache* local ao utilizar uma dessas duas listas, e somente se não encontrar é que é utilizada a tabela de roteamento, ou *finger table* para buscar o nó que armazena o objeto desejado. É evidente que o tamanho desse *cache* deve ser controlado para evitar também uma sobrecarga de armazenamento desnecessária nos nós.

O conceito das listas serve apenas para registrar e organizar os dados em *cache* efetivamente armazenados. Do ponto de vista prático, a ideia é manter os objetos em memória RAM sempre que possível, ao implementar uma área de *cache* com tamanho configurável na memória principal, ao mesmo tempo que se utiliza um ou mais discos rígidos, idealmente com uma velocidade de trasmissão alta, dedicados ao armazenamento secundário. A área de *cache* em RAM é local a cada nó, que a atualiza com base nas nas suas próprias requisições, e é possível que nós utilizem áreas de *cache* com tamanhos e políticas diferentes entre si.

Na Figura 3.10 é possível observar o esquema de acesso para operações de GET no sistema, com indicativos do uso do *cache*. O que ocorre de diferente é que em cada solicitação de GET é feita uma verificação nas listas do algoritmo ARC e, caso o objeto solicitado esteja no cache local, o mesmo é retornado. Contudo, em último caso, o objeto é buscado na rede por meio do mesmo mecanismo de tabela de roteamento descrito nas seções anteriores. Nesses casos, as tabelas de controle do *cache* são atualizadas conforme o necessário.

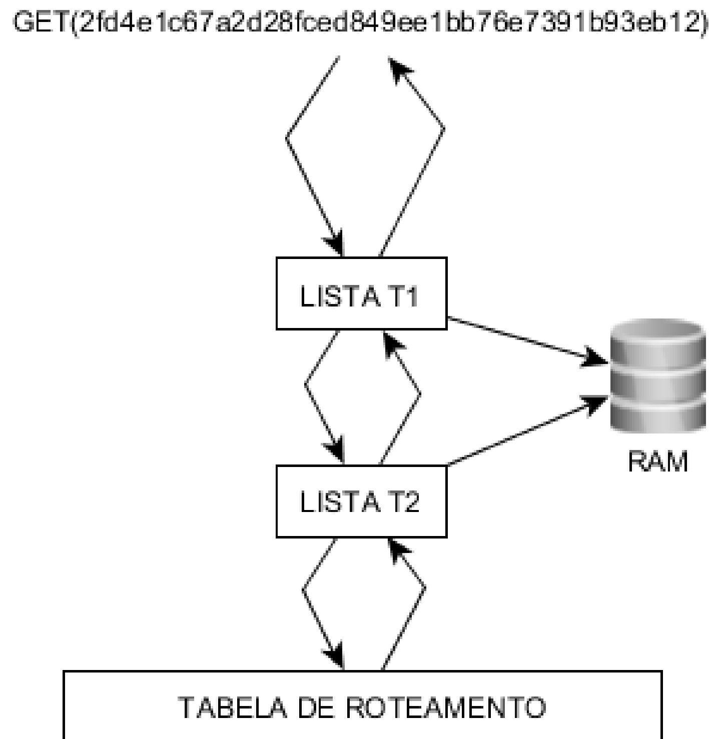


Figura 3.10: Representação do acesso ao cache em cada um dos nós do sistema distribuído

3.8 Considerações Parciais

Neste capítulo foi apresentada a arquitetura Griddler. Foram descritos os principais aspectos estruturais do trabalho desenvolvido, visto que parte da motivação e foco deste trabalho está voltada a explorar uma arquitetura totalmente descentralizada para armazenamento de dados. Foram detalhadas tecnicamente cada uma das funcionalidades implementadas. Foi também apresentada uma das principais contribuições do trabalho, que é utilizar o sistema de *cache* com a estratégia ARC, a qual combina políticas LRU e LFU. Assim, foram demonstradas algumas das especificações que tornam este trabalho único entre os demais encontrados na literatura.

CAPÍTULO 4 – Experimentos e Resultados

4.1 Considerações iniciais

Neste capítulo são apresentados os experimentos realizados no sentido de avaliar a arquitetura Griddler efetivamente. Isso é feito com o objetivo de validar as estratégias sugeridas e as vantagens que decorrem de um sistema distribuído P2P conforme o proposto neste trabalho. Em particular, os testes avaliaram quesitos de escalabilidade, tolerância a falhas e principalmente qualidade no acesso. Em termos do sistema de *cache* foram avaliados os tempos de resposta com e sem a estratégia sugerida, para verificar se o uso de *cache* realmente agiliza a busca na rede. Espera-se, com a exposição dos respectivos testes, demonstrar otimizações frente a outros trabalhos correlatos. Os testes apresentados foram realizados sob três aspectos da arquitetura de armazenamento desenvolvida: latência de acesso, tempo de processamento na codificação e decodificação dos dados, e sobrecarga devido à redundância. Num primeiro momento apenas os dados obtidos a partir deste trabalho são apresentados, para ao final dedicar uma seção exclusivamente para comparações com os demais trabalhos da área e discussões acerca dos resultados obtidos.

4.2 Ambiente de testes e Conjunto de dados

Quatro experimentos foram conduzidos, e serão detalhados a seguir. Nesta fase de testes, o ambiente usou a virtualização de *hardware* em um único computador físico. As especificações gerais do computador físico consistem de um processador Intel Core i7 Haswell - 4700MQ 2,4 GHz, 6 MB de cache (3,40 GHz com Max Turbo), 16 GB de memória Corsair Vengeance DDR3 (1600 MHZ) / (2x 8 GB) e um disco rígido de 1TB, a 7200 RPM. Sobre este hardware foram criados 6 servidores virtuais, cada um com acesso a uma única CPU, 2GB de RAM e 50GB de espaço de armazenamento, conectados à uma mesma rede virtual interna a eles. Todos os nós virtuais criados usam a versão de 64 bits do sistema operacional Linux. Os dados utilizados variaram entre conjuntos de objetos individuais com alguns megabytes até um conjunto de 1 terabyte com milhares de arquivos binários. Dessa forma, ao atingir a ordem do terabyte, o conjunto de dados é se aproxima um pouco mais do que pode ser considerado um volume de dados desafiador em termos de armazenamento. O projeto consiste inteiramente em código C/C++, incluindo algumas bibliotecas de *software*.
Foram incluídos em alguns momentos o uso de testes-t (STUDENT, 1908) (HOPKINS; GLASS; HOPKINS, 1987) para reforçar a relevância dos resultados com base em comprovações obtidas estatisticamente. O intuito dos testes-t é determinar se a diferença entre as médias dos resultados que foram obtidos realmente se aplica a

todos os casos para quaisquer amostras de dois grupos comparados ou se foi causada especificamente para a amostra em questão. Em geral isso é mais provável de acontecer quando a diferença entre as médias dos resultados é maior, quando as amostras utilizadas são maiores, e quando os valores obtidos para as diversas medições se encontram consistentemente próximos à média.

4.3 Latência de acesso

O primeiro experimento teve como objetivo verificar a latência de acesso a dados, com e sem o uso do *cache*. Para esta experiência, o modelo de medição de latência seguiu as definições de RFC 2544 (BRADNER; MCQUAID, 1999), um padrão comum para cálculo da latência de rede e cujas definições foram expandidas nesse teste para latência do disco. Dessa forma, foi calculada a média de 20 medições realizadas para cada objeto de dados, que neste caso são arquivos binários. Esses objetos haviam sido inseridos previamente no sistema de armazenamento, que para fins de simplificação tinha apenas um nó ativo. Ou seja, todos os objetos estavam armazenados em um mesmo nó e um mesmo disco rígido, o que permite testar as capacidades do *cache*, que é local. As medições foram realizadas a partir da variação do tamanho dos objetos armazenados com replicação de três vias ou com códigos de correção de erros. Para este e os testes seguintes, o algoritmo de codificação utilizado foi o Liberation (PLANK, 2008) com parâmetros (6,2). Esse código foi escolhido por ser MDS, ou seja, possuir eficiência ótima em termos de armazenamento e, em comparação com os demais, apresentar boa performance para tempos de codificação e decodificação (PLANK, 2008). Ao mesmo tempo, os parâmetros de algoritmo foram passados de tal a forma a gerar 6 partes, das quais até 2 podem ser perdidas simultaneamente, o que é o equivalente em termos de redundância da replicação em três vias. Os resultados da Tabela 4.1 e Fig. 4.1 representam medições para dados replicados. As medições dos dados armazenados por codificação seguem a Tabela 4.2 e a Fig. 4.2. Adicionalmente, os dados específicos de cada uma das observações estão disponíveis nas Tabelas 4.3, 4.4, 4.5 e 4.6.

Tabela 4.1: Latência média para diferentes objetos, replicação em 3x

Tamanho do arquivo	Sem cache	Com cache (ARC)
14 MB	0.075427s	0.027292s
277 MB	1.069707s	0.196161s
553 MB	2.176905s	0.385806s
1,1 GB	15.241922s	0.808186s

Tabela 4.2: Latência média para diferentes objetos, codificados

Tamanho do arquivo	Sem cache	Com cache (ARC)
26 MB	0.203074s	0.058756s
519 MB	3.565754s	0.551705s
1,1 GB	17.730196s	1.054749s
1,6 GB	21.805975s	1.269461s

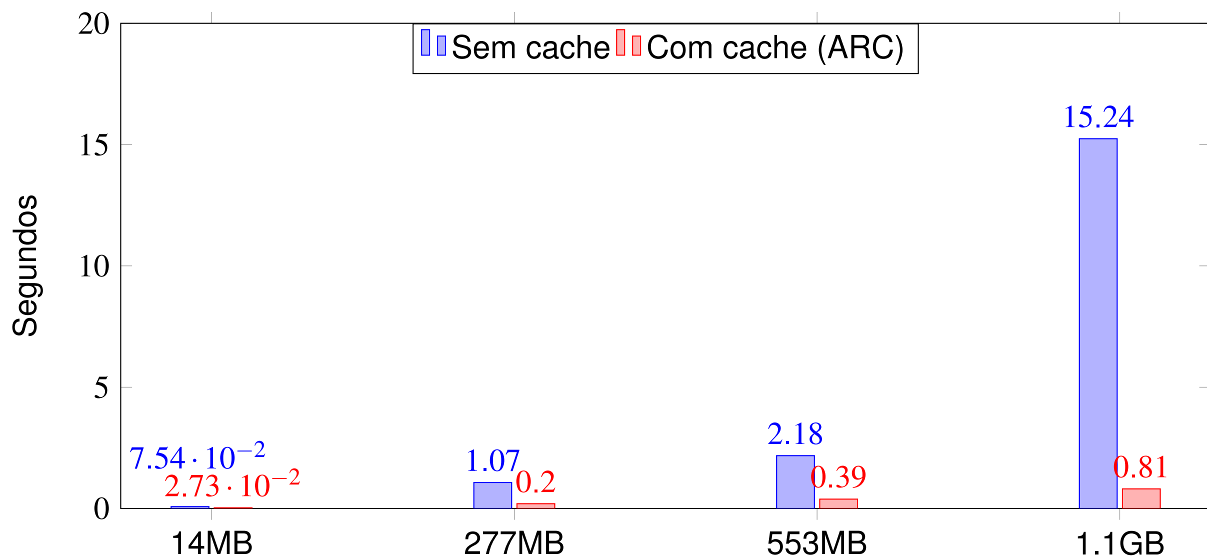


Figura 4.1: Gráfico da latência de acesso com e sem o uso de *cache* para dados replicados

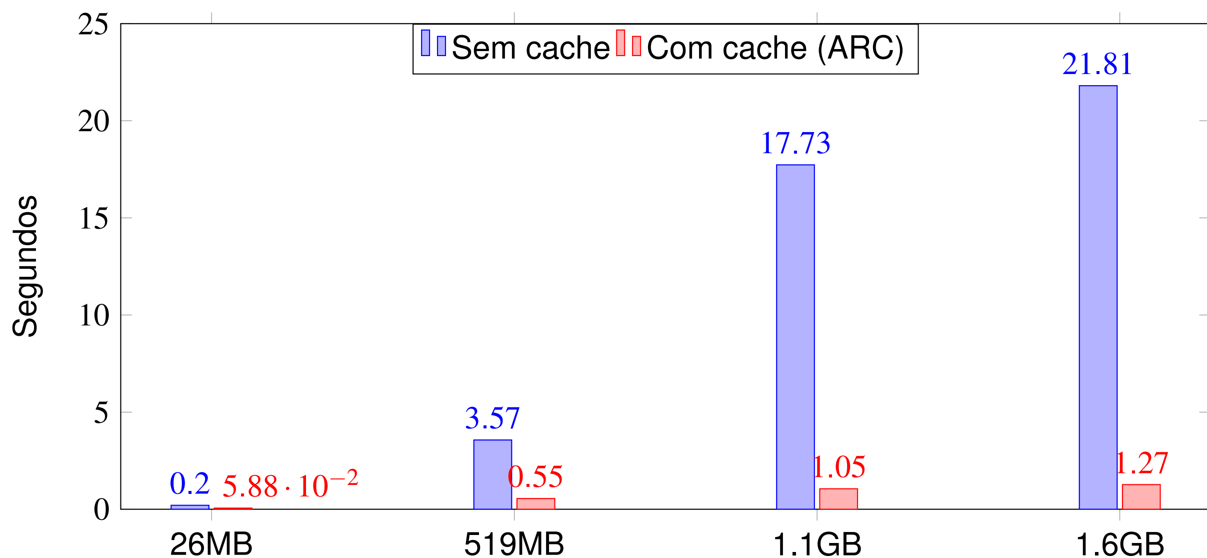


Figura 4.2: Gráfico da latência de acesso com e sem o uso de *cache* para dados codificados

Tabela 4.3: Observações de latência para diferentes objetos, replicação em 3x, sem cache

Observações	14MB	277MB	553MB	1,1GB
1	0.070662s	1.056696s	2.202838s	17.706871s
2	0.074368s	1.047700s	2.147036s	16.431497s
3	0.085687s	1.061151s	2.123224s	14.679129s
4	0.076863s	1.058246s	2.220209s	14.799051s
5	0.073702s	1.070705s	2.153540s	14.441770s
6	0.082172s	1.049061s	2.136225s	14.895479s
7	0.073562s	1.053181s	2.273718s	15.109067s
8	0.077908s	1.107263s	2.212358s	15.439353s
9	0.072300s	1.097814s	2.152454s	14.725767s
10	0.073370s	1.098210s	2.174496s	15.442988s
11	0.074858s	1.067940s	2.215891s	15.477651s
12	0.072496s	1.071386s	2.217964s	15.417187s
13	0.072513s	1.080854s	2.145907s	14.785722s
14	0.078076s	1.084609s	2.209091s	15.918755s
15	0.073838s	1.068076s	2.147773s	15.687626s
16	0.073522s	1.067607s	2.179776s	14.888277s
17	0.074543s	1.073403s	2.126565s	14.593281s
18	0.074962s	1.053625s	2.134893s	14.793862s
19	0.072612s	1.061112s	2.125586s	14.508928s
20	0.080524s	1.065505s	2.238565s	15.096185s

Embora a diferença de tempo seja visualmente expressiva, é interessante reforçar os resultados de modo a comprovar que os valores obtidos são estatisticamente relevantes. Como todas as medições foram feitas para um mesmo conjunto de dados em dois contextos, com e sem o uso de estruturas de *cache*, é possível realizar um teste-t. Em resumo, um teste desse tipo analisa a média de duas amostras de dados em um experimento, as quais devem representar resultados obtidos antes e depois de alguma influência externa. Parte-se de duas hipóteses: H0, em que se supõe que não há diferença significativa entre os resultados obtidos com e sem a influência externa, e H1, em que se supõe que há diferença significativa entre os resultados obtidos. Com base na média das amostras são calculadas algumas estatísticas, com destaque para a estatística p-value, que auxilia a identificar se os resultados obtidos efetivamente representam um padrão nos dados e não são apenas resultados isolados. Quanto menor o valor para p-value maior a relevância da hipótese H1. Esse teste só é possível se os resultados obtidos obedecerem à distribuição normal, ou se enquadrarem como uma aproximação da distribuição normal. Para todos os conjuntos de dados avaliados foi realizado o teste de normalidade de Anderson-Darling (ANDERSON; DARLING, 1954), que embora tenha sido proposto há muitas décadas ainda é considerado um dos

Tabela 4.4: Observações de latência para diferentes objetos, replicação em 3x, com cache

Observações	14MB	277MB	553MB	1,1GB
1	0.026811s	0.194611s	0.389896s	0.795928s
2	0.027964s	0.196103s	0.392334s	0.794260s
3	0.028692s	0.196722s	0.397790s	0.789526s
4	0.028610s	0.195807s	0.385075s	0.800920s
5	0.027044s	0.204743s	0.387494s	0.805033s
6	0.028017s	0.204636s	0.380040s	0.806491s
7	0.026883s	0.195281s	0.381494s	0.808139s
8	0.027059s	0.194899s	0.387622s	0.809932s
9	0.027305s	0.193923s	0.381812s	0.817593s
10	0.027195s	0.199886s	0.382815s	0.822805s
11	0.029716s	0.194831s	0.382975s	0.810284s
12	0.026335s	0.193972s	0.384650s	0.804997s
13	0.026420s	0.193825s	0.382523s	0.818729s
14	0.026114s	0.196464s	0.384854s	0.818012s
15	0.028666s	0.192456s	0.385989s	0.812859s
16	0.026258s	0.193641s	0.390892s	0.815763s
17	0.026822s	0.194564s	0.391965s	0.819201s
18	0.026277s	0.194819s	0.380844s	0.800175s
19	0.027207s	0.194672s	0.383457s	0.809836s
20	0.026461s	0.197379s	0.381598s	0.803238s

testes mais robustos para determinar uma função de distribuição com base em dados empíricos. Todos os conjuntos de dados, em maior ou menor grau, corresponderam bem ao teste de normalidade de Anderson-Darling, a partir do qual foi possível concluir que são todos aproximações da distribuição normal. Por esse motivo, foi possível realizar os testes-t que serão detalhados a seguir.
Ao todo foram realizados 8 testes-t, um para cada um dos objetos para os quais foram feitas as comparações de latência no acesso com e sem o uso de *cache*. Todos os cálculos foram realizados com o suporte da ferramenta Wessa (WESSA, 2016), usada por outros trabalhos publicados na literatura anteriormente (LARSEN et al., 2010). Os resultados obtidos são detalhados a seguir.

Teste 1
Objeto: 14MB, replicação 3x, com e sem cache

- Diferença entre as médias = 0.0481341
- t-stat = 59.8549180294139

Tabela 4.5: Observações de latência para diferentes objetos, codificados, sem cache

Observações	26MB	519MB	1,1GB	1,6GB
1	0.704742s	3.609652s	17.807491s	26.173406s
2	0.180549s	3.570050s	17.751033s	21.855750s
3	0.217023s	3.527834s	17.771326s	21.531253s
4	0.135916s	3.598784s	18.134677s	21.407851s
5	0.187200s	3.693366s	17.905386s	21.230319s
6	0.244305s	3.641080s	17.457883s	21.376667s
7	0.131676s	3.189544s	17.243182s	21.241226s
8	0.242322s	3.340807s	17.601377s	21.313464s
9	0.233407s	3.841463s	17.986012s	21.251941s
10	0.184020s	3.955369s	17.577673s	21.949062s
11	0.124435s	3.532882s	17.957571s	21.287696s
12	0.234626s	3.321190s	17.828751s	21.266521s
13	0.132824s	3.950442s	17.988029s	21.654600s
14	0.122081s	3.612754s	18.155430s	21.881317s
15	0.132638s	3.540965s	17.171402s	21.805993s
16	0.178381s	3.757771s	17.316517s	21.960053s
17	0.129403s	3.939509s	17.665861s	21.845208s
18	0.137330s	3.215846s	17.872807s	21.581198s
19	0.241778s	3.107018s	17.825759s	21.529584s
20	0.166832s	3.368765s	17.585757s	21.976400s

- graus de liberadade (df) = 19
- p-value = 4.16226546056268e-23
- Tipo de comparação = two.sided
- Nível de confiança = 0.95
- Intervalo de confiança = [0.0464509328943699,0.0498172671056301]

Teste 2
Objeto: 277MB, replicação 3x, com e sem cache

- Diferença entre as médias = 0.8735455
- t-stat = 224.541127194024
- graus de liberadade (df) = 19
- p-value = 5.3568336474551e-34
- Tipo de comparação = two.sided

Tabela 4.6: Observações de latência para diferentes objetos, codificados, com cache

Observações	26MB	519MB	1,1GB	1,6GB
1	0.063865s	0.525431s	1.061506s	1.292868s
2	0.058969s	0.535075s	1.076710s	1.304793s
3	0.057725s	0.513803s	1.079037s	1.341306s
4	0.057304s	0.566159s	1.071679s	1.334417s
5	0.064090s	0.505827s	1.081948s	1.307591s
6	0.058315s	0.558282s	1.041127s	1.266752s
7	0.056850s	0.512688s	1.086862s	1.279060s
8	0.061775s	0.597283s	1.063841s	1.272100s
9	0.057440s	0.579644s	1.030917s	1.279629s
10	0.057144s	0.559932s	1.011097s	1.237120s
11	0.057005s	0.593614s	1.083515s	1.225584s
12	0.057109s	0.509219s	1.041655s	1.275528s
13	0.057650s	0.565518s	1.017000s	1.236254s
14	0.056782s	0.543492s	1.073151s	1.234529s
15	0.057276s	0.562274s	1.030719s	1.244253s
16	0.059097s	0.527369s	1.052526s	1.272688s
17	0.058746s	0.584084s	1.061147s	1.241032s
18	0.059444s	0.595504s	1.026690s	1.298966s
19	0.058983s	0.535403s	1.015583s	1.244132s
20	0.059554s	0.563499s	1.088273s	1.200625s

- Nível de confiança = 0.95
- Intervalo de confiança = [0.865402885918704,0.881688114081296]

Teste 3

Objeto: 553MB, replicação 3x, com e sem cache

- Diferença entre as médias = 1.7910995
- t-stat = 176.259958132276
- graus de liberadade (df) = 19
- p-value = 5.31727079700717e-32
- Tipo de comparação = two.sided
- Nível de confiança = 0.95
- Intervalo de confiança = [1.76983083309991,1.81236816690009]

Teste 4
Objeto: 1,1GB, replicação 3x, com e sem cache

- Diferença entre as médias = 14.43373625
- t-stat = 83.1356471166018
- graus de liberadade (df) = 19
- p-value = 8.28448028067905e-26
- Tipo de comparação = two.sided
- Nível de confiança = 0.95
- Intervalo de confiança = [14.0703523321018,14.7971201678982]

Teste 5
Objeto: 26MB, codificação, com e sem cache

- Diferença entre as médias = 0.14431825
- t-stat = 5.15893987396049
- graus de liberadade (df) = 19
- p-value = 5.58134286088054e-05
- Tipo de comparação = two.sided
- Nível de confiança = 0.95
- Intervalo de confiança = [0.0857671569227665,0.202869343077233]

Teste 6
Objeto: 519MB, codificação, com e sem cache

- Diferença entre as médias = 3.01404955
- t-stat = 54.0828492883347
- graus de liberadade (df) = 19
- p-value = 2.82772344115701e-22
- Tipo de comparação = two.sided

- Nível de confiança = 0.95
- Intervalo de confiança = [2.89740484114423,3.13069425885577]

Teste 7
Objeto: 1,1GB, codificação, com e sem cache

- Diferença entre as médias = 16.67544705
- t-stat = 268.395181091016
- graus de liberadade (df) = 19
- p-value = 1.80823503412428e-35
- Tipo de comparação = two.sided
- Nível de confiança = 0.95
- Intervalo de confiança = [16.5454070408918,16.8054870591082]

Teste 8
Objeto: 1,6GB, codificação, com e sem cache

- Diferença entre as médias = 20.5365141
- t-stat = 86.4383253459635
- graus de liberadade (df) = 19
- p-value = 3.95922902463705e-26
- Tipo de comparação = two.sided
- Nível de confiança = 0.95
- Intervalo de confiança = [20.0392414164657,21.0337867835343]

Ao efetuar a análise dos resultados é possível observar valores muito baixos para o p-value, que é o principal ponto de importância nesse tipo de teste. Valores de p-value tão baixos significam, em todos os casos, que os resultados obtidos são relevantes estatisticamente. Dessa forma é possível concluir que as médias obtidas representam não somente as amostras utilizadas mas, com um alto grau de confiança, se repetiriam em novos testes e com amostras maiores. Dessa forma o uso do *cache* se mostrou de fato importante e representou ganhos efetivos na latência.

4.4 Codificação e Decodificação

O segundo experimento teve por objetivo demostrar que os algoritmos utilizados para codificação e decodificação são robustos o suficiente para processar grandes volumes de dados mesmo em um único nó, em tempo hábil. Dessa forma, uma arquitetura totalmente descentralizada, além de não ter pontos únicos de falha, permite explorar ao máximo as capacidades do sistema e não há necessidade de existirem mestres ou nós com funções específicas de processamento, armazenamento e gerenciamento como trabalhos anteriores sugeriram. Neste experimento um número variável *n* de objetos de dados binários, cada um com 10MB de tamanho, compôs cada conjunto de dados. O tamanho total dos conjuntos de dados variou de 50 GB a 1 TB de dados e as operações foram executadas em um único nó. Os resultados estão ilustrados na Tabela 4.7 e na Fig. 4.3.

Tabela 4.7: Tempos de codificação para diferentes volumes de dados

n	Tamanho total	Tempo decorrido
5000	50 GB	131.390s
10000	100 GB	269.676s
50000	500 GB	1389.382s
100000	1 TB	3063.399s

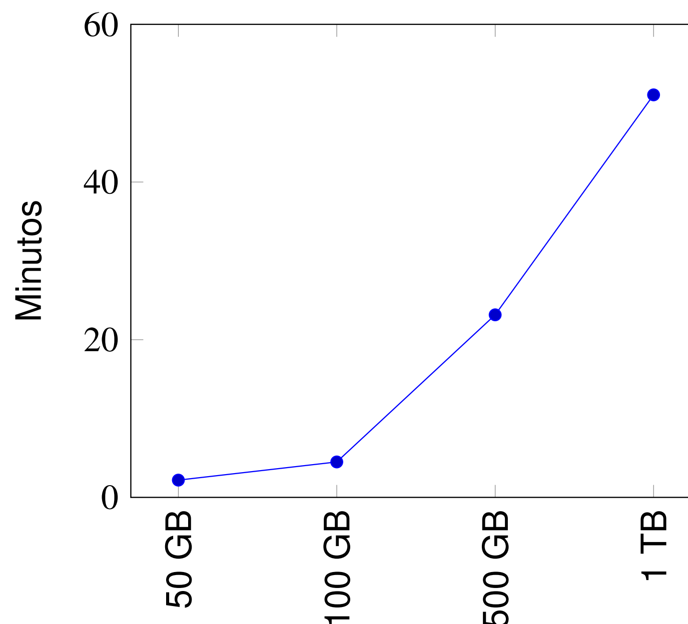


Figura 4.3: Tempo decorrido na codificação de diferentes conjuntos de dados

De um modo semelhante, o objetivo é demostrar que é possível explorar o poder de processamento de cada nó e que isto permite contribuir para operações de decodificação dos dados à medida em que o sistema seja totalmente descentralizado. Neste experimento um número variável *n* de objetos de dados binários, cada um com 10MB de tamanho, compôs cada conjunto de dados. O tamanho total dos conjuntos de dados variou de 50 GB a 1 TB de dados e as operações foram executadas em um único nó. Os resultados seguem na Tabela 4.8 e na Fig. 4.4.

Tabela 4.8: Tempos de decodificação para diferentes volumes de dados

n	Tamanho total	Tempo decorrido
5000	50 GB	89.237s
10000	100 GB	183.615s
50000	500 GB	874.102s
100000	1 TB	1914.271s

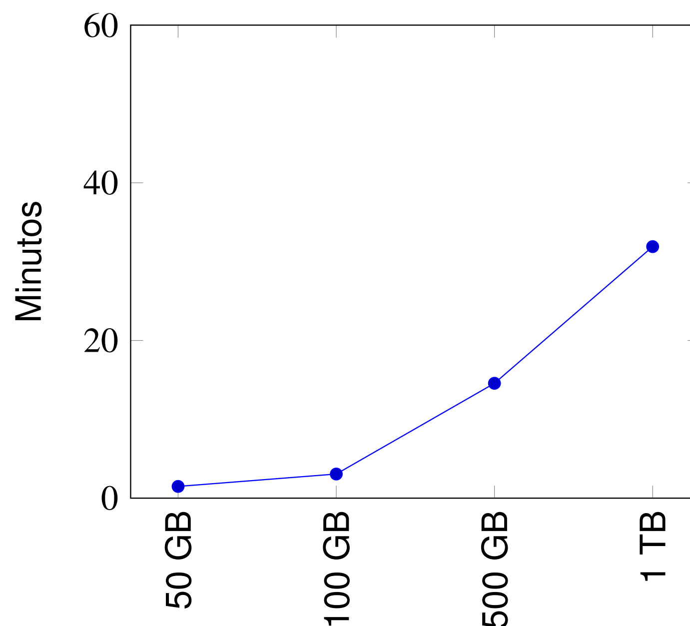


Figura 4.4: Tempo decorrido na decodificação de diferentes conjuntos de dados

Na rede *peer-to-peer* em anel, utilizada na arquitetura proposta neste trabalho, cada nó é capaz de receber e processar simultaneamente pedidos como os testados neste experimento, visto que estas operações são independentes. Foi possível observar que, embora o comportamento do tempo de execução tenha características exponenciais, é possível processar dezenas de *gigabytes* num único nó em poucos minutos. Dessa forma é possível estimar que um número maior de nós de processamento em paralelo iria diminuir a codificação e decodificação globalmente, em especial quando a

quantidade de dados atinge a ordem de *terabytes*. O modelo *peer-to-peer* permite a utilização de todos os nós em tais situações, e tem os benefícios de não se limitar a pontos únicos de falha, o que não foi considerado em trabalhos anteriores.

4.5 Sobrecarga de armazenamento

O terceiro experimento pretende mostrar que, embora a codificação requeira processamento adicional, ela tem muito menos sobrecarga quando comparada à replicação pura. As observações foram realizadas com variação do tamanho dos objetos utilizados, da ordem de 50MB até 1GB. Os resultados seguem na Tabela 4.9 e na Fig. 4.5 e para melhor precisão representam o número total de *bytes* após a inclusão da redundância. É possível perceber que para um mesmo tamanho de objeto original sem redundância, a opção de redundância por meio da codificação dos dados possibilita obter um número total de *bytes* armazenados consideravelmente menor do que se utilizasse replicação. Neste experimento, o algoritmo de codificação utilizado ainda foi o Liberation com parâmetros (6,2), de modo que a tolerância a falhas obtida é igual a replicação em 3 vias, mas com uma sobrecarga de armazenamento menor. Esta característica das técnicas de codificação é particularmente interessante para arquivos maiores, visto que para estes o espaço em disco exigido pela replicação pode consumir um espaço em disco muito maior se utilizado em larga escala.

4.6 Discussão dos resultados e próximos passos

Em relação ao mecanismo de *cache* utilizado, dado que o algoritmo ARC é notavelmente uma melhoria quando comparado ao algoritmo LRU em termos de taxa de acerto – vide comparação apresentada na Fig. 2.11 – a primeira questão a ser analisada era se um sistema de *cache* local em cada nó de um sistema de armazenamento em rede P2P fornece quaisquer benefícios para o acesso a dados. Como foi mostrado na primeira experiência, o sistema de cache introduzido reduz drasticamente os tempos de acesso em cada nó, proporcionando mais de 80% de melhoria na latência de acesso para dados replicados e codificados quando comparado a uma alternativa sem *cache*. Como cada nó pode ativamente processar solicitações, em um ambiente real, vários usuários se beneficiarão de respostas mais rápidas. Como na Griddler o algortimo ARC foi implementado na memória principal, era esperada uma latência menor do que o disco rígido. Dessa forma, é possível concluir que o uso de ARC em vez de LRU indica uma melhoria com relação ao trabalho que propõe a arquitetura CAROM (MA et al., 2013), que usa o LRU. Quando comparado aos demais trabalhos, que não têm qualquer estrutura de otimização no acesso, a arquitetura proposta se destaca ainda

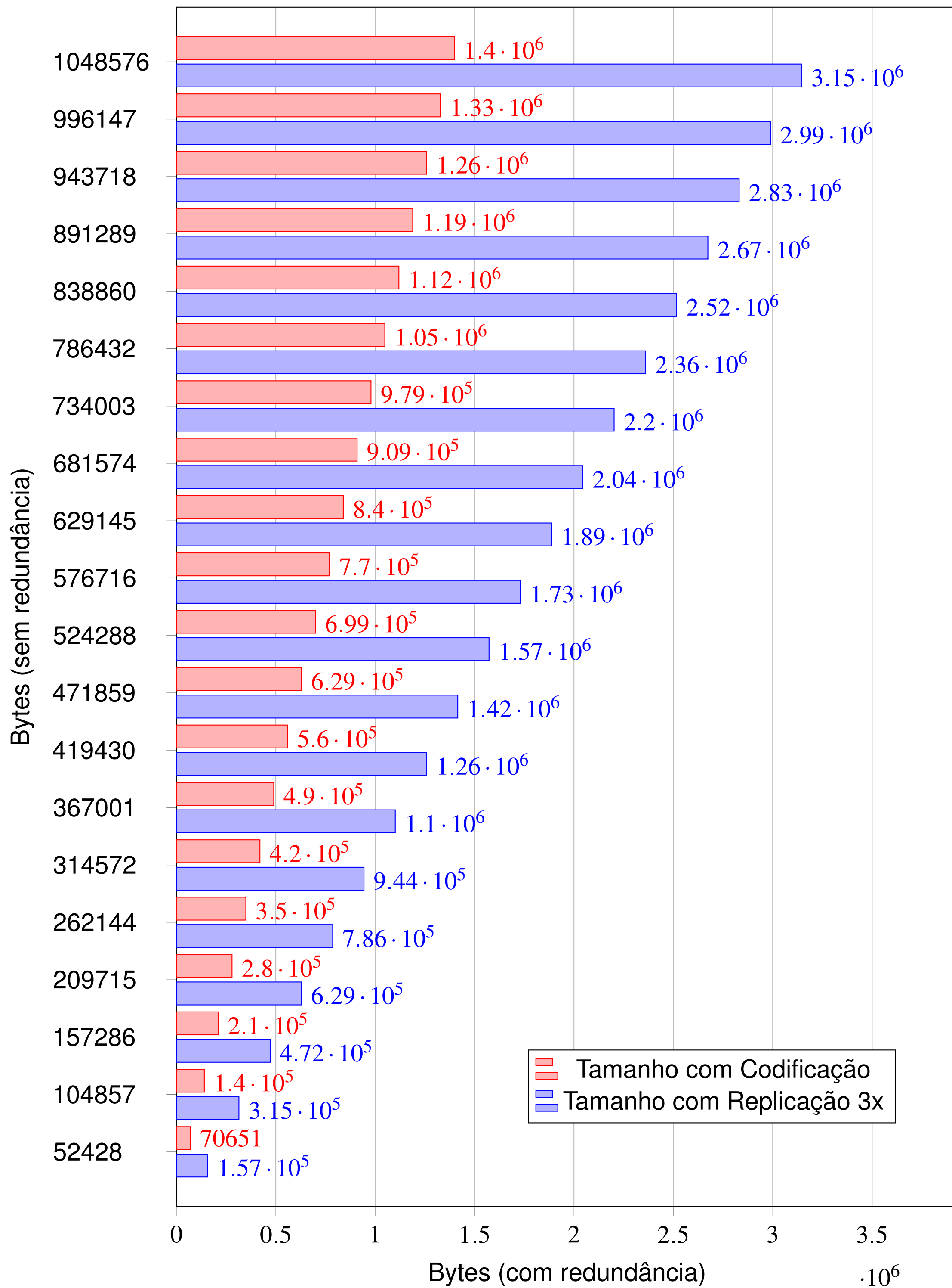


Figura 4.5: Comparação de sobrecarga para ambas as técnicas de redundância

Tabela 4.9: Sobrecarga para redundância no armazenamento de dados binários

Tamanho do Objeto	Tamanho com Replicação 3x	Tamanho com Codificação
52428 B	157284 B	70651 B
104857 B	314571 B	140382 B
157286 B	471858 B	210112 B
209715 B	629145 B	279842 B
262144 B	786432 B	349573 B
314572 B	943716 B	420220 B
367001 B	1101003 B	489951 B
419430 B	1258290 B	559681 B
471859 B	1415577 B	629411 B
524288 B	1572864 B	699142 B
576716 B	1730148 B	769790 B
629145 B	1887435 B	839520 B
681574 B	2044722 B	909250 B
734003 B	2202009 B	978980 B
786432 B	2359296 B	1048711 B
838860 B	2516580 B	1119359 B
891289 B	2673867 B	1189089 B
943718 B	2831154 B	1258819 B
996147 B	2988441 B	1328549 B
1048576 B	3145728 B	1398280 B

mais.

O segundo experimento mostrou que os algoritmos atuais para codificação de dados têm tempo de execução viável em cada nó. Portanto, é natural querer maximizar o número de nós de processamento e, nesse sentido, o modelo P2P fornece uma solução mais próxima do ideal, pois todos os nós contribuem ativamente para todo o sistema. Ao assumir uma rede de n nós em paralelo, como um *cluster* em um ambiente em nuvem, os trabalhos anteriores usavam apenas n - k desses nós para codificação, mas com a abordagem proposta neste trabalho os ganhos reais de desempenho em termos de tempo de processamento podem eventualmente ultrapassar em muitas vezes os demais trabalhos, dado que todos os n nós da rede processam dados ativamente. A separação entre nós de processamento e nós de armazenamento só adicionaria tempo de transferência de rede adicional e, por esse motivo, a arquitetura Griddler usa nós para ambas as funções. Esta é também uma melhoria em trabalhos anteriores, como o trabalho que propõe a arquitetura MICS (TANG et al., 2015).

O terceiro experimento reforçou a importância dos códigos de correção de erros em termos de custos gerais de armazenamento e sobrecarga, e dessa forma demonstra que a arquitetura proposta supera trabalhos anteriores limitados a apenas uma das

técnicas. Quando comparado com a replicação de três vias, os mesmos dados, quando codificados, incorrem em torno de 55% menos sobrecarga, na média. Quando se trata de Big Data, essa característica permitiria maiores volumes de dados armazenados para o uso de um mesmo *hardware*. Quando o tempo de processamento é um aspecto mais desejável do que a sobrecarga de armazenamento, o sistema proposto é capaz de armazenar dados de forma replicada também, o que sugere uma melhoria quando comparado há alguns trabalhos anteriores que se limitaram apenas à codificação, como é o caso do trabalho que propõe a arquitetura Robot (YIN et al., 2013).

4.7 Considerações Parciais

Neste capítulo foram apresentados alguns experimentos para validar os principais aspectos da arquitetura de armazenamento P2P Griddler, além de comparar os resultados obtidos com o de trabalhos correlatos da literatura. A arquitetura desenvolvida se mostrou eficiente nos principais aspectos avaliados, e superou em alguns pontos as arquiteturas similares existentes. Em particular, o uso do algoritmo de *caching* para melhoria do acesso e a capacidade de execução dos algoritmos de codificação em tempo hábil demonostraram características desejáveis para desafios na área de Big Data. Com tais resultados, é esperado que seja possível criar um ambiente base para futuros trabalhos que possam se valer de uma arquitetura de armazenamento desse tipo.

CAPÍTULO 5 – Considerações Finais

5.1 Discussão sobre o trabalho desenvolvido

O armazenamento eficiente e seguro dos dados é primordial em atividades que dependam do acesso contínuo aos dados. Ao mesmo tempo em que se presencia um aumento considerável no volume dos dados, o que demanda tecnologias inovadoras para aumentar a capacidade de armazenamento, é ainda mais importante utilizar técnicas de tolerância a falhas para garantia de disponibilidade, como replicação e códigos de correção de erros. Este cenário serviu de motivação para o trabalho, o qual buscou desenvolver uma arquitetura de armazenamento de dados *peer-to-peer* com tolerância a falhas mista, em que se combina as duas técnicas mencionadas, de forma automatizada e configurável. Como contribuição adicional foi implementado um esquema de *caching*, a partir do qual foi possível obter uma melhora da qualidade de acesso aos dados. A arquitetura proposta, para a qual foi dada o nome Griddler, leva em conta ainda a utilização de um ambiente totalmente descentralizado de armazenamento, o que é algo que ainda não havia sido considerado em alguns trabalhos semelhantes. Para elaboração dessa proposta, foi realizado inicialmente um levantamento bibliográfico das áreas de armazenamento de dados, sistemas distribuídos, e códigos de correção de erros, em que foram encontrados diferentes tipos de tecnologias e arquiteturas semelhantes à proposta. Com base em sugestões de melhorias e carências do estado-da-arte é que foram definidos os objetivos base para o trabalho e, a partir deles, a arquitetura desenvolvida foi descrita e cada etapa de seu funcionamento foi detalhada. Para a validação da Griddler, seu resultado foi comparado com o de trabalhos correlatos da literatura. Foram medidos tempo de acesso (escrita e leitura) dos dados no sistema distribuído, além de outros fatores como sobrecarga de armazenamento. Por fim, é evidente que este trabalho encontra diversas aplicações em situações reais e se espera, ao consolidar seu desenvolvimento, colocá-lo à disposição em situações que dependam de armazenamento de dados distribuído em larga escala. Na Tabela 5.1 está representado o trabalho proposto em comparação com seus trabalhos correlatos. Nessa tabela estão destacados os principais diferenciais da arquitetura proposta: ser totalmente descentralizada e utilizar técnicas de *caching* otimizadas para facilitar o acesso aos dados, além de possuir tolerância a falhas mista. A partir dos resultados obtidos foi possível validar as hipóteses feitas, visto que os diferenciais implementados apresentaram vantagens significativas para um sistema de armazenamento distribuído.

5.2 Contribuições do trabalho

A seguir é apresentada uma lista com as principais melhorias do trabalho desenvolvido em relação aos trabalhos correlatos, com o objetivo de destacar as contribuições do trabalho realizado:

- Integração de técnicas de *caching* customizável para otimização de acesso em sistemas de armazenamento distribuído, com a utilização do algoritmo ARC. Esse algoritmo é uma combinação dos algoritmos LRU e LFU.
- Arquitetura totalmente descentralizada para armazenamento de objetos, baseada em ambiente *peer-to-peer* em anel com o algoritmo Chord para gerenciamento dos dados e otimização nas buscas. Este recurso é caracterizado como uma das diferenças do ambiente descentralizado proposto em comparação com outros existentes.
- Implementação de algoritmos de codificação de dados para adicionar redundância aos dados armazenados, ao mesmo tempo que disponibiliza mecanismos de replicação quando necessário. Ao contemplar a possibilidade de simultaneamente armazenar dados codificados e replicados, se destaca entre os demais trabalhos da literatura.

Tabela 5.1: Comparação entre os trabalhos correlatos e o trabalho proposto

Aspectos arquiteturais \ Nome do trabalho	CAROM	MICS	HRSPC	Robot	HDFS-Xorbas	Griddler
Redundância	R/EC	R/EC	R/EC	EC	EC	R/EC
Estruturação	OS	OS	OS	OS	OS	OS
Design	ME	ME	ME/P2P	ME/P2P	ME	P2P
Armazenamento	HDD	HDD	HDD	HDD	HDD	HDD
Processamento	CPU	CPU	CPU	CPU	CPU	CPU
Cache	LRU	-	-	-	-	ARC

R = replicação; EC = erasure coding; OS = object storage; ME = mestre-escravo.

5.3 Trabalhos futuros

A proposta implementada permite identificar ainda vários pontos que podem ser melhorados, muito embora as funcionalidades principais tenham sido implementadas. Um dos pontos de interesse nesse sentido é quanto à segurança do sistema de armazenamento, visto que isto ainda não foi contemplado. Para que a arquitetura proposta seja

utilizada em larga escala é preciso principalmente investir em mecanismos de controle de acesso, bem como analisar a possibilidade de incluir criptografia nos dados armazenados. Com relação à criptografia é bastante interessante verificar o comportamento de algoritmos extremamente seguros como AES256 para grandes volumes de dados, e no momento algumas implementações nesse sentido foram feitas mas ainda carecem de testes mais detalhados. Para o armazenamento, planeja-se atualizar a cifra de *hashing* utilizada para outras mais recentes, como por exemplo SHA-256, para diminuir a chance de conflitos e aumentar o número de identificadores possíveis. Ainda em termos do armazenamento, é preciso melhorar a forma de gerenciamento dos metadados dos objetos armazenados, visto que são essenciais na recuperação de dados codificados. O ideal é extrair o máximo em termos de meta-dados na etapa de armazenamento, para poder oferecer essas informações posteriormente caso se deseje realizar algum processo de prospecção sobre o que está armazenado. Uma outra ideia ainda em fase de planejamento é a instalação do software base da arquitetura em um sistema computacional maior, como um ambiente em Nuvem, para analisar seu comportamento por períodos de tempo maiores e com um maior número de usuários. Para atrair os usuários será preciso criar uma interface *web* que facilite a interação com o sistema de armazenamento, de forma a simplificar o máximo possível a sintaxe necessária para realizar suas operações. Também se trabalha com a ideia da implementação no futuro de um mecanismo de versionamento dos objetos armazenados, de modo a poder controlar alterações realizadas nos mesmos.

Referências Bibliográficas

ABHIJITH, U.; TARANISEN, M.; KUMAR, A.; MUDDI, L. The efficient use of Storage Resources in SAN for Storage Tiering and Caching. In: IEEE. COMPUTATIONAL INTELLIGENCE AND NETWORKS (CINE), 2016 2ND INTERNATIONAL CONFERENCE ON. 2016. **Anais...**, [S.l.], 2016. p. 118–122.

AGGARWAL, V.; CHEN, Y.-F. R.; LAN, T.; XIANG, Y. Sprout: A functional caching approach to minimize service latency in erasure-coded storage. In: IEEE. DISTRIBUTED COMPUTING SYSTEMS (ICDCS), 2016 IEEE 36TH INTERNATIONAL CONFERENCE ON. 2016. **Anais...**, [S.l.], 2016. p. 753–754.

AL-KISWANY, S.; GHARAIBEH, A.; SANTOS-NETO, E.; YUAN, G.; RIPEANU, M. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In: ACM. PROCEEDINGS OF THE 17TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING. 2008. **Anais...**, [S.l.], 2008. p. 165–174.

ALNAFOOSI, A. B.; STEINBACH, T. An integrated framework for evaluating big-data storage solutions-IDA case study. In: IEEE. SCIENCE AND INFORMATION CONFERENCE (SAI), 2013. 2013. **Anais...**, [S.l.], 2013. p. 947–956.

ANDERSON, D. P. Boinc: A system for public-resource computing and storage. In: IEEE. GRID COMPUTING, 2004. PROCEEDINGS. FIFTH IEEE/ACM INTERNATIONAL WORKSHOP ON. 2004. **Anais...**, [S.l.], 2004. p. 4–10.

ANDERSON, T. W.; DARLING, D. A. A test of goodness of fit. **Journal of the American statistical association**, Taylor & Francis, v. 49, n. 268, p. 765–769, 1954.

ASSUNÇÃO, M. D.; CALHEIROS, R. N.; BIANCHI, S.; NETTO, M. A.; BUYYA, R. Big Data computing and clouds: Trends and future directions. **Journal of Parallel and Distributed Computing**, Elsevier, v. 79, p. 3–15, 2015.

BIAN, J.; SEKER, R. The Jigsaw secure distributed file system. **Computers & Electrical Engineering**, Elsevier, v. 39, n. 4, p. 1142–1152, 2013.

BORTHAKUR, D. HDFS architecture guide. **HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf**, p. 39, 2008.

BRADNER, S.; MCQUAID, J. **Benchmarking methodology for network interconnect devices**. [S.l.], 1999.

CAIWEI, D.; LEI, W.; LIANSHENG, S. Load balancing technology based on erasure code in distributed storage system. In: IEEE. CONTROL CONFERENCE (CCC), 2012 31ST CHINESE. 2012. **Anais...**, [S.l.], 2012. p. 5558–5563.

CARON, S.; GIROIRE, F.; MAZAURIC, D.; MONTEIRO, J.; PÉRENNES, S. P2P storage systems: Study of different placement policies. **Peer-to-Peer Networking and Applications**, Springer, v. 7, n. 4, p. 427–443, 2014.

CARPENTIER, P. Replication and Erasure Coding Explained. **Caringo Elastic Content Protection Technical Overview, White Paper**, 2013.

CURRY, M. L.; SKJELLUM, A.; WARD, H. L.; BRIGHTWELL, R. Gibraltar: A Reed-Solomon coding library for storage applications on programmable graphics processors. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 23, n. 18, p. 2477–2495, 2011.

DATTA, A.; OGGIER, F. An overview of codes tailor-made for better repairability in networked distributed storage systems. **ACM SIGACT News**, ACM, v. 44, n. 1, p. 89–105, 2013.

DELL'AMICO, M.; MICHIARDI, P.; TOKA, L.; CATALDI, P. Adaptive redundancy management for durable P2P backup. **Computer Networks**, Elsevier, v. 83, p. 136–148, 2015.

DENG, M.-Z.; CHEN, Z.-G.; DU, Y.-M.; XIAO, N.; LIU, F. Erasure codes in big data era. In: IEEE. CONTROL, AUTOMATION AND INFORMATION SCIENCES (ICCAIS), 2014 INTERNATIONAL CONFERENCE ON. 2014. **Anais...**, [S.l.], 2014. p. 218–223.

DUMINUCO, A.; BIERSACK, E. Hierarchical codes: How to make erasure codes attractive for peer-to-peer storage systems. In: IEEE. PEER-TO-PEER COMPUTING, 2008. P2P'08. EIGHTH INTERNATIONAL CONFERENCE ON. 2008. **Anais...**, [S.l.], 2008. p. 89–98.

EASTLAKE, D.; JONES, P. RFC 3174-US Secure Hash Algorithm 1 (SHA1)(2001). **URL http://www. ietf. org/rfc/rfc3174. txt**, 2012.

EDITION, I. P. C. Software Defined Storage For Dummies. **New Jersey**, p. 4–5, 2014.

ESINER, E.; DATTA, A. Layered security for storage at the edge: on decentralized multi-factor access control. In: ACM. PROCEEDINGS OF THE 17TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING AND NETWORKING. 2016. **Anais...**, [S.l.], 2016. p. 9.

FACTOR, M.; METH, K.; NAOR, D.; RODEH, O.; SATRAN, J. Object storage: The future building block for storage systems. In: IEEE. 2005 IEEE INTERNATIONAL SYMPOSIUM ON MASS STORAGE SYSTEMS AND TECHNOLOGY. 2005. **Anais...**, [S.l.], 2005. p. 119–123.

FIKES, A. Storage architecture and challenges. **Talk at the Google Faculty Summit**, 2010.

GANDOMI, A.; HAIDER, M. Beyond the hype: Big data concepts, methods, and analytics. **International Journal of Information Management**, Elsevier, v. 35, n. 2, p. 137–144, 2015.

GANTZ, J.; REINSEL, D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. **IDC iView: IDC Analyze the future**, v. 2007, p. 1–16, 2012.

GONIZZI, P.; FERRARI, G.; GAY, V.; LEGUAY, J. Data dissemination scheme for distributed storage for IoT observation systems at large scale. **Information Fusion**, Elsevier, v. 22, p. 16–25, 2015.

GREGG, C.; HAZELWOOD, K. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: IEEE. PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE (ISPASS), 2011 IEEE INTERNATIONAL SYMPOSIUM ON. 2011. **Anais...**, [S.l.], 2011. p. 134–144.

GRIBAUDO, M.; IACONO, M.; MANINI, D. Improving reliability and performances in large scale distributed applications with erasure codes and replication. **Future Generation Computer Systems**, Elsevier, v. 56, p. 773–782, 2016.

HASHEM, I. A. T.; YAQOOB, I.; ANUAR, N. B.; MOKHTAR, S.; GANI, A.; KHAN, S. U. The rise of “big data” on cloud computing: Review and open research issues. **Information Systems**, Elsevier, v. 47, p. 98–115, 2015.

HOPKINS, K. D.; GLASS, G. V.; HOPKINS, B. **Basic statistics for the behavioral sciences**. [S.l.]: Prentice-Hall, Inc, 1987.

HSIEH, J.-W.; SU, C.-J. Parity management scheme for a hybrid-storage RAID. In: ACM. PROCEEDINGS OF THE 31ST ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING. 2016. **Anais...**, [S.l.], 2016. p. 1774–1776.

JEDDA, A.; MOUFTAH, H. T. Enhancing DHT-based object naming service architectures with geographic-awareness. In: IEEE. NETWORK OF THE FUTURE (NOF), 2015 6TH INTERNATIONAL CONFERENCE ON THE. 2015. **Anais...**, [S.l.], 2015. p. 1–6.

JIN, X.; WAH, B. W.; CHENG, X.; WANG, Y. Significance and challenges of big data research. **Big Data Research**, Elsevier, v. 2, n. 2, p. 59–64, 2015.

KANG, S.-H.; KOO, D.-H.; KANG, W.-H.; LEE, S.-W. A case for flash memory ssd in hadoop applications. **International Journal of Control and Automation**, v. 6, n. 1, p. 201–210, 2013.

KAPADIA, A.; RAJANA, K.; VARMA, S. **OpenStack Object Storage (Swift) Essentials**. [S.l.]: Packt Publishing Ltd, 2015.

KATSIGIANNIS, S.; DIMITSAS, V.; MAROULIS, D. A GPU vs CPU performance evaluation of an experimental video compression algorithm. In: IEEE. QUALITY OF MULTIMEDIA EXPERIENCE (QOMEX), 2015 SEVENTH INTERNATIONAL WORKSHOP ON. 2015. **Anais...**, [S.l.], 2015. p. 1–6.

KHAN, O.; BURNS, R. C.; PLANK, J. S.; PIERCE, W.; HUANG, C. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In: FAST. 2012. **Anais...**, [S.l.: s.n.], 2012. p. 20.

KO, A. C.; ZAW, W. T. Fault Tolerant Erasure Coded Replication for HDFS Based Cloud Storage. In: IEEE. BIG DATA AND CLOUD COMPUTING (BDCLOUD), 2014 IEEE FOURTH INTERNATIONAL CONFERENCE ON. 2014. **Anais...**, [S.l.], 2014. p. 104–109.

LAKSHMI, C. S.; KUMAR, G. S.; VENKATACHALAM, V. Survey on Caching and Replication Algorithm for Content Distribution in Peer To Peer Networks. **International Journal of Computer Science and Network Security (IJCSNS)**, International Journal of Computer Science and Network Security, v. 15, n. 10, p. 78, 2015.

LARSEN, N.; VOGENSEN, F. K.; BERG, F. W. van den; NIELSEN, D. S.; ANDREASEN, A. S.; PEDERSEN, B. K.; AL-SOUD, W. A.; SØRENSEN, S. J.; HANSEN, L. H.; JAKOBSEN, M. Gut microbiota in human adults with type 2 diabetes differs from non-diabetic adults. **PloS one**, Public Library of Science, v. 5, n. 2, p. e9085, 2010.

LEE, S.; CHO, S.; KIM, H.; WON, Y. Performance analysis of SSD/HDD hybrid storage manager. In: IEEE. NANO, INFORMATION TECHNOLOGY AND RELIABILITY (NASNIT), 2011 15TH NORTH-EAST ASIA SYMPOSIUM ON. 2011. **Anais...**, [S.l.], 2011. p. 136–139.

LEVENTHAL, A. Flash storage today. **Queue**, ACM, v. 6, n. 4, p. 24–30, 2008.

LI, J.; LI, B. Erasure coding for cloud storage systems: a survey. **Tsinghua Science and Technology**, TUP, v. 18, n. 3, p. 259–272, 2013.

LI, J.; WU, J.; DÁN, G.; ARVIDSSON, Å.; KIHL, M. Performance analysis of local caching replacement policies for internet video streaming services. In: IEEE. SOFTWARE, TELECOMMUNICATIONS AND COMPUTER NETWORKS (SOFTCOM), 2014 22ND INTERNATIONAL CONFERENCE ON. 2014. **Anais...**, [S.l.], 2014. p. 341–348.

LI, S.; CAO, Q.; WAN, S.; QIAN, L.; XIE, C. HRSPC: a hybrid redundancy scheme via exploring computational locality to support fast recovery and high reliability in distributed storage systems. **Journal of Network and Computer Applications**, Elsevier, v. 66, p. 52–63, 2016.

LI, W.; GUO, W.; FRANZINELLI, E. Achieving Dynamic Workload Balancing for P2P Volunteer Computing. In: IEEE. PARALLEL PROCESSING WORKSHOPS (ICPPW), 2015 44TH INTERNATIONAL CONFERENCE ON. 2015. **Anais...**, [S.l.], 2015. p. 240–249.

LIN, B.; LI, S.; LIAO, X.; LIU, X.; ZHANG, J.; JIA, Z. CareDedup: cache-aware deduplication for reading performance optimization in primary storage. In: IEEE. DATA SCIENCE IN CYBERSPACE (DSC), IEEE INTERNATIONAL CONFERENCE ON. 2016. **Anais...**, [S.l.], 2016. p. 1–10.

LINT, J. H. V. **Introduction to coding theory**. [S.l.]: Springer Science & Business Media, 2012. v. 86.

MA, S.; CHEN, H.; SHEN, Y.; LU, H.; WEI, B.; HE, P. Providing hybrid block storage for virtual machines using object-based storage. In: IEEE. 2014 20TH IEEE INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS (ICPADS). 2014. **Anais...**, [S.l.], 2014. p. 150–157.

MA, Y.; NANDAGOPAL, T.; PUTTASWAMY, K. P.; BANERJEE, S. An ensemble of replication and erasure codes for cloud file systems. In: IEEE. INFOCOM, 2013 PROCEEDINGS IEEE. 2013. **Anais...**, [S.l.], 2013. p. 1276–1284.

MAQABLEH, M. M. Fast Parallel Keyed Hash Functions Based on Chaotic Maps (PKHC). In: WESTERN EUROPEAN WORKSHOP ON RESEARCH IN CRYPTOLOGY. 2011. **Anais. . .** , [S.l.: s.n.], 2011. p. 33–40.

MARTALÒ, M.; AMORETTI, M.; PICONE, M.; FERRARI, G. Sporadic decentralized resource maintenance for P2P distributed storage networks. **Journal of Parallel and Distributed Computing**, Elsevier, v. 74, n. 2, p. 2029–2038, 2014.

MEGIDDO, N.; MODHA, D. S. Outperforming LRU with an adaptive replacement cache algorithm. **Computer**, IEEE, v. 37, n. 4, p. 58–65, 2004.

MESNIER, M.; GANGER, G. R.; RIEDEL, E. Object-based storage. **IEEE Communications Magazine**, IEEE, v. 41, n. 8, p. 84–90, 2003.

OGGIER, F.; DATTA, A. **Coding techniques for repairability in networked distributed storage systems**. [S.l.]: Now Publishers Incorporated, 2013.

PAMIES-JUAREZ, L.; OGGIER, F.; DATTA, A. Data insertion and archiving in erasure-coding based large-scale storage systems. In: **Distributed Computing and Internet Technology**. [S.l.]: Springer, 2013. p. 47–68.

PAPAILIOPOULOS, D. S.; DIMAKIS, A. G. Locally repairable codes. **Information Theory, IEEE Transactions on**, IEEE, v. 60, n. 10, p. 5843–5855, 2014.

PARK, G. S.; SONG, H. A novel hybrid P2P and cloud storage system for retrievability and privacy enhancement. **Peer-to-Peer Networking and Applications**, Springer, v. 9, n. 2, p. 299–312, 2016.

PLANK, J. S. The RAID-6 Liberation Codes. In: FAST-2008: 6TH USENIX CONFERENCE ON FILE AND STORAGE TECHNOLOGIES. 2008. **Anais. . .** , [S.l.: s.n.], 2008.

PLANK, J. S.; GREENAN, K. M. **Jerasure: A library in C facilitating erasure coding for storage applications–version 2.0**. [S.l.], 2014.

PURSER, M.; HOUSE, A. **Introduction to error-correcting codes**. [S.l.]: Artech House Boston, 1995.

QIN, A.; HU, D.-M.; LIU, J.; YANG, W.-J.; TAN, D. Fatman: Building Reliable Archival Storage Based on Low-Cost Volunteer Resources. **Journal of Computer Science and Technology**, Springer, v. 30, n. 2, p. 273–282, 2015.

RAIGOZA, J.; SUN, J. Temporal join processing with the adaptive Replacement Cache-Temporal Data policy. In: IEEE. COMPUTER AND INFORMATION SCIENCE (ICIS), 2014 IEEE/ACIS 13TH INTERNATIONAL CONFERENCE ON. 2014. **Anais. . .** , [S.l.], 2014. p. 131–136.

RASHMI, K.; SHAH, N. B.; GU, D.; KUANG, H.; BORTHAKUR, D.; RAMCHANDRAN, K. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In: ACM. ACM SIGCOMM COMPUTER COMMUNICATION REVIEW. 44., 2014. **Anais. . .** , [S.l.], 2014. p. 331–342.

RASHMI, K.; SHAH, N. B.; KUMAR, P. V.; RAMCHANDRAN, K. Explicit construction of optimal exact regenerating codes for distributed storage. In: IEEE. COMMUNICATION, CONTROL, AND COMPUTING, 2009. ALLERTON 2009. 47TH ANNUAL ALLERTON CONFERENCE ON. 2009. **Anais. . .** , [S.l.], 2009. p. 1243–1249.

REED, I. S.; SOLOMON, G. Polynomial codes over certain finite fields. **Journal of the society for industrial and applied mathematics**, SIAM, v. 8, n. 2, p. 300–304, 1960.

RINNEN, P. Hype Cycle for Storage Technologies, 2016. **Gartner. Np**, v. 1, 2016.

RIPEANU, M. Peer-to-peer architecture case study: Gnutella network. In: IEEE. PEER-TO-PEER COMPUTING, 2001. PROCEEDINGS. FIRST INTERNATIONAL CONFERENCE ON. 2001. **Anais. . .** , [S.l.], 2001. p. 99–100.

RIZVI, S. S.; CHUNG, T.-S. Flash SSD vs HDD: High performance oriented modern embedded and multimedia storage systems. In: IEEE. COMPUTER ENGINEERING AND TECHNOLOGY (ICCET), 2010 2ND INTERNATIONAL CONFERENCE ON. 7., 2010. **Anais. . .** , [S.l.], 2010. p. V7–297.

ROSENBAND, D. L.; ROSENBAND, T. A design case study: CPU vs. GPGPU vs. FPGA. In: IEEE. FORMAL METHODS AND MODELS FOR CO-DESIGN, 2009. MEMOCODE'09. 7TH IEEE/ACM INTERNATIONAL CONFERENCE ON. 2009. **Anais. . .** , [S.l.], 2009. p. 69–72.

SAHNI, S.; VAIRAKTARAKIS, G. The master-slave paradigm in parallel computer and industrial settings. **Journal of Global Optimization**, Springer, v. 9, n. 3-4, p. 357–377, 1996.

SATHIAMOORTHY, M.; ASTERIS, M.; PAPAILIOPOULOS, D.; DIMAKIS, A. G.; VADALI, R.; CHEN, S.; BORTHAKUR, D. Xoring elephants: Novel erasure codes for big data. In: VLDB ENDOWMENT. PROCEEDINGS OF THE VLDB ENDOWMENT. 6., 2013. **Anais. . .** , [S.l.], 2013. p. 325–336.

SAXENA, P.; KUMAR, P. Performance evaluation of HDD and SSD on 10GigE, IPoIB & RDMA-IB with Hadoop Cluster Performance Benchmarking System. In: IEEE. CONFLUENCE THE NEXT GENERATION INFORMATION TECHNOLOGY SUMMIT (CONFLUENCE), 2014 5TH INTERNATIONAL CONFERENCE-. 2014. **Anais. . .** , [S.l.], 2014. p. 30–35.

SHAH, N. B.; RASHMI, K.; KUMAR, P. V.; RAMCHANDRAN, K. Interference alignment in regenerating codes for distributed storage: Necessity and code constructions. **Information Theory, IEEE Transactions on**, IEEE, v. 58, n. 4, p. 2134–2158, 2012.

SHVACHKO, K.; KUANG, H.; RADIA, S.; CHANSLER, R. The hadoop distributed file system. In: IEEE. 2010 IEEE 26TH SYMPOSIUM ON MASS STORAGE SYSTEMS AND TECHNOLOGIES (MSST). 2010. **Anais. . .** , [S.l.], 2010. p. 1–10.

SOBE, P. Parallel coding for storage systems—An OpenMP and OpenCL capable framework. In: IEEE. ARCS WORKSHOPS (ARCS), 2012. 2012. **Anais. . .** , [S.l.], 2012. p. 1–6.

STOICA, I.; MORRIS, R.; LIBEN-NOWELL, D.; KARGER, D. R.; KAASHOEK, M. F.; DABEK, F.; BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. **IEEE/ACM Transactions on Networking (TON)**, IEEE Press, v. 11, n. 1, p. 17–32, 2003.

STUDENT. The probable error of a mean. **Biometrika**, JSTOR, p. 1–25, 1908.

SUH, C.; RAMCHANDRAN, K. Exact-repair MDS codes for distributed storage using interference alignment. In: IEEE. INFORMATION THEORY PROCEEDINGS (ISIT), 2010 IEEE INTERNATIONAL SYMPOSIUM ON. 2010. **Anais...**, [S.l.], 2010. p. 161–165.

TANG, Y.; YIN, J.; LO, W.; LI, Y.; DENG, S.; DONG, K.; PU, C. MICS: Mingling Chained Storage Combining Replication and Erasure Coding. In: IEEE. RELIABLE DISTRIBUTED SYSTEMS (SRDS), 2015 IEEE 34TH SYMPOSIUM ON. 2015. **Anais...**, [S.l.], 2015. p. 192–201.

TIAN, C. AC library of repair-efficient erasure codes for distributed data storage systems. In: IEEE. BIG DATA (BIG DATA), 2014 IEEE INTERNATIONAL CONFERENCE ON. 2014. **Anais...**, [S.l.], 2014. p. 21–26.

TZOVARAS, D.; GRAMMALIDIS, N.; STRINTZIS, M. G.; MALASSIOTIS, S. Coding for the storage and communication of visualisations of 3D medical data. **Signal Processing: Image Communication**, Elsevier, v. 13, n. 1, p. 65–87, 1998.

WEATHERSPOON, H.; KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In: **Peer-to-Peer Systems**. [S.l.]: Springer, 2002. p. 328–337.

WESSA, P. Free statistics software, office for research development and education, version 1.1. 23-r7. **URL http://www. wessa.net**, 2016.

YIN, C.; WANG, J.; XIE, C.; WAN, J.; LONG, C.; BI, W. Robot: an efficient model for big data storage systems based on erasure coding. In: IEEE. BIG DATA, 2013 IEEE INTERNATIONAL CONFERENCE ON. 2013. **Anais...**, [S.l.], 2013. p. 163–168.

ZHAO, D.; WANG, K.; QIAO, K.; LI, T.; SADOOGHI, I.; RAICU, I. Toward high-performance key-value stores through GPU encoding and locality-aware encoding. **Journal of Parallel and Distributed Computing**, Elsevier, v. 96, p. 27–37, 2016.