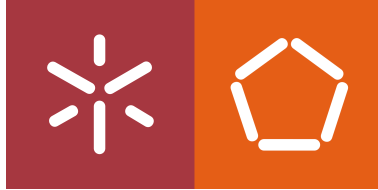
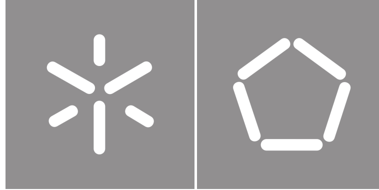


Universidade do Minho
Escola de Engenharia

Alexandre José Branco Ferreira

Armazenamento confiável e em larga escala para aplicações compatíveis com POSIX

Junho de 2023



Universidade do Minho
Escola de Engenharia

Alexandre José Branco Ferreira

Armazenamento confiável e em larga escala para aplicações compatíveis com POSIX

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho efetuado sob a orientação de:
Doutor João Tiago Paulo
Doutor Francisco Almeida Maia

Junho de 2023

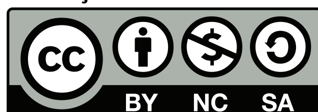
DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositoriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Creative Commons Atribuição-NãoComercial-CompartilhaIgual 4.0 Internacional
CC BY-NC-SA 4.0

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.pt

Agradecimentos

Esta dissertação de mestrado vem pôr um fim à etapa mais difícil, mas mais gratificante vivenciada nestes últimos 5 anos na Universidade do Minho. Esta não seria possível sem um conjunto especial de pessoas que me ajudaram a ultrapassar desafios e contribuíram para conseguir atingir os meus objetivos.

Em primeiro lugar, quero agradecer aos meus orientadores desta dissertação. Ao Doutor João Tiago Paulo, um grande agradecimento por toda a dedicação e ajuda nos momentos mais desafiantes, pela forma pragmática como abordava os problemas com que nos deparamos e pela enorme disponibilidade demonstrada para esclarecer qualquer dúvida. De igual forma, agradeço ao Doutor Francisco Maia pelo tempo despendido e apoio prestado no sentido de encontrar soluções para os mais diversos problemas deste projeto, e ainda por todo o conhecimento transmitido. Quero ainda agradecer a todos os colaboradores do HASLab pela forma como me receberam e pelos diversos momentos de convívio proporcionados.

Um obrigado a todos os meus colegas da universidade, que me acompanharam durante este longo percurso académico. Em especial ao Alexandre Miranda e ao João Azevedo por estarem presentes nos bons e maus momentos desta etapa, pela entreajuda e solidariedade que demonstraram. À minha família, em especial aos meus pais e irmã, o maior agradecimento de todos, por me ajudarem, serem compreensivos nos momentos em que mais precisava e por me darem as ferramentas que me permitiram concluir esta etapa.

Por último, quero expressar a minha gratidão à Fundação para a Ciência e a Tecnologia (FCT), por patrocinarem o desenvolvimento desta dissertação.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Resumo

Armazenamento confiável e em larga escala para aplicações compatíveis com POSIX

A Internet of Things (IoT) é uma das áreas tecnológicas que necessita de sistemas distribuídos que suportem o armazenamento e acesso à enorme quantidade de dados constantemente produzidos por centenas a milhares de dispositivos. Até agora, a maioria dos sistemas desenvolvidos encontravam-se adaptados a instalações em centro de dados, impulsionados pela adoção de serviços de computação em nuvem, porém, o sistema distribuído Large Scale File System (LSFS), veio mudar o paradigma atual e mover o armazenamento distribuído para infraestruturas totalmente descentralizadas. Este é um sistema de ficheiros *peer-to-peer* não estruturado, parcialmente compatível com a interface POSIX, que permite a realização de leituras por parte de múltiplos utilizadores, mas escritas por parte de um só utilizador. Foi construído para atingir alta disponibilidade e resiliência e encontra-se preparado para escalar para infraestruturas do futuro. Todavia, o LSFS não apresenta operações essenciais de um sistema de ficheiros, como a eliminação ou modificação de dados, a grande carga que é exercida sobre a rede tem consequências negativas no sistema como um todo e a forma como este foi avaliado levanta várias questões.

Com o propósito de resolver estes desafios, desenvolveu-se o *improved* Large Scale File System (iLSFS), um sistema de ficheiros que estende o sistema LSFS, dotando-o de uma melhor usabilidade, mas preservando todas as suas características fundamentais como a escalabilidade, a resiliência e a disponibilidade. Para isso, o sistema adota soluções, como *Tombstones*, que viabilizam a eliminação de dados e a disponibilização de uma interface com maior compatibilidade com o standard POSIX, implementa métodos, como *Version Vectors*, que permitem o controlo de concorrência entre múltiplos utilizadores, e mecanismos, como caches, que ajudam a mitigar o problema de saturação da rede.

Os resultados obtidos, demonstram que o iLSFS, com todas as funcionalidades introduzidas, apresenta uma melhor usabilidade sem, no entanto, comprometer significativamente o desempenho. Quando introduzido num caso de estudo real, demonstra-se que o sistema é capaz de escalar para ambientes de larga escala, com centenas de nodos, e mesmo quando submetido a cenários de instabilidade, onde a ocorrência de falhas aleatórias é a norma, o iLSFS mostra-se capaz de tolerar a falha de uma grande quantidade de nodos de armazenamento, sem que esta provoque uma disrupção do seu funcionamento.

Palavras-chave: Sistema de Ficheiros, POSIX, LSFS, Gossip, Escalabilidade, Tolerância a faltas

Abstract

Fault-tolerant and large-scale storage for POSIX compliant applications

The Internet of Things (IoT) is one of the main technological areas that widely needs distributed systems to accommodate the storage and access to large amounts of data, being continuously produced by hundreds to thousands of devices. Until now, most developed systems were adapted to data centers installations, driven by the adoption of the cloud, nonetheless, one system, the Large Scale File System (LSFS), is attempting to change the current model and move the distributed storage to highly decentralized infrastructures. This is an unstructured peer-to-peer file system, partially compatible with the POSIX interface, that allows multi-user reads, but only single-user writes. It was built to achieve high availability and resilience, and was designed to scale for the infrastructures of the future. However, LSFS does not offer key operations of a traditional file system, such as deleting and modifying files, the large load that is exerted over the network has negative consequences on the system, and its evaluation raises various concerns.

In order to solve these challenges, the *improved* Large Scale File System (iLSFS) was developed, an extension of the LSFS system, providing it with better usability, but preserving all the fundamental features, its scalability, resilience and availability. To do so, the system adopts solutions, e.g., Tombstones, which allow the development of the delete operation and the provision of a more complete POSIX interface, implements methods, such as Version vectors, which enable for concurrency control among multiple users in the file system, and mechanisms, namely caches, that help mitigate the network saturation problem.

The experimental results show that iLSFS, with all its features, presents better usability, without significantly compromising performance. When placed in a real case study, it is further demonstrated that the system is capable of scaling to large scale environments, with hundreds of nodes, and that even when subjected to unreliable scenarios, in which the occurrence of random failures is the norm, iLSFS proves to be capable of tolerating the failure of a large number of storage nodes, without causing the disruption of its normal operation.

Keywords: File System, POSIX, LSFS, Gossip, Scalability, Fault Tolerance

Índice

Índice de Figuras **x**

Índice de Tabelas **xi**

Siglas **xiv**

1 Introdução **1**

1.1 Problema 2
1.2 Objetivos 3
1.3 Contribuições 4
1.4 Estrutura da dissertação 4

2 Estado de arte **6**

2.1 Sistemas de gestão de dados distribuídos 6
2.2 Sistemas de ficheiros distribuídos 9
2.3 Large Scale File System 11
2.3.1 Arquitetura 11
2.3.2 Modelo de coerência 15
2.3.3 Organização dos dados e metadados 15
2.3.4 Modelo de acesso a dados 16
2.3.5 Operações e paralelização 17
2.3.6 Construção de grupos e Replicação 18
2.3.7 Balanceamento da carga 21
2.3.8 Recuperação de dados 21
2.4 Discussão 22

3 Arquitetura e Fluxo de pedidos **24**

3.1 Visão geral do sistema 24

3.2 Modelo de acesso a dados 26
3.3 Operações 27
3.3.1 Interface POSIX 27
3.3.2 API do Cliente 27
3.4 Fluxo de Pedidos 29

4 Versionamento, Otimizações e Protótipo 32
4.1 Versionamento 32
4.2 Eliminação de dados 35
4.3 Reconciliação de conflitos 36
4.3.1 Seleção do ficheiro 37
4.3.2 Junção de diretorias 37
4.4 Versão apresentada 39
4.5 Cache 41
4.5.1 Ficheiros abertos 42
4.5.2 Cache de diretorias 43
4.6 Metadados de diretorias 44
4.7 Armazenamento persistente 47
4.8 Anti-Entropia 49
4.9 Protótipo iLSFS 51

5 Avaliação 52
5.1 Desempenho 52
5.1.1 Metodologia 52
5.1.2 Resultados 57
5.1.3 Discussão 74
5.2 Escalabilidade e Resiliência 75
5.2.1 Metodologia 75
5.2.2 Resultados 79
5.2.3 Discussão 85

6 Conclusão 86
6.1 Trabalho Futuro 87

Bibliografia 89

Apêndices

A Apêndice 93

A.1 Utilização de recursos 93
A.1.1 Avaliação de desempenho 93

Índice de Figuras

1 Arquitetura do LSFS . . . 12
2 Organização hierárquica dos blocos no LSFS . . . 16
3 Mecanismo de paralelização no LSFS . . . 18
4 Exemplificação do mecanismo de construção de grupos . . . 19

5 Arquitetura do Cliente iLSFS . . . 25
6 Arquitetura do Nodo DataFlasks . . . 26
7 Fluxo de pedidos no iLSFS . . . 29

8 *Version vectors* com *id-per-client* no iLSFS [1] . . . 35
9 Problema na eliminação de referências de uma diretoria . . . 38
10 Junção de diretorias no iLSFS . . . 39
11 Consulta de última versão a múltiplos nodos . . . 40
12 Cenário de Cache de diretorias sem atualização . . . 44
13 Fluxo de operações responsável pela criação de uma diretoria . . . 46
14 Fluxo de operações na consulta dos metadados de uma diretoria . . . 47
15 Organização de dados no armazenamento do nodo DataFlasks . . . 48
16 Mecanismo de Anti-Entropia no iLSFS . . . 50

17 Ambiente experimental - Cloud Privada . . . 53
18 Ambiente experimental - Cloud GCP . . . 76
19 Treino LeNet - Injeção de 1% de falhas no iLSFS . . . 81
20 Treino LeNet - Injeção de 3% de falhas no iLSFS . . . 82
21 Treino LeNet - Injeção de 5% de falhas no iLSFS . . . 83

Índice de Tabelas

1 Estado de arte dos sistemas de gestão de dados distribuídos 9

2 Resultados Filebench - Escritas sequenciais em diversas configurações 58
3 Resultados Filebench - Leituras sequenciais em diversas configurações 60
4 Resultados Filebench - Leituras aleatórias em diversas configurações 61
5 Resultados Filebench - Criação de ficheiros em diversas configurações 63
6 Resultados Filebench - Eliminação de ficheiros em diversas configurações 63
7 Resultados Filebench - Consulta dos metadados de ficheiros em diversas configurações 65
8 Resultados Filebench - Escrita sequencial no cenário macro 68
9 Resultados Filebench - Leitura sequencial no cenário macro 69
10 Resultados Filebench - Leitura aleatória no cenário macro 70
11 Resultados Filebench - Criação de ficheiros no cenário macro 71
12 Resultados Filebench - Eliminação de ficheiros no cenário macro 71
13 Resultados Filebench - Consulta dos metadados de ficheiros no cenário macro 71
14 Recursos utilizados - Consulta de metadados de ficheiros no cenário macro 72
15 Resultados Filebench - Servidor web no cenário macro - Avaliação ao balanceador de carga 72
16 Recursos utilizados - Servidor web no cenário macro - Avaliação ao balanceador de carga 73
17 Resultados Filebench - Servidor web no cenário macro - Avaliação à anti-entropia 73
18 Treino LeNet - Comparação entre o sistema de ficheiros Local e iLSFS 80
19 Treino LeNet - Recursos consumidos pelo sistema de ficheiros local 80
20 Treino LeNet - Recursos consumidos pelo iLSFS 81
21 Distribuição de *churn* pelos 16 grupos de replicação, em cada iteração 84
22 Treino LeNet - Débito médio do treino no iLSFS durante a injeção de falhas 85
23 Treino LeNet - Recursos consumidos pelo iLSFS durante a injeção de falhas 85

24 Recursos utilizados - Escritas sequenciais em diversas configurações 93
25 Recursos utilizados - Leituras sequenciais em diversas configurações 94
26 Recursos utilizados - Leituras aleatórias em diversas configurações 94

27 Recursos utilizados - Criação de ficheiros em diversas configurações 94
28 Recursos utilizados - Eliminação de ficheiros em diversas configurações 95
29 Recursos utilizados - Consulta de metadados de ficheiros em diversas configurações . . 95
30 Recursos utilizados - Escritas sequenciais no cenário macro 96
31 Recursos utilizados - Leituras sequenciais no cenário macro 97
32 Recursos utilizados - Leituras aleatórias no cenário macro 98
33 Recursos utilizados - Criação de ficheiros no cenário macro 98
34 Recursos utilizados - Eliminação de ficheiros no cenário macro 98
35 Recursos utilizados - Servidor web no cenário macro - Avaliação à anti-entropia 99

Índice de Listagens

3.1 Operações sobre dados 28
3.2 Operações direcionadas à paralelização de dados 28
3.3 Operações direcionadas a metadados 28

Siglas

CRDTs Conflict-free Replicated Data Types

DHTs Distributed Hash Tables

GCP Google Cloud Platform

iLSFS *improved* Large Scale File System

IoT Internet of Things

LRU Least Recently Used

LSFS Large Scale File System

LWW Last-Write-Wins

POSIX Portable Operating System Interface

SPS Serviço de Peer Sampling

TCP Transmission Control Protocol

UDP User Datagram Protocol

1

Introdução

Atualmente tem-se verificado uma crescente utilização de aplicações e dispositivos tecnológicos que se enquadram no conceito de Internet of Things (IoT) [54]. A noção de IoT cresceu com a difusão de dispositivos, que têm como propósito a conexão e a partilha de dados entre sistemas através da internet. Estes dispositivos podem ser encontrados em cidades, em casas ou veículos inteligentes, em diversas áreas da indústria e da medicina para automação ou monitorização e em outros locais e setores da nossa sociedade, o que demonstra a sua enorme aplicabilidade. Não é, assim, de admirar que exista um elevado número de estudos que apresentem a imensa quantidade de dados produzidos pelo espaço IoT. Um exemplo disso é o estudo desenvolvido pela Statista [51], que prevê que durante o ano de 2025 possam ser gerados e consumidos, aproximadamente, 180 Zettabytes (ZB) de dados úteis, mais do dobro das previsões avançadas para o ano de 2021, que atingiam os 79 Zettabytes (ZB) de dados. Estes valores são já representativos de uma enorme quantidade de dados produzida, e a tendência é para um crescimento exponencial ao longo dos próximos anos, acompanhando a evolução e adoção dos dispositivos IoT.

Hoje em dia existem infraestruturas montadas especificamente para o armazenamento dos dados produzidos pelos dispositivos IoT. Prestadores de serviços como a Amazon Web Services [3], ou a IBM [20] dispõem de soluções dedicadas à computação e armazenamento distribuído destes dados na cloud. Todo o ecossistema disponibilizado pela cloud pública, a sua acessibilidade e custos monetários baixos são algumas das razões que levaram ao exponencial crescimento da implementação de aplicações IoT na última década [28]. Porém, todo este avanço para uma adoção da cloud poderá não ser compatível com a evolução do mundo das aplicações IoT [55]. Questões como a privacidade e a segurança são constantemente levantadas, pela falta de controlo nos serviços que são utilizados na cloud e pela confiança necessária que advém de ter os serviços hospedados em centros de dados remotos. Para além disso, com o crescimento espectável do espaço IoT, aproximando-se dos 75,5 biliões de dispositivos para o ano de 2025 [19], será cada vez mais desafiante acomodar, numa única infraestrutura, a quantidade de dados que virá a ser produzida, assim como o envio dos mesmos para a cloud, que representará uma saturação da largura de banda disponibilizada, já muitas vezes restringida pelos prestadores de serviços.

Neste sentido foram desenvolvidos sistemas de armazenamento distribuídos e totalmente descentralizados, desenhados para acomodar a persistência e acesso a dados em infraestruturas de muita larga

escala e lidar com a problemática introduzida pelo crescente mundo que é o espaço IoT, à qual a cloud não conseguirá dar resposta. A construção de sistemas que têm como principal propósito acomodar este tipo de infraestruturas necessita de dispor de uma elevada escalabilidade num ambiente de muita larga escala. Porém, este pressuposto levanta diversos desafios, desde a gestão da entrada e saída de nodos do sistema, e da necessária coordenação entre estes para o armazenamento dos dados, até às recorrentes falhas de componentes ou de comunicações, que originam disrupções no normal funcionamento dos sistemas distribuídos.

Os sistemas de armazenamento recorrem, normalmente, a protocolos estruturados para a gestão e coordenação do armazenamento de dados entre os vários nodos da sua infraestrutura, contudo, encontram-se limitados quando sujeitos a um elevado número de entradas e saídas destes nodos do sistema. Por outro lado, existem outros protocolos, ainda pouco explorados, nomeadamente não estruturados, baseados em protocolos epidémicos, capazes de uma ágil recuperação quando presentes no mesmo cenário.

Em sistemas de ficheiros, o recurso a protocolos não estruturados é apenas abordado pela solução LSFS [47], que foi desenvolvida para acomodar o elevado volume de dados a ser produzidos pelos dispositivos IoT, aproximando o armazenamento da fonte de produção. Todavia, esta solução apresenta vários desafios que esta dissertação pretende colmatar.

1.1 Problema

Quando pensamos num sistema de ficheiros pensamos nas diferentes operações sobre dados que fazem parte do mesmo, como por exemplo, ler e escrever ficheiros, criar diretorias, modificar e apagar ficheiros, entre outras. Mas, quando estamos a falar de sistemas distribuídos descentralizados, que têm dificuldade na aplicação de uma coerência forte, algumas destas operações não têm implementações triviais, por essa mesma razão. De facto, o sistema de ficheiros LSFS implementa apenas parte das operações da interface Portable Operating System Interface (POSIX) [18] supramencionadas. A um utilizador é permitido ler e escrever um ficheiro, bem como ler e escrever diretorias e algumas operações sobre metadados associadas. Contudo, carece da introdução de operações fundamentais, como a modificação ou eliminação de ficheiros, vastamente utilizadas num tradicional sistema de ficheiros e essenciais para algumas aplicações. Sem elas, um sistema de ficheiros, onde só seja permitido inserir objetos, crescerá infinitamente em termos de quota de armazenamento, podendo facilmente tornar-se num problema, se considerarmos que este é um recurso limitando, mesmo sendo fornecido por diversos nodos. Por esta razão, este sistema de armazenamento não se encontra adaptado a uma utilização em ambientes repletos de dados, nem em aplicações que necessitem deste tipo de operações.

Um outro problema associado à natureza da arquitetura deste sistema de ficheiros é a grande carga que é exercida sobre a rede. Este problema resulta da utilização de um protocolo *gossip*, que apesar de necessário por se tratar de um sistema não estruturado, provoca uma inundação da rede com o mais variado tipo de mensagens, seja para a manutenção dos grupos de replicação de dados, ou para a

disseminação das mensagens do cliente pela rede de nodos. Este problema é ainda agravado pela lógica inerente a um sistema de ficheiros, que exige, que por cada operação realizada pelo cliente, possivelmente várias outras operações tenham de vir a ser executadas. Esta lógica apesar de deter um impacto reduzido num sistema local, levanta diversos desafios quando passamos para um ambiente totalmente distribuído. Para além de contribuir para a saturação da rede, por implicar uma maior circulação de pedidos, tem ainda um impacto negativo, para o cliente, no que toca ao tempo de resposta, e, por este motivo, existe a necessidade de encontrar soluções que permitam mitigar este problema.

Outro desafio refere-se à forma como o sistema foi avaliado. Os problemas encontrados dizem, principalmente, respeito aos testes realizados quando o sistema de ficheiros é submetido a um elevado *churn*, ou seja, quando é exposto a um elevado número de entradas e saídas, graciosas ou em falha, de nodos do sistema. As falhas introduzidas foram executadas de uma forma uniforme entre os diferentes grupos de replicação de dados. Evidentemente, num cenário real é esperado que as falhas sejam aleatórias e espalhadas irregularmente pelo sistema de ficheiros e não uniformes por cada grupo de replicação, o que suscita, claramente, diferentes implicações na resiliência e disponibilidade do sistema a ser avaliado.

Para dar resposta às necessidades do espaço IoT e do armazenamento distribuído é preciso que nos sistemas de ficheiros atuais, que recorrem a *gossip*, sejam ultrapassadas as limitações apresentadas, de modo a que possa ser considerado uma alternativa viável, com todos os pressupostos de um tradicional sistema de ficheiros distribuído.

1.2 Objetivos

O objetivo principal desta dissertação consiste em melhorar a usabilidade dos sistemas de armazenamento para muito larga escala. Assim, recorrendo ao LSFS, como sistema base que apresenta várias limitações, pretende-se desenvolver os seguintes tópicos:

Cobertura POSIX Garantir que o sistema suporta uma maior quantidade de operações da interface POSIX, possibilitando que vários clientes possam editar e apagar informação armazenada, alcançando assim uma melhor usabilidade.

Inundação da Rede Mitigar o problema de inundação da rede, de modo a evitar que a quantidade de mensagens propagadas entre os diversos nodos, essenciais para a manutenção da coerência e algoritmos implementados, tenha a menor influência possível na escalabilidade e desempenho do sistema de ficheiros.

Avaliação Completa Realizar uma avaliação ao sistema de ficheiros distribuído, de modo a validar o seu desempenho perante as contribuições efetuadas, bem como avaliar a sua escalabilidade e resiliência, de forma a realçar as vantagens e possíveis limitações que um modelo não estruturado apresenta quando sujeito a níveis de *churn* elevados.

1.3 Contribuições

De modo a atingir os objetivos descritos no ponto anterior, apresentam-se as seguintes contribuições:

- Adaptação e implementação de um **mecanismo de versionamento**, *version vector* [7], de forma a obter uma relação de causalidade entre as atualizações que circulam pelo sistema, e, deste modo, poder efetivamente disponibilizar a nova operação de modificação de ficheiros entre utilizadores. Ainda, e complementando este mecanismo de versionamento, recorre-se a uma abordagem de ***tombstones*** de maneira a guardar quais os ficheiros que se encontram apagados no sistema, possibilitando que a operação perdure e se propague entre as réplicas daquele conteúdo.

- Otimizações, em diversos componentes, para **redução da saturação na rede**. Começando no mecanismo de recuperação de dados, é diminuída a quantidade de dados transmitidos, de forma a que apenas os relevantes sejam anunciados para os nodos da rede. Com o mesmo objetivo é redesenhada a organização da transmissão dos metadados do sistema de ficheiros, evitando que informação repetitiva seja disseminada. Para além disso, é implementada uma cache de metadados capaz de diminuir a quantidade de mensagens solicitadas aos nodos de armazenamento, através do *caching*, dos objetos mais acedidos pelo utilizador no sistema de ficheiros.

- Desenvolvimento de um protótipo de um sistema de ficheiros totalmente descentralizado não estruturado, o **iLSFS**, que faculta uma interface POSIX completa.

- **Avaliação extensiva** ao sistema de ficheiros, onde se inclui uma análise às otimizações efetuadas, de forma a encontrar a configuração do sistema que permite extrair o melhor desempenho possível, e uma comparação entre o protótipo iLSFS e o sistema LSFS, com a finalidade de perceber o impacto das alterações introduzidas no sistema desenvolvido. A avaliação inclui ainda uma validação da escalabilidade e resiliência, através da injeção de falhas, o mais realista possível, durante o treino de uma rede neuronal, recorrendo à *framework* Tensorflow [1] para o efeito.

Com estas contribuições pretende-se que o sistema de ficheiros desenvolvido seja mais competitivo em relação a outros sistemas de armazenamento distribuídos, mas também motivar a evolução do armazenamento distribuído em redes não estruturadas e redes de muita larga escala.

1.4 Estrutura da dissertação

Este documento encontra-se organizado em seis capítulos, sendo o primeiro a Introdução. O Capítulo 2 constitui o enquadramento teórico que orientou o desenvolvimento deste projeto e que contribuiu para a contextualização necessária no âmbito do mesmo. Começa-se por dar a conhecer os tipos de sistemas de gestão de dados distribuídos mais relevantes, bem como as garantias de coerência e disponibilidade características de cada um, mas também as diferentes abordagens e respetiva topologia empregue por

cada sistema. De seguida, é apresentada uma evolução dos sistemas de ficheiros distribuídos, onde são realçadas as especificidades que os distinguem uns dos outros, em geral e do LSFS em particular. Este capítulo finaliza com uma descrição detalhada da arquitetura e respetivos componentes e estratégias utilizadas pelo LSFS, assim como com uma discussão genérica das limitações que o sistema apresenta.

Nos capítulos seguintes - 3 e 4 - descreve-se, de forma detalhada, o protótipo iLSFS desenvolvido, começando no Capítulo 3 por apresentar a arquitetura, as operações disponibilizadas e o fluxo de pedidos no sistema. Segue-se, no Capítulo 4, a descrição das extensões realizadas no iLSFS, destacando as novas funcionalidades e decisões tomadas para atingir os objetivos da dissertação, terminando com as expecificações de implementação do protótipo.

De seguida, no Capítulo 5, é detalhada a avaliação realizada ao sistema iLSFS, dividindo-se em duas componentes em análise, o desempenho do sistema e a sua escalabilidade e resiliência. Em cada uma destas é apresentada a metodologia de testes seguida, bem como as cargas de trabalho a que o sistema é submetido e as configurações definidas nos cenários de teste. Este capítulo só fica completo com a análise e discussão dos resultados obtidos, em ambas as componentes.

Por fim, no Capítulo 6, são apresentadas as conclusões finais, e expressas algumas recomendações que poderão constituir sugestões para trabalho futuro a ser realizado sobre o sistema desenvolvido.

2

Estado de arte

Neste capítulo, organizado em quatro secções, define-se, através de revisão de literatura, o referencial teórico que orientou o desenvolvimento desta dissertação e que contribui para a contextualização necessária no âmbito da mesma. Começa-se por fazer referência, na Secção 2.1, às diferentes abordagens aos sistemas de gestão de dados distribuídos existentes, assim como às garantias de coerência e disponibilidade que os mesmos apresentam. Na Secção 2.2 será dado destaque aos diversos sistemas de ficheiros distribuídos desenvolvidos ao longo dos anos e às distintas implementações tomadas pelos mesmos. De seguida, a Secção 2.3, centra-se na descrição detalhada do sistema de ficheiros LSFS. Por fim, na Secção 2.4, é realizada uma discussão onde são apontados os principais problemas em aberto.

2.1 Sistemas de gestão de dados distribuídos

Ao longo dos anos foram desenvolvidos diversos sistemas de gestão de dados, cada um com objetivos bem definidos, ultrapassando problemas que sistemas mais antigos não conseguiam superar. Com a evolução destes sistemas de armazenamento formaram-se três grandes grupos: os sistemas de base de dados relacionais; os sistemas de base de dados chave-valor e os sistemas de ficheiros.

Estes três grupos de sistemas, apesar de possuírem características distintas, e estarem adaptados a circunstâncias de armazenamento de dados diferentes, passaram pelo mesmo processo de evolução. Inicialmente, apareceram os sistemas de armazenamento centralizados que foram sendo, gradualmente, substituídos por versões distribuídas devido a diversas limitações de escalabilidade, disponibilidade e desempenho, que decorrem do facto de terem os sistemas de armazenamento alojados numa única máquina. Desafios como a falha da máquina ou mesmo a sua sobrecarga começaram a ser experienciados, restringindo a sua aplicabilidade. Esta categoria de sistemas costuma ter limites de crescimento bem definidos, já que só conseguem escalar de uma forma vertical, isto é, aumentando os recursos da única máquina que constitui o sistema de armazenamento. Por outro lado, as soluções distribuídas abordam este problema através de um crescimento horizontal, aumentando o número de máquinas e distribuíndo a carga por estas, o que lhes permite atingir uma alta disponibilidade e resiliência, que se traduz numa capacidade de lidar com enormes quantidades de dados e suportar a carga de milhares de clientes. No

entanto, para chegarmos aos sistemas de armazenamento atuais, com estas características, muitas foram as soluções apresentadas, desde sistemas distribuídos estruturados, que possuem uma topologia bem definida, ou seja, onde há um conhecimento exato de todos os componentes do sistema, até aos não estruturados que são capazes de suportar uma maior escala, mas onde existe apenas um conhecimento parcial.

Um dos sistemas de gestão de dados mais utilizado, no mundo dos sistemas de armazenamento, é o sistema de base de dados relacionais. Este conjunto de sistemas, mais conhecido por base de dados SQL, é muitas vezes, associado a implementações numa só máquina. Os sistemas de armazenamento relacionais têm como uma das principais propriedades a sua forte coerência, isto é, uma transação tem de ser executada, ao mesmo tempo, em todos as máquinas, que possam atender pedidos do cliente, de modo a que todas estas possam ter os mesmos dados, em qualquer altura. Claro que obter esta propriedade num sistema centralizado é trivial, já que é suficiente garantir que a transação ocorreu na única máquina, no entanto, em sistemas distribuídos obter esta propriedade tem outras implicações. Segundo o teorema de CAP [14] não é possível, para um sistema distribuído, ter uma alta disponibilidade e coerência forte na presença de partições na rede. Como em sistemas distribuídos partições na rede são um dado adquirido, devido a possíveis quebras de comunicações, seja por perda ou atraso de mensagens entre as máquinas, os sistemas de gestão de dados têm que escolher entre priorizar a disponibilidade ou a coerência. Em sistemas de base de dados relacionais, o modelo distribuído, através de mecanismos de replicação, é bastante utilizado como uma alternativa à implementação centralizada. Exemplo disso são o PostgreSQL [32] e o MySQL [30], onde são disponibilizados modos de execução distribuída estruturados, em que é sempre mantida uma topologia bem definida. Porém, e tal como enunciado no teorema de CAP, estes sistemas para atingirem uma coerência forte na presença de partições na rede, não só têm de abdicar da disponibilidade, mas também de recorrer a protocolos de coordenação que provocam um impacto negativo na sua escalabilidade.

A disponibilidade e escalabilidade dos sistemas é, em geral, crucial para o bom funcionamento das aplicações de hoje em dia, momentos de indisponibilidade, por mais curtos que sejam, podem traduzir-se em perdas de vendas ou de clientes. Para atingir a disponibilidade e escalabilidade necessárias, outro tipo de sistemas de gestão de dados teve que ser adotado, nomeadamente sistemas de base de dados chave-valor. Este tipo de sistemas de armazenamento, contrariamente aos tradicionais sistemas de base de dados relacionais, relaxam nas garantias de coerência para conseguir atingir alta disponibilidade, na presença de partições da rede. Entre alguns dos sistemas distribuídos chave-valor mais conhecidos estão o PAST [12], o Dynamo [9], o Cassandra [22], o ScyllaDB [43] e o Riak [21], construídos para suportar diversos casos de estudo, mas também alguns menos conhecidos, como o EdgeKV [48], desenvolvido com o objetivo de mover o armazenamento para os *edge devices* das aplicações IoT. Todos estes sistemas possuem em comum a maneira como implementam a distribuição e a procura de dados nas suas redes *peer-to-peer*. O recurso a Distributed Hash Tables (DHTs) como o CAN [37], o Chord [49], o Pastry [39], o Tapestry [56], o Kademlia [26], ou a outras variantes, permite a estes sistemas obter uma distribuição dos dados pelos nodos que constituem a sua rede. Uma DHT dota o sistema de uma estrutura que

permite o mapeamento, a localização e a replicação de dados, representados como chave-valor, pelos diversos nodos, bem como a gestão do dinamismo da rede, reorganizando a distribuição de dados consoante a entrada e saída de nodos. Esta estratégia concede aos sistemas uma elevada disponibilidade, e consequente escalabilidade, por via da implementação de uma coerência eventual, sem a necessidade de recurso a protocolos de coordenação. Isto significa que os dados presentes nos nodos de armazenamento, nem sempre se encontram coerentes entre si, no entanto, eventualmente conseguem convergir para o mesmo estado. A estruturação característica das DHTs, que dota os sistemas de uma incrível resiliência e disponibilidade, obriga a uma manutenção custosa, de modo a preservar o conhecimento da infraestrutura, essencial para o normal funcionamento dos mecanismos presentes nestes sistemas distribuídos. Esta característica, apesar de positiva em diversos aspetos, estabelece-se como um problema, quando o sistema é inserido num ambiente com elevados níveis de *churn*, dado que pode originar procuras demoradas direcionadas a nodos não responsivos, falhas no encaminhamento de pedidos, vistas inconsistentes dos nodos da rede, entre diversos outros problemas. Estas limitações são evidenciadas em vários estudos [38, 23] a diferentes tipos de protocolos e implementações de DHTs.

Os sistemas que recorrem a DHTs para a distribuição e procura dos dados caem dentro da categoria de sistemas distribuídos estruturados, tal como acontece em todos os sistemas distribuídos de base de dados relacionais, uma vez que possuem um conhecimento global de todos os nodos que compõem a sua topologia. Contudo, existe um conjunto de sistemas distribuídos, denominados não estruturados, que implementam protocolos para permitir o funcionamento do sistema, sem que este possua um conhecimento de todos os nodos que fazem parte da sua rede, necessitando apenas de conhecer um pequeno grupo destes. No conjunto de sistemas de armazenamento chave-valor, existe um sistema distribuído que segue esta abordagem de uma topologia não estruturada. O DataFlasks [25], é um sistema de armazenamento chave-valor, baseado em algoritmos epidémicos, que recorre a protocolos *peer-to-peer* não estruturados. Tal como os outros sistemas antes mencionados, também este foi desenvolvido para atingir uma alta disponibilidade, conseguindo, por isso, apenas alcançar uma coerência eventual. O facto de seguir uma abordagem não estruturada confere-lhe características ideais para escalar para redes de muita larga escala, como milhares de nodos, estando apenas limitado pela possível saturação da rede.

A maior parte dos sistemas de gestão de dados relacionais ou chave-valor descritos até aqui, tiveram o seu desenvolvimento assente noutro tipo de sistemas de armazenamento, amplamente conhecido e denominado por sistema de ficheiros. Esta categoria de sistemas de gestão de dados, em particular, constituiu, desde cedo, um dos principais meios de armazenamento de dados, nos primeiros sistemas informáticos. Naturalmente, com a evolução tecnológica, os sistemas de ficheiros distribuídos tornaram-se também objeto de desenvolvimento. A necessidade de ter um sistema de ficheiros distribuído com alta disponibilidade levou ao desenvolvimento de soluções estruturadas, como o HDFS [46], o Ceph [53] ou o MooseFS [27]. Mais recentemente, a necessidade de ter um sistema com outras características, capaz de ser altamente disponível e escalar para infraestruturas de muita larga escala, levou à criação de uma solução não estruturadas, o LSFS, que veio introduzir o mundo das redes não estruturadas, já presente em base de dados chave-valor, aos sistemas de ficheiros.

Assim, conseguimos classificar os sistemas de gestão de dados distribuídos, presentes na literatura, de acordo com a sua topologia de armazenamento, como é apresentado na Tabela 1.

	Sistema de gestão de dados distribuído		
	Sistema de ficheiros	Base de dados Chave-Valor	Base de dados Relacional
Estruturado	HDFS, Ceph, ...	Dynamo, Riak, ...	PostgreSQL, MySQL, ...
Não estruturado	LSFS	DataFlasks	×

Tabela 1: Estado de arte dos sistemas de gestão de dados distribuídos

Os três grupos de sistemas de gestão de dados distribuídos apresentam soluções estruturadas e não estruturas, à exceção do sistema de base de dados relacional, que pela sua natureza apenas apresenta soluções estruturadas.

Um dos sistemas de gestão de dados que melhor cumpre os requisitos para melhorar a usabilidade dos sistemas de armazenamento para muita larga escala, um dos objetivos deste projeto, é o sistema de ficheiros, uma vez que se encontra adaptado à introdução de uma interface de ficheiros completa que permite a integração do sistema com aplicações compatíveis com POSIX. Assim, importa estudar a evolução das diferentes abordagens tomadas nos sistemas de ficheiros distribuídos ao longo dos anos.

2.2 Sistemas de ficheiros distribuídos

Os sistemas de ficheiros constituíram, desde sempre, um importante substrato de armazenamento, para o mais variado formato de dados. Desde a década de 80, quando começaram a aparecer os primeiros sistemas de partilha de ficheiros remotos, que se procura implementar o modelo distribuído nos sistemas de ficheiros. O NFS [41] e o AFS [17] encontram-se entre os primeiros sistemas a oferecer a um utilizador a possibilidade de acesso remoto aos seus ficheiros e diretorias. O NFS, baseado no modelo cliente-servidor, possibilita que vários utilizadores consigam aceder a um sistema de ficheiros partilhado, alojado num único servidor. Por seu lado, o AFS realiza uma repartição do sistema de ficheiros por vários servidores e permite o acesso remoto aos ficheiros através de *workstations*, que efetuam *caching* dos ficheiros mais acedidos pelo utilizador. Posteriormente, descendente do AFS, foi desenvolvido o Coda [42], um sistema de ficheiros com todas as características do AFS, mas que implementa mecanismos de replicação dos dados, mantendo cópias em vários servidores, e que assume a possibilidade de falha na rede, permitindo a execução de operações sobre dados e metadados, mesmo quando desconectado dos servidores, o que dota o sistema de uma acrescida disponibilidade.

Mais tarde, começaram a aparecer as primeiras implementações de DHTs que, naturalmente, vieram a ser adotadas por sistemas de ficheiros que se pretendiam escaláveis. O Ivy [29] e o Pastis [6] são alguns exemplos de sistemas de ficheiros distribuídos que recorrem a DHTs para o mapeamento, a localização e a replicação dos seus dados pela rede *peer-to-peer*. O Ivy é um sistema de ficheiros distribuído que

permite a leitura e a escrita de dados de diversos utilizadores, porém, segue uma abordagem pouco convencional no armazenamento dos seus dados. Este sistema de ficheiros, em vez de ser constituído por blocos de dados e de metadados, é formado por um conjunto de *logs*, onde cada *log* corresponde a um conjunto de operações de um utilizador. Desta forma, qualquer modificação realizada por um utilizador tem de ser adicionada ao seu respetivo *log* que será armazenado na DHT, a DHash [8]. Se o utilizador quiser realizar uma leitura de dados de um determinado ficheiro é necessário consultar todos os *logs*, com referências para esse ficheiro, e construí-lo consoante uma ordem definida. Pelo contrário, o Pastis utiliza o mesmo substrato de armazenamento para guardar tanto os dados como os metadados dos ficheiros que constituem o seu sistema, e recorre a uma estrutura hierárquica, semelhante ao Unix, para representação dos seus blocos de dados. Cada bloco de metadados é guardado como um bloco modificável, com referência para os blocos, imutáveis, de dados de um ficheiro, e a mesma lógica é implementada para diretorias. Para mapear, replicar e localizar os dados de ficheiros na sua rede de nodos, o sistema recorre à DHT PAST.

Mais recentemente, foram desenvolvidos sistemas de ficheiros adaptados às necessidades tecnológicas atuais e que seguem abordagens díspares às das DHTs dos sistemas anteriores. O HDFS, o Ceph, o GFS [13] e o MooseFS são exemplos de sistemas de ficheiros distribuídos estruturados que, maioritariamente, procuram fornecer uma alta disponibilidade, escalabilidade e resiliência. No que toca à estruturação da infraestrutura, a abordagem tomada pelos quatro sistemas de ficheiros é distinta. O HDFS implementa um servidor de metadados centralizado (*Name Node*), responsável por armazenar e gerir todos os metadados dos ficheiros guardados, e várias instâncias de servidores de dados (*Data Nodes*) para distribuição e replicação dos dados dos ficheiros. Toda a gestão e manutenção da replicação e conhecimento de onde se encontram os dados é realizada pelo servidor centralizado, e por isso qualquer contacto para escrita ou leitura tem de transitar por este. Pela sua natureza, o servidor de metadados existente num *cluster* HDFS (v1.0) pode ser considerado um ponto único de falha, uma vez que ocorrendo qualquer indisponibilidade, todo o sistema de ficheiros deixa de funcionar. Por outro lado, temos outras soluções mais resilientes como o GFS e o MooseFS. Estes sistemas também fazem uma divisão física entre os dados e metadados do sistema, possuindo ambos um servidor de metadados centralizado (*Master server*), que age como um coordenador, possuindo toda a informação relevante sobre todos os servidores de dados (*Chunkservers*) da topologia. No entanto, e ao contrário do HDFS, nestes sistemas de ficheiros são implementados mecanismos de replicação dos metadados, através da manutenção de ficheiros de estado e de *backups* de metadados, que possibilitam que outros servidores equivalentes sejam capazes de assumir se o servidor central de metadados falhar. Por seu lado, o Ceph para além de ter uma distribuição e replicação dos dados por várias instâncias, também o faz para os metadados do sistema, o que significa que dispõe de uma arquitetura totalmente distribuída. Neste sistema, tanto os dados como os metadados são guardados e replicados em armazenamento persistente de igual forma, sem que haja máquinas dedicadas ao armazenamento de metadados, como acontece nos outros sistemas, descritos anteriormente. Todavia, o Ceph institui uma camada de servidores de metadados distribuídos, dedicados

ao armazenamento em cache dos metadados dos ficheiros, o que confere ao sistema uma alta disponibilidade. Adicionalmente, é mantido atualizado um mapeamento dos dados com recurso a monitores e a protocolos de consenso distribuído, essencial para a implementação de uma coerência forte, mas que impede o sistema de escalar para redes de muita larga escala.

Para além da infraestrutura que estes sistemas apresentam, também a forma como permitem a interação com o sistema é relevante analisar. O HDFS, o Ceph e o MooseFS têm em comum o facto de exportarem uma interface POSIX, que lhes permite obter a portabilidade do sistema de ficheiros entre sistemas compatíveis com POSIX. No entanto, todos eles relaxam algumas das propriedades da interface POSIX, seja pela ausência de garantias de atomicidade durante a escrita de dados [11], como acontece no sistema de ficheiros Ceph, ou pela carência de determinadas operações, como a modificação de dados no HDFS.

As diversas soluções que foram desenvolvidas ao longo dos anos tentaram sempre inovar e apresentar sistemas capazes de responder às necessidades de armazenamento do mundo tecnológico. Até agora, a maioria dos sistemas de ficheiros desenvolvidos seguiram sempre a abordagem de uma rede bem estruturada, principalmente por via das DHTs, tal como nos sistemas Ivy e Pastis, ou então por via de mecanismos não tão padronizados, como o *cluster map* [53] do Ceph. Contudo, outro sistema de ficheiros foi desenvolvido, o LSFS, tendo diferentes pressupostos no que à sua estruturação diz respeito. Este é um sistema de ficheiros único, que segue uma abordagem de uma topologia não estruturada, capacitando-o com as características ideais para um armazenamento em muita larga escala.

2.3 Large Scale File System

O LSFS [47] constitui o ponto de partida para atingir uma melhor usabilidade nos sistemas de armazenamento de muita larga escala. Para isso, é relevante ter um conhecimento aprofundado do funcionamento deste sistema, de modo a compreender as contribuições desta dissertação. Esta secção será, assim, dedicada à exposição da arquitetura do LSFS, bem como à descrição do funcionamento dos diversos componentes que o integram.

2.3.1 Arquitetura

O LSFS é um sistema de ficheiros distribuído *peer-to-peer*, completamente descentralizado, construído para acomodar redes de muita larga escala. Foi desenvolvido tendo por base o sistema de armazenamento DataFlasks, herdando a sua tolerância a faltas e respetivos algoritmos epidémicos, que o permitem recuperar de elevados níveis de instabilidade no sistema, sem comprometer a sua disponibilidade. O recurso a uma rede não estruturada distingue-o de outros sistemas que tentam abordar o tema do armazenamento distribuído, não só pela falta de uma topologia bem definida, mas também pela flexibilidade e escalabilidade que a sua rede consegue atingir. Esta estruturação só se torna possível com a aplicação de um protocolo *gossip*, utilizado para disseminação de mensagens pelos diversos nodos da rede, o que

permite que, mesmo numa rede não estruturada, seja possível a receção de mensagens por todos os nodos que fazem parte do sistema de ficheiros.

A arquitetura geral do LSFS pode ser subdividida em dois grandes grupos, o Cliente LSFS e o Nodo DataFlasks, tal como apresentado na Figura 1.

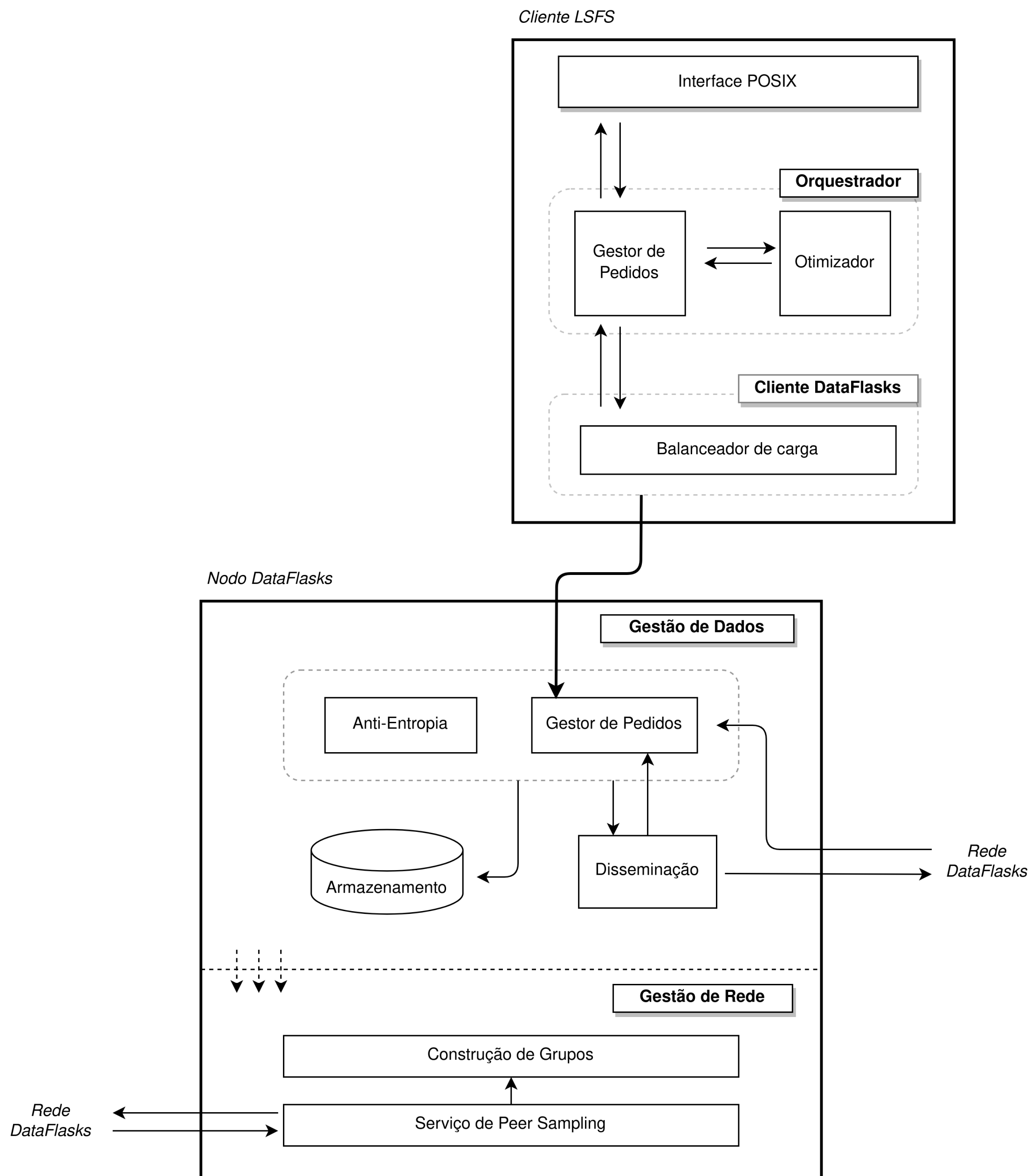


Figura 1: Arquitetura do LSFS

Cliente LSFS

O Cliente LSFS é uma abstração que permite a qualquer utilizador uma interação com o sistema de ficheiros e os seus nodos de armazenamento, possibilitando a execução das operações disponibilizadas pelo

LSFS. Os principais componentes que constituem o Cliente LSFS são a Interface POSIX, o Orquestrador e o Cliente DataFlasks.

A **Interface POSIX** tem como função a exposição de uma interface compatível com POSIX, que favoreça a compatibilidade entre um diverso conjunto de aplicações. Para esse efeito, o LSFS, recorre ao FUSE [24], uma solução *userspace*, para disponibilizar uma API capaz de intercetar as chamadas ao sistema de ficheiros, com o intuito de que estas possam ser traduzidas em operações lógicas conhecidas pelo LSFS e, posteriormente, direcionadas para o Orquestrador para tratamento.

O **Orquestrador** é formado por outros dois componentes, o **Gestor de pedidos** e o **Otimizador**. O primeiro tem como função a associação de cada operação ao respetivo conjunto de procedimentos que devem ser tomados para conseguir obter uma resposta ao pedido do cliente. É também neste componente que é realizada a divisão dos ficheiros em blocos e o respetivo mapeamento dos blocos de dados e metadados em pares chave-valor para serem, posteriormente, armazenados nos nodos DataFlasks. O segundo componente, o Otimizador, efetua *caching* de metadados de ficheiros, e implementa paralelização em operações que assim o permitam, com o objetivo de tornar o fluxo de pedidos de armazenamento mais eficiente.

Por último, o **Cliente DataFlasks** disponibiliza uma API constituída por um conjunto de operações capazes de efetuar a comunicação entre as chamadas do sistema de ficheiros, processadas no Orquestrador, e os nodos de armazenamento DataFlasks, recorrendo ao componente Balanceador de carga para efetivar as comunicações. O **Balanceador de carga** tem um papel preponderante na lógica do sistema de ficheiros, uma vez que está encarregue de fazer a distribuição dos pedidos do cliente pela rede de nodos. O LSFS apresenta duas estratégias diferentes para o balanceador que serão desenvolvidas na Secção 2.3.7 e que têm um grande impacto na latência dos pedidos e na sobrecarga da rede.

Nodos DataFlasks

Os diversos nodos de armazenamento DataFlasks, que integram a rede do sistema de ficheiros, foram desenvolvidos para atuarem de uma forma descentralizada e sem nenhum protocolo de coordenação que sincronize a execução dos algoritmos que os compõem. Esta abordagem permite que todos os nodos sejam iguais entre si, sem haver distinção nos seus componentes nem funções, independentemente de estarem a guardar metadados ou dados dos ficheiros, o que permite que o armazenamento persistente dos dados recebidos do cliente seja realizado através de uma única interface chave-valor, que todos eles disponibilizam.

Deste modo, cada nodo de armazenamento contém componentes dedicados à gestão de dados, que permitem o armazenamento e recuperação de dados e a inicialização dos nodos quando incorporados num novo grupo.

O **Armazenamento** é um componente modular com a função de armazenar os vários pares chave-valor que lhe são atribuídos, e recorre a um base de dados *embedded*, a LevelDB [15], para o efeito.

O componente **Disseminação** tem como principal função a propagação de mensagens pela rede,

que podem ser provenientes de componentes, como por exemplo, a Anti-Entropia. O mecanismo de **Anti-Entropia** garante que aquando de uma entrada de um novo nodo no grupo, este recebe as chaves pelas quais vai ser responsável, e que durante o normal funcionamento do sistema de ficheiros, as chaves que não tenham sido processadas por nodos do grupo possam ser recuperadas, de modo a preservar a coerência do sistema. Por outro lado, as mensagens também podem chegar ao componente Disseminação através do **Gestor de Pedidos**. Este é responsável pelo processamento de pedidos de dados vindo tanto do cliente, como de outros nodos da rede, sendo estes posteriormente enviados para o componente de Armazenamento e/ou Disseminação, consoante as características da mensagem.

Aliados à gestão de dados, existem também os componentes direcionados à gestão da rede *peer-to-peer*, como a Construção de Grupos e o Serviço de Peer Sampling (SPS) que possibilitam a constituição e manutenção da rede de nodos.

A **Construção de Grupos** é um algoritmo que realiza a divisão dos diversos nodos da rede em grupos com tamanho idêntico, onde cada nodo do grupo é responsável pelo armazenamento do mesmo conjunto de chaves e respetivos valores. Este processo permite ao sistema obter características como a resiliência, através da replicação dos dados por um subconjunto de nodos do sistema, o qual designamos por grupos. Para além disso, este mesmo processo dota o sistema de uma elevada disponibilidade, uma vez que se encontra desenhado para que haja, com uma grande probabilidade, um nodo do grupo capaz de responder a um pedido efetuado por um cliente. No entanto, este componente é bem mais complexo do que esta descrição apresenta e é aprofundado na Secção 2.3.6.

O **SPS** tem como função a manutenção do conhecimento que cada nodo tem sobre a rede, através da adição e/ou remoção de nodos à sua vista, à medida que estes entram e saem do sistema. Esta constante atualização é de uma grande importância, pois permite obter um conjunto de nodos com os quais é possível estabelecer uma conexão fiável, facilitando as comunicações entre nodos na rede, seja através de disseminação, ou contacto direto. Para além disso, é através deste componente que cada nodo consegue obter um conhecimento, seja parcial ou na sua totalidade, do conjunto de nodos que são responsáveis por armazenar os mesmos dados que ele, ou seja, que fazem parte do seu grupo de replicação.

Para a grande maioria das comunicações na rede DataFlasks, seja entre os Nodos DataFlasks ou entre o Cliente iLSFS e os nodos de armazenamento, recorre-se ao protocolo User Datagram Protocol (UDP).

Bootstrapper

Para além dos dois grandes grupos de componentes que compõe o LSFS, existe outro, que apesar de mais simplista na sua complexidade, detém uma grande importância no normal funcionamento do sistema, nomeadamente aquando da entrada de nodos neste.

Uma das características mais relevantes neste sistema é a sua adaptação à elevada probabilidade de falha dos nodos de armazenamento DataFlasks, o que pode originar constantes saídas, graciosas ou

não, destes componentes do sistema, e à entrada de novos, ou até à sua substituição, para colmatar estas saídas. Adicionalmente, cada nodo, ao entrar no sistema, necessita de poder comunicar com os restantes nodos da rede, de modo a poder iniciar os seus mecanismos, e participar efetivamente como parte integral do armazenamento de dados. Para isso, é necessário a obtenção de um conhecimento sobre um conjunto de nodos com os quais se pode estabelecer comunicação, que neste sistema é possível através do Bootstrapper.

O Bootstrapper é um componente que tem como função fornecer a novos participantes no sistema um conjunto de nodos da rede com os quais é possível estabelecer uma linha de comunicação. Para isso, cada novo nodo inicia o seu ciclo de vida contactando o Bootstrapper para obter informações sobre os restantes nodos da rede, e por sua vez, o Bootstrapper armazena informação sobre o novo nodo, para que possa ser fornecida sempre que requerida por um outro nodo, acabado de entrar.

2.3.2 Modelo de coerência

O LSFS é um sistema de ficheiros dedicado a infraestruturas de muita larga escala, baseado num sistema assíncrono onde não se podem fazer quaisquer assunções relativamente ao tempo que uma mensagem pode demorar a chegar ao seu destino, nem ao tempo que determinado algoritmo demora a executar. Aliado a isto, o sistema distancia-se dos protocolos de coordenação que limitam a escalabilidade de infraestruturas. Com esta abordagem, este sistema consegue apenas atingir uma coerência eventual, o que significa que em determinados momentos os nodos do sistema, responsáveis por armazenar as mesmas chaves poderão encontrar-se incoerentes entre si, seja por possíveis atrasos nos envios de mensagens, ou até, por perda das mesmas. Porém, e através de mecanismos adicionais implementados, nomeadamente o componente de Anti-Entropia, eventualmente, é possível convergir para um estado coerente em todos os nodos do grupo de replicação.

2.3.3 Organização dos dados e metadados

Um sistema de ficheiros distribuído tradicional realiza uma separação explicita entre as operações que ocorrem sobre dados, das que ocorrem sobre metadados. Do mesmo modo, também esta separação ocorre no LSFS.

Um ficheiro é formado por blocos de dados, denominados de *dblocks*, que contêm o conteúdo do ficheiro distribuído por vários *dblocks*, e por um bloco de metadados, denominado de *iblock*, que contém informação sobre o ficheiro como permissões, tamanho, entre outras. (Semelhante ao *inode* do sistema de ficheiros Unix).

Para a representação das diretorias, o LSFS segue uma organização hierárquica. Cada diretoria é representada como um *iblock* com os seus metadados e referências para *iblocks* relativos a diretorias ou ficheiros, hierarquicamente abaixo.

Com esta organização é possível definir as chaves que representam cada bloco, tal como ilustrado na Figura 2.

- *iblock* representado pelo seu *path* completo, por exemplo */home* .
- *dblock* pela junção do *path* completo com o *offset* do bloco, por exemplo */home/file.txt:4096*.

É também de realçar que cada um dos blocos de dados (*dblocks*) de um ficheiro tem um tamanho fixo, definido por defeito como sendo de 4 KiB (4096 Bytes).

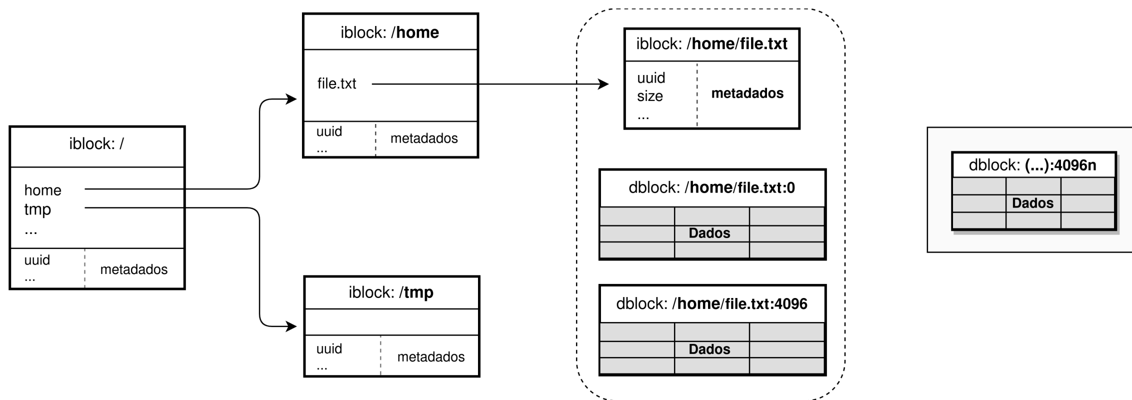


Figura 2: Organização hierárquica dos blocos no LSFS

2.3.4 Modelo de acesso a dados

Para além do modelo de coerência adotado pelo sistema, o LSFS também definiu um modelo de acesso a dados que permite caracterizar como um utilizador pode interagir com os dados do sistema de ficheiros. O padrão de acesso aos dados disponibilizado segue um formato *Write Once Read Many*. Por outras palavras, o acesso aos dados neste formato significa que um cliente pode escrever um determinado ficheiro, e depois, da escrita acabar, este não mais poderá ser modificado. No caso de alguma alteração ser necessária, todo o ficheiro terá de ser reescrito com uma versão superior. Por outro lado, no que à leitura deste diz respeito, não existem quaisquer restrições, podendo cada ficheiro ser lido inúmeras vezes de forma sequencial ou aleatória.

Para controlar o modelo de acesso a dados, tanto a nível dos ficheiros como das diretorias, o LSFS utiliza o conceito de versionamento, associado a uma chave. Porém, o versionamento implementado não introduz uma relação de causalidade entre utilizadores, mas apenas para o utilizador singular, representando uma linha de operações do utilizador em questão.

Perante isto, o sistema estipula algumas políticas que descrevem como este lida com um cenário de múltiplos utilizadores em simultâneo.

1. Os utilizadores do sistema de ficheiros operam sobre ficheiros diferentes, descartando possíveis conflitos que poderiam ser originados por atualizações concorrentes a nível dos ficheiros do sistema.

2. Os utilizadores podem criar ficheiros e/ou diretorias sobre a mesma diretoria, com conflitos a ser passíveis de resolução.

Tal como descrito no primeiro ponto, é assumida a inexistência de colisões entre ficheiros concorrentes, na presença de múltiplos utilizadores, logo, resta perceber como o LSFS trata possíveis conflitos de diretorias.

Junção de diretorias

Tendo em conta a característica de coerência eventual e o modelo de acesso a dados descrito, o LSFS disponibiliza um método de juntar diretorias capaz de lidar com conflitos originados pela ocorrência de versões concorrentes. Para isso, o sistema oferece uma função de *merge* que permite que diferentes versões da mesma diretoria possam ser juntas, tendo como resultado final o conjunto completo da união dos dados das respetivas versões, sem perdas de dados.

De uma forma simplificada, consideremos duas versões, *v1{a,b}* e *v2{c}*, relativas à mesma diretoria, mas inseridas por utilizadores diferentes, onde *v1* tem referência para o ficheiro *a* e *b*, e *v2* para o ficheiro *c*. No caso de não existir resolução de conflitos nas diretorias, apenas uma versão poderia ser apresentada ao utilizador, o que resultaria na perda de dados, relativos à versão descartada. Por esta razão, através da função de *merge* é realizada uma união entre estas duas versões para uma superior, *v3{a,b,c}*, que passa a conter os ficheiros *a*, *b* e *c*.

2.3.5 Operações e paralelização

Em qualquer sistema de armazenamento é relevante perceber as operações e os respetivos mecanismos utilizados para o normal funcionamento do sistema. Como já foi brevemente apresentado no esquema geral do LSFS, Secção 2.3.1, existem dois componentes distintos responsáveis pelas operações disponibilizadas, a Interface POSIX e a API do Cliente.

A Interface POSIX expõe um conjunto de operações responsáveis pela interação com o sistema de ficheiros, onde se destacam a leitura (chamada ao sistema *read*) e escrita (chamada ao sistema *write*) de ficheiros. Nestas operações, para além de ser empregue a sua lógica inerente, isto é, a divisão em blocos de dados e respetivo processamento, é também introduzido um **mecanismo de paralelização**. Este mecanismo presente no componente Otimizador, está encarregue de agrupar os blocos de tamanho 4 KiB em grupos de tamanho *n*, consoante um valor de paralelização pré-definido, para que possam ser enviados, paralelamente, para a camada de armazenamento.

Na Figura 3 é possível visualizar como este mecanismo funciona.

Suponhamos que o componente Orquestrador recebe um bloco de tamanho 32 KiB para ser escrito no sistema de ficheiros. O primeiro passo é dividir o bloco recebido em blocos mais pequenos de tamanho 4 KiB, correspondente ao tamanho fixo máximo dos blocos de dados armazenados no LSFS. Deste processo resulta um conjunto de 8 blocos cada um com tamanho 4 KiB. De seguida, estes blocos são agrupados

até que a soma total do tamanho dos blocos do grupo atinja o valor de paralelização definido. Desta forma obtemos dois grupos distintos, A e B, cada um com 4 blocos, perfazendo os 16 KiB de paralelização em cada grupo. Por fim, os 4 blocos do grupo A são enviados paralelamente para os nodos de armazenamento DataFlasks, através da API do cliente, e quando for obtida a resposta a todos esses blocos, o processo repete-se para o grupo B.

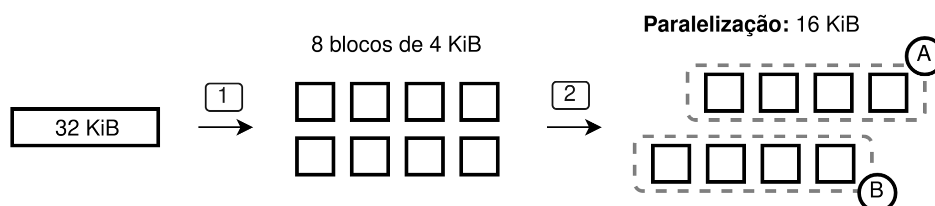


Figura 3: Mecanismo de paralelização no LSFS

2.3.6 Construção de grupos e Replicação

A construção de grupos é um componente fundamental para o funcionamento do LSFS. Os nodos do sistema têm que se dividir em grupos, para que seja possível uma distribuição dos dados e respetiva replicação nos nodos. Esta estratégia é implementada de modo que a divisão ocorra de uma forma automática, consoante um intervalo de valores (mínimo e máximo) definido, correspondente ao tamanho pretendido para cada grupo, ou seja, ao fator de replicação dos dados. Para o normal funcionamento da construção de grupos, cada nodo possui uma vista de grupo, uma estimativa do número de grupos no sistema e uma posição num espaço de endereçamento entre]0, 1].

A **vista de grupo** corresponde a um conjunto de nodos do seu grupo, geridos pelo mecanismo de SPS, onde, para além de ser armazenada informação sobre os nodos, é mantido um valor de idade, que permite identificar se um determinado elemento saiu do sistema.

A **estimativa do número de grupos** é calculada com a informação recolhida pelo SPS, e vai sendo atualizada ao longo do tempo de vida do nodo, convergindo para o valor, aproximado, de grupos no sistema.

A **posição** que o nodo tem **no espaço de endereçamento** é atribuída ao iniciar o nodo e não é mais alterada enquanto este se encontrar vivo. Através desta posição é possível calcular a que grupo o nodo pertence, dividindo o espaço de endereçamento pela estimativa de grupos[1].

Mais concretamente, uma função de *hash* mapeia as chaves que podem ir de]0, *hash_max_ value*], num intervalo fixo de chaves entre]0, 1], tal como o intervalo de valores do espaço de endereçamento. Seguindo o exemplo presente na Figura 4, é possível visualizar esta divisão do espaço de endereçamento, após a entrada do nodo *n4*, quando o sistema passa a apresentar dois grupos. Nesse momento, todos os nodos evidenciam uma estimativa de grupo de valor dois, o que corresponde exatamente ao número de grupos presente no sistema. Como tal, o intervalo de chaves é partido em metade,]0, 0.5]-]0.5, 1],

[1]Saber a que grupo o nodo pertence é relevante, pois possibilita a atribuição das chaves que correspondem ao seu grupo.

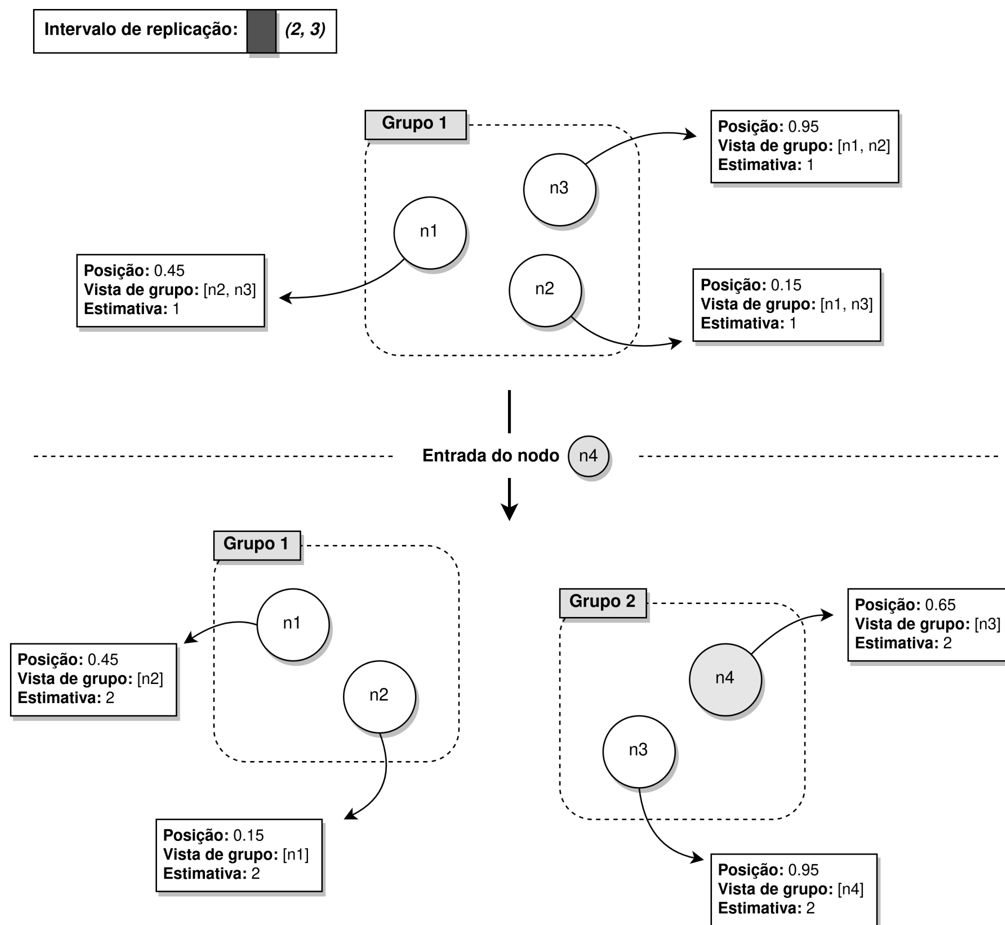


Figura 4: Exemplificação do mecanismo de construção de grupos

e a posição que o nodo tem no espaço de endereçamento dita em que metade, e em que grupo, o nodo se encontra. Assim, o nodo *n1* com posição 0.45, permite-nos assumir que este faz parte do primeiro grupo, uma vez que 0.45 ∈]0, 0.5], mas também que está responsável por armazenar a primeira metade do conjunto total de chaves. Da mesma forma, o nodo *n3*, com posição 0.95, encontra-se no segundo grupo, 0.95 ∈]0.5, 1], e está responsável por armazenar a segunda metada das chaves.

Contudo, para que este processo funcione, o nodo necessita de tomar um conjunto de passos para atingir uma estimativa de grupos estável, essencial para um cálculo correto das chaves a armazenar. Quando um nodo entra no LSFS, este não tem conhecimento de mais nenhum outro nodo, e age como sendo o único grupo no sistema, que ocupa todo o espaço de endereçamento. À medida que este vai estendendo o seu conhecimento da rede, necessita de verificar se o número de elementos do grupo se mantém dentro do intervalo de replicação pré-definido. Como o nodo acredita fazer parte do único grupo do sistema, qualquer nodo descoberto vai aumentar, rapidamente, o número de nodos do grupo, e fazer exceder o valor máximo de replicação. Nesse momento, o nodo é obrigado a atualizar a sua estimativa de número de grupos, dobrando [2] este valor e ao mesmo tempo dividindo o seu espaço de endereçamento,

[2] O número de grupos existentes em cada momento no sistema segue uma distribuição exponencial de base dois ($2^n \rightarrow$ 1, 2, 4, 8, 16, 32, ...).

como explicado anteriormente. A partir daqui o nodo faz parte de um novo grupo, e armazena um conjunto de chaves mais restrito. Da mesma forma, também quando é detetada uma saída do grupo de replicação, isto é, quando o nodo da vista atinge a idade máxima, tem de se repetir este processo, podendo originar uma junção de grupos, caso o intervalo mínimo de replicação seja ultrapassado.

De forma a exemplificar o processo descrito, consideremos, mais uma vez, o exemplo da Figura 4, no qual, num primeiro momento, existem três nodos no sistema, e onde todos eles têm como conhecimento base: a sua posição no espaço de endereçamento e o intervalo de replicação. Adicionalmente, cada um deles vai adquirindo conhecimento, através do mecanismo de SPS, sobre os restantes elementos da rede, no qual se inclui os nodos que fazem parte do seu grupo, permitindo-lhe chegar a uma vista de grupo estável, e por sua vez a uma estimativa correta. Num determinado momento, um outro nodo, *n4*, entra no sistema, e a sua presença vem alterar a disposição de grupos. Ao se aperceberem da sua presença, os restantes nodos que pertenciam ao mesmo grupo identificam que esta nova adição viola o intervalo de replicação definido, logo, existe a necessidade de dobrar o número de grupos para ir ao encontro da replicação desejada. Desta forma, esses três nodos (*n1*, *n2*, *n3*), atualizam a sua estimativa de grupos e, como explicado anteriormente, ficam responsáveis por armazenar um conjunto de chaves mais reduzido. Já o novo nodo, *n4*, à medida que vai conhecendo mais elementos da rede, passa também por este processo. No início, e ao ter conhecimento de apenas mais dois nodos, ainda acredita fazer parte do único grupo no sistema, uma vez que o seu conhecimento é limitado. No entanto, ao identificar um terceiro nodo "do seu grupo", percebe que excede o intervalo máximo de replicação e que a sua estimativa de grupos se encontra incorreta.

Em cada nodo, este processo ocorre sempre que existe uma reorganização dos grupos, devido a entradas ou saídas de outros nodos da rede que deixam os grupos desequilibrados, ou seja, fora do intervalo do fator de replicação definido. Isto provoca o particionamento e junções de grupos e consequentes mudanças no conjunto de chaves a serem guardados pelos nodos.

No entanto, numa rede já madura e estável, um nodo que entre no sistema encontra-se rodeado por outros que possuem uma vista mais abrangente do que a dele. Como tal, e em vez de passar pelo processo iterativo de conhecimento da rede e dos nodos do seu grupo, este lança uma mensagem a pedir, a nodos que se identifiquem como sendo do seu grupo, que lhe transfiram o seu estado. O nodo tem que disseminar esta mensagem pela rede, pois não consegue saber quais os elementos do grupo ao qual ele pertence, já que não possui uma estimativa correta do número de grupos existentes. Através desta transferência de estado, o nodo consegue ter uma vista de grupo maior e conhecer, de uma forma mais rápida, a estimativa do número de grupos do sistema, e assim convergir para um estado final.

Em DataFlasks [25] encontra-se uma análise mais aprofundada, bem como a prova de correção do algoritmo descrito anteriormente, e que permite a divisão do sistema em grupos balanceados.

2.3.7 Balanceamento da carga

Um dos aspetos mais importantes no LSFS é o seu balanceador de carga que permite a disseminação dos pedidos, efetuados pelos clientes, pelos diversos nodos de armazenamento. O LSFS toma duas abordagens distintas no que diz respeito ao balanceador, um *Random Load Balancer* e um *Smart Load Balancer*, que se traduz num impacto diferente no desempenho do sistema.

O ***Random Load Balancer***, tal como o nome sugere, realiza uma seleção aleatória sob um conjunto muito reduzido de nodos, obtido através da componente SPS. O balanceador quando recebe um pedido de um cliente, envia-o para *n* dos nodos que selecionou aleatoriamente, porém, se este mesmo nodo não conseguir processar o pedido, terá de o enviar para um outro até que chegue, eventualmente, ao nodo certo, ou então, que seja atingido um *timeout*, que reiniciará todo o processo no balanceador. Esta estratégia, apesar de ser bastante simples e escalável, não é a mais eficiente.

O ***Smart Load Balancer***, foi construído como uma alternativa mais inteligente ao balanceador aleatório. Este tenta realizar uma aprendizagem contínua durante o funcionamento do sistema, guardando um conjunto de nodos agrupados por grupo de replicação. Com esta abordagem torna-se possível estabelecer um mapeamento entre as chaves que cada nodo do grupo armazena e o pedido que o cliente está a solicitar, o que resulta numa ligação direta com um nodo que satisfará o pedido, e numa latência reduzida para o cliente. Contudo, nem sempre haverá uma correspondência exata entre a chave procurada pelo cliente e o conjunto de nodos aprendidos pelo balanceador. Neste caso, é selecionado um nodo aleatório de entre os conhecidos, tal como acontece no *Random Load Balancer*.

Para além do armazenamento deste conjunto de informação sobre os nodos, este balanceador também precisa de gerir possíveis reorganizações dos grupos no sistema, já que implica uma mudança no conjunto de chaves que os nodos armazenam. Para dar resposta a este problema é executada uma instância da construção de grupos no componente de balanceamento, de modo que os nodos conhecidos se encontrem sempre representativos da rede do sistema. Esta estratégia implica também que se consiga lidar com saídas imprevistas de nodos do sistema, de modo a evitar o envio de pedidos para nodos que já não possam satisfazer o pedido do cliente. Para isso, é adicionado um "fator idade", que atribui a cada nodo conhecido uma marca temporal consoante a sua atividade com o cliente, permitindo optar por nodos contactados mais recentemente e eliminar nodos que ultrapassem uma idade máxima.

2.3.8 Recuperação de dados

Como já foi descrito, o LSFS é um sistema que se encontra, por vezes, incoerente e por este motivo tem a necessidade de implementar mecanismos que garantam a convergência do sistema para um estado coerente. Para isso, o sistema de ficheiros utiliza um mecanismo de recuperação de dados que se encontra totalmente integrado no componente Anti-Entropia.

O mecanismo de recuperação de dados do LSFS, atua em dois momentos:

- O primeiro é referente a uma nova entrada no sistema de um nodo - **Fase de recuperação**.

Quando tal acontece, o nodo inicia o seu ciclo de vida sem incluir o armazenamento local de quaisquer dados, o que o impede de responder corretamente a pedidos recebidos, tanto de clientes como de outros nodos. Perante este impedimento, o nodo tenta fazer uma requisição aos elementos do seu grupo de replicação das suas base de dados. Quando é obtida uma resposta positiva, é aberta uma conexão Transmission Control Protocol (TCP) entre os dois nodos e iniciada a transferência das chaves armazenadas para o novo nodo, até que sejam todas incorporadas. A partir deste momento o nodo pode processar mensagens do cliente, passando para uma fase operacional e estável.

- O segundo momento diz respeito a incoerências originadas pela natureza do sistema - **Fase de operação**. Quando um cliente realiza uma operação, esta nem sempre chega a todos os elementos do grupo de replicação correspondente à chave. Isto pode acontecer devido a falhas na rede, originadas pela natureza do protocolo de transporte UDP, predominantemente utilizado para as comunicações, ou então pela indisponibilidade do nodo para o tratamento dos pedidos. Perante isto existe uma necessidade de eventualmente atingir uma coerência entre os nodos dos diversos grupos de replicação. Para alcançar esta coerência, o nodo DataFlasks segue uma estratégia de anunciar, em cada momento, um "subconjunto contíguo de chaves" [47], para que nodos que não as possuam sejam capazes de as recuperar.

2.4 Discussão

Os sistemas de gestão de dados distribuídos, ocuparam, desde cedo, um papel preponderante no desenvolvimento de plataformas robustas, onde o armazenamento de dados remoto representa uma necessidade. As diferentes características oferecidas por estes sistemas de gestão de dados permitem que se encontrem adaptados a diferentes ambientes de utilização, conforme a interface disponibilizada ou as garantias de coerência, disponibilidade e escalabilidade manifestadas. Num cenário de muita larga escala, tal como o que se encontra em estudo, a escalabilidade e respetiva disponibilidade oferecidas pelos sistemas de armazenamento são dois pontos fundamentais que devem ser priorizados. Em sistemas estruturados, baseados em DHTs, tais como as base de dados chave-valor, Dynamo ou Cassandra, e os sistemas de ficheiros Ivy ou Pastis, a obtenção de uma elevada escalabilidade encontra-se limitada pela capacidade de adaptação a elevados níveis de agitação do sistema, regularmente presente em redes de muita larga escala. Este problema persiste quando exploramos outras implementações estruturadas mais modernas e amplamente utilizadas, nomeadamente em sistemas de ficheiros, como é o caso do Ceph ou do HDFS. O recurso a protocolos de coordenação, em favorecimento de uma coerência forte, e o uso de componentes centralizados são características que restringem a escalabilidade dos sistemas distribuídos. Por outro lado, existe outra categoria de sistemas distribuídos, baseados em protocolos *peer-to-peer* não estruturados, capazes de agilmente recuperar de grandes instabilidades na rede. O sistema de armazenamento chave-valor DataFlasks e o sistema de ficheiros LSFS recorrem a protocolos epidémicos para

conseguirem alcançar elevados níveis de escalabilidade, sem comprometer o normal funcionamento do sistema, o que os dota de características fundamentais para suportar uma infraestrutura de muita larga escala, como as existentes no espaço IoT.

Em especial, o LSFS introduz o modelo distribuído não estruturado nos sistemas de ficheiros, o que permite a construção de um sistema altamente escalável e disponível, exportando, ao mesmo tempo, uma interface compatível com POSIX. No entanto, a solução desenvolvida ainda apresenta diversas limitações que o torna inadequado para uma adoção num ambiente para o qual foi proposto. A falta de determinadas operações, como a modificação e eliminação de ficheiros, não só restringe a sua utilização, como também diminui a sua usabilidade. As estratégias implementas nos diversos componentes não são suficientes para evitar a grande carga exercida sobre a rede, restringindo o seu desempenho. Da mesma forma, também a avaliação da solução não é a mais correta, já que se apoia em testes pouco realistas que levam a diferentes implicações na resiliência e disponibilidade do sistema de ficheiros.

Assim sendo, esta dissertação tem como principal objetivo ultrapassar as limitações que este sistema apresenta, de modo a conceber um sistema com uma melhor usabilidade.

3

Arquitetura e Fluxo de pedidos

A arquitetura do LSFS, já apresentada na Capítulo 2, é a base com que iremos trabalhar, e a partir daí efetuar as alterações necessárias para atingir os objetivos pretendidos. Deste modo, à arquitetura original do LSFS, serão adicionados e alterados componentes, sendo que a maioria das transformações não são realizadas a nível arquitetural, mas sim ao nível dos algoritmos e mecanismos utilizados pelo sistema de ficheiros. Assim, estende-se o sistema de ficheiros, dando origem ao iLSFS.

3.1 Visão geral do sistema

O modelo não estruturado, totalmente descentralizado, é uma das características que torna este sistema de ficheiro único e que permite que escale para redes de muita larga escala. Com este intuito, o sistema encontra-se dividido em dois grandes componentes, o Cliente iLSFS e o Nodo DataFlasks. É possível visualizar a arquitetura do sistema nas Figuras 5 e 6, onde são realçados, através de uma cor sombreada as modificações realizadas. A uma cor cinza-escuro são realçados os componentes que foram adicionados, nomeadamente a Cache no Cliente iLSFS e a Tombstone no Nodo DataFlasks. Por outro lado, diversos outros componentes sofreram modificações profundas, em relação ao sistema LSFS original, como a Interface POSIX, o Gestor de Pedidos e a API do Cliente no Cliente iLSFS, e a Anti-Entropia, o Gestor de Pedidos e o Armazenamento no Nodo DataFlasks.

Cliente iLSFS

O Cliente iLSFS mantém a sua estrutura base original, dividindo-se na Interface POSIX, no Orquestrador e no Cliente DataFlasks. Estes grupos de componentes são responsáveis por viabilizar a interação entre o utilizador do sistema de ficheiros e os nodos DataFlasks.

A Interface POSIX para além de expor todas as operações outrora disponibilizadas, expõe agora a nova operação de eliminação de ficheiros (*unlink*) e diretorias (*rmdir*).

O Orquestrador permanece representado pelo Gestor de pedidos e o Otimizador, estando estes dois

componentes interligados na implementação da lógica para a obtenção das respostas aos pedidos do utilizador. É no Gestor de Pedidos que é efetuada a ponte entre as chamadas realizadas à Interface POSIX e as operações da API do Cliente. Durante este processo, sempre que necessário, é contactado o Otimizador, seja para a paralelização dos blocos para escrita e leitura, ou para evitar a repetição de pedidos à API do Cliente, recorrendo a uma Cache para o efeito. A **Cache** disponibiliza de forma temporária, o conjunto de metadados de diretorias mais acedidos pelo Cliente, mas também os metadados dos ficheiros que se encontram momentaneamente em utilização. Este armazenamento, em memória efémera, permite ao Cliente iLSFS acelerar o processo de consultas efetuadas pela maioria das operações que manipulem metadados.

No que diz respeito ao Cliente DataFlasks, este é composto por dois componentes, a API do Cliente e o Balanceador de carga. A API do Cliente, tal como o nome indica, faculta uma API, propícia à interação com os nodos DataFlasks. No iLSFS, esta API é estendida de forma a incluir as novas operações e interações com a camada de armazenamento. Cada uma das operações disponibilizadas efetua uma gestão do versionamento associado ao cliente, carimbando os dados com um contexto que permite a manutenção da ordem causal das atualizações no sistema de ficheiros. Já o Balanceador de carga, mantém-se, tal como foi explicado na Secção 2.3.7, encaminhando as mensagens para determinados nodos, consoante o tipo de balanceador utilizado.

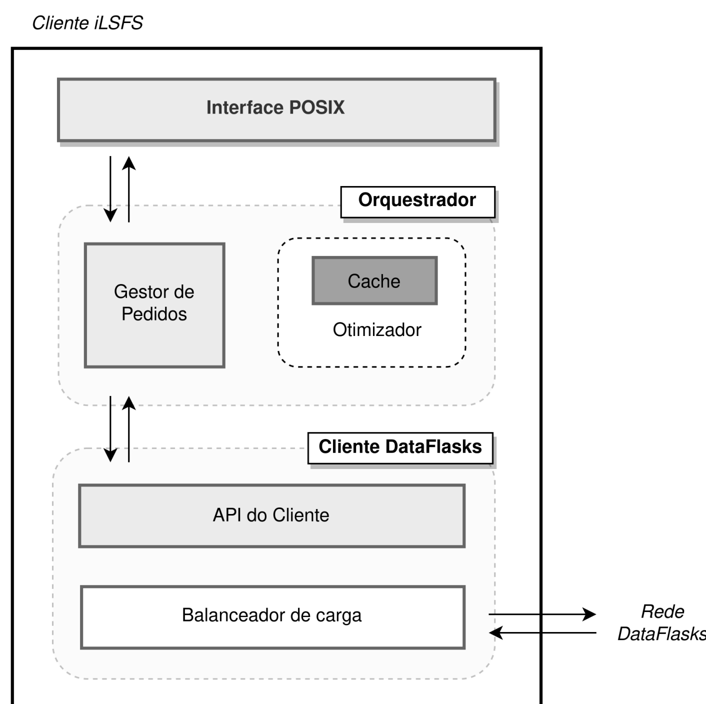


Figura 5: Arquitetura do Cliente iLSFS

Nodo DataFlasks

O Nodo DataFlasks preserva a sua estrutura de componentes, inicialmente desenvolvidos no sistema DataFlasks, desde os que têm o papel de gestão dos dados até aos responsáveis pela gestão da rede. Dentro dos componentes dedicados à gestão de dados, apenas um sofreu alterações significativas na sua arquitetura, o componente Armazenamento. A este foi adicionada uma nova base de dados, denominada de **Tombstone**, responsável por guardar todos os dados eliminados no sistema de ficheiros, de modo a que estes possam ser propagados entre os diversos nodos da rede. Já relativamente à base de dados principal, a estratégia de armazenamento utilizada foi totalmente redesenhada, de forma a permitir a realização de procuras mais eficientes de acordo com as estratégias adotadas no sistema.

No que toca aos restantes componentes, efetuaram-se diversas modificações nos mecanismos que estes utilizam, seja para lidar com as novas operações introduzidas, ou para acomodar a propagação dos dados que se encontram eliminados. Em relação a estas alterações, uma análise mais aprofundada é realizada nas secções subsequentes.

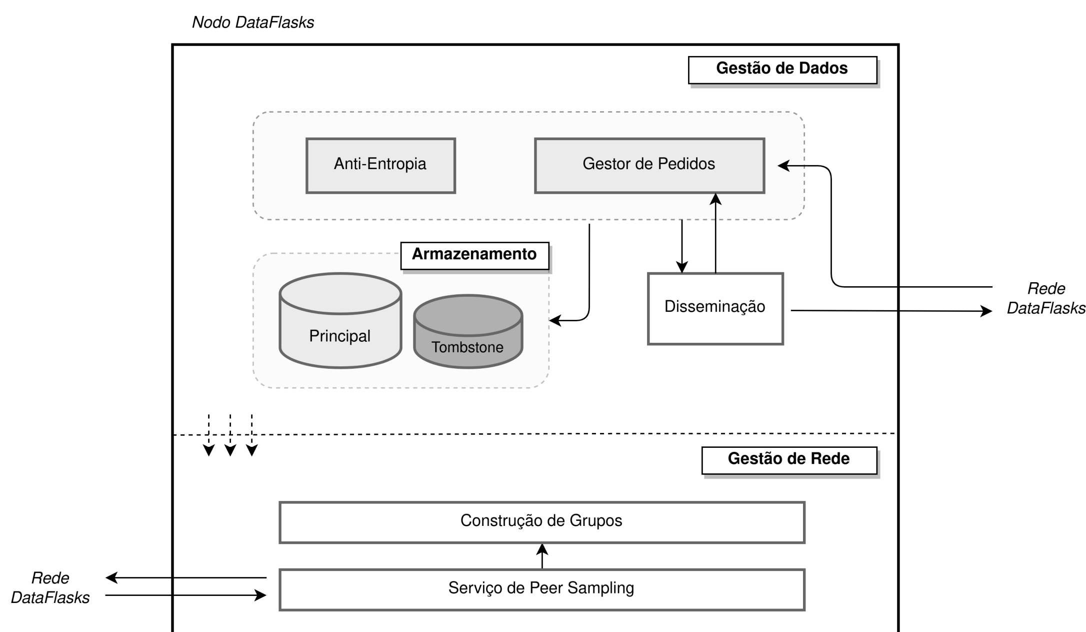


Figura 6: Arquitetura do Nodo DataFlasks

3.2 Modelo de acesso a dados

Com as alterações efetuadas no sistema de ficheiros, obtemos um sistema que segue um modelo de acesso a dados diferente, colmatando lacunas que o modelo *Write Once Read Many* apresentava. Aos utilizadores do iLSFS é permitida a criação, leitura, alteração e eliminação de ficheiros e diretorias do sistema de ficheiros.

Num cenário com múltiplos utilizadores, um utilizador pode criar e escrever um ficheiro, e quando a escrita acabar, qualquer outro utilizador pode realizar alterações ao ficheiro. Do mesmo modo, também esse ficheiro pode ser lido após a sua criação ou alteração, tanto de uma forma sequencial como aleatória. Se um ficheiro for eliminado por um utilizador este não é mais apresentado para leitura. Em relação às diretorias, a lógica de acesso aos dados, perante as diversas operações disponibilizadas, é bastante similiar ao que acontece com os ficheiros e ao que já era implementado pelo LSFS. Um utilizador pode criar uma diretoria, e qualquer outro utilizador pode realizar alterações à mesma, através da criação ou eliminação de diretorias ou ficheiros hierarquicamente abaixo, sendo todas estas escritas visíveis em leituras posteriores. A eliminação de diretorias pode acontecer por parte de qualquer utilizador e inicia um processo recursivo de eliminação de todos os ficheiros e/ou diretorias que façam parte da diretoria em questão. Adicionalmente, é assumido que depois de apagado, um ficheiro ou diretoria não volta a ser criado, contudo, esta restrição é apenas causada pela lógica implementada na junção de diretorias, Secção 4.3.2.

Com este modelo de acesso a dados obtemos um sistema de ficheiros mais competitivo e que oferece características similares às que outros sistemas de ficheiros distribuídos oferecem.

3.3 Operações

Uma das contribuições deste trabalho passou pelo melhoramento da usabilidade do LSFS, através da introdução de operações fundamentais. Como já foi sendo referido na visão geral do sistema, o iLSFS é composto por dois componentes que disponibilizam dois grupos de operações distintas, mas que em conjuntos possibilitam o fluxo de pedidos, desde a interface do sistema de ficheiros, até aos nodos de armazenamento. Estes dois componentes são a Interface POSIX e a API do Cliente.

3.3.1 Interface POSIX

A Interface POSIX tem como função disponibilizar um conjunto de operações ou chamadas ao sistema passivéis de serem intercetadas pela solução *userspace*, FUSE. Quando intercetadas, as chamadas ao sistema de ficheiros são redirecionadas para a nossa implementação lógica dessas operações, no componente Orquestrador, para que possam ser processadas e propagadas para a camada de armazenamento, através da API do Cliente. Deste modo, a Interface POSIX é estendida para facultar as novas operações de eliminação de ficheiros e diretorias, mediante a disponibilização das chamadas ao sistema *unlink* e *rmdir*.

3.3.2 API do Cliente

A API do Cliente fornece um conjunto de operações capazes de efetuar a comunicação entre a camada do cliente e os nodos de armazenamento. Esta API, para além de apresentar operações bastante similares

às que uma base de dados chave-valor oferece, necessita de expandir a sua definição de forma a englobar as estratégias implementadas no iLSFS.

As principais operações, enumeradas em 3.1, dizem respeito a *put*, *get*, *del* (*delete*) de dados, idênticas a uma API de uma base de dados chave-valor. As operações de *put* e *del* necessitam da especificação da sua chave e versão, mas também do tipo de dados a que se referem, podendo, num sistema de ficheiros, apenas ser ficheiros ou diretorias. Relativamente às operações de consulta, também enumeradas em 3.1, estas podem ser divididas em três operações: *get*, *get_latest_version* e *get_latest*. A primeira corresponde a uma consulta comum, tendo como parâmetros a chave e a versão que procuramos. A segunda, *get_latest_version*, satisfaz a necessidade de consulta da última versão de uma determinada chave, o que se torna pertinente a partir do momento em que queremos apresentar a versão mais atual ao utilizador, num sistema totalmente descentralizado com versionamento. Da mesma forma, temos a terceira operação de consulta, *get_latest*, que se apresenta como uma combinação das outras duas, requerendo os dados associados à última versão da chave a procurar.

```
put(key, version, file_type, data, size)
del(key, version, file_type)
data get(key, version)
version get_latest_version(key)
data get_latest(key)
```

Listagem 3.1: Operações sobre dados

Para além das operações base, são disponibilizadas outras direcionadas às estratégias de otimização implementadas pelo LSFS e que se encontram especificadas na Listagem 3.2.

As operações *put_batch* e *get_latest_batch* representam o seguimento do mecanismo de paralelização (Secção 2.3.5) associado às operações de leitura e escrita de um ficheiro, e que possibilita o envio, de uma forma paralela, dos vários pedidos de inserção ou consulta de chaves para a camada de armazenamento. A operação *put_batch* realiza a inserção de cada uma das chaves que recebe, enquanto a operação *get_latest_batch* recolhe a última versão das chaves e os dados que lhe são associados, de modo a apresentar para leitura dados mais atualizados.

```
put_batch(keys, datas, sizes)
get_latest_batch(keys, &datas)
```

Listagem 3.2: Operações direcionadas à paralelização de dados

Por outro lado, temos também um conjunto de operações específicas para a imposição da nova estratégia relativa ao tratamento dos metadados de diretorias, listadas em 3.3, e que têm um impacto positivo na saturação da rede. Estas operações serão alvo de uma análise aprofundada na Secção 4.6.

```
put_child(key, version, child_object)
data get_latest_metadata_size(key)
data get_latest_metadata_stat(key)
```

```
get_metadata_batch(keys, &datas)
```

Listagem 3.3: Operações direcionadas a metadados

3.4 Fluxo de Pedidos

Depois de possuído um conhecimento abrangente dos diversos componentes que compõe o iLSFS, é relevante entender quais os fluxos que um determinado pedido, que chegue ao sistema de ficheiros, pode seguir até que haja uma resposta. Para facilitar a compreensão deste fluxo, é importante acompanhar e visualizar os possíveis percursos que um pedido pode fazer, dentro do Cliente iLSFS e na rede de nodos, e que se encontram ilustrados na Figura 7.

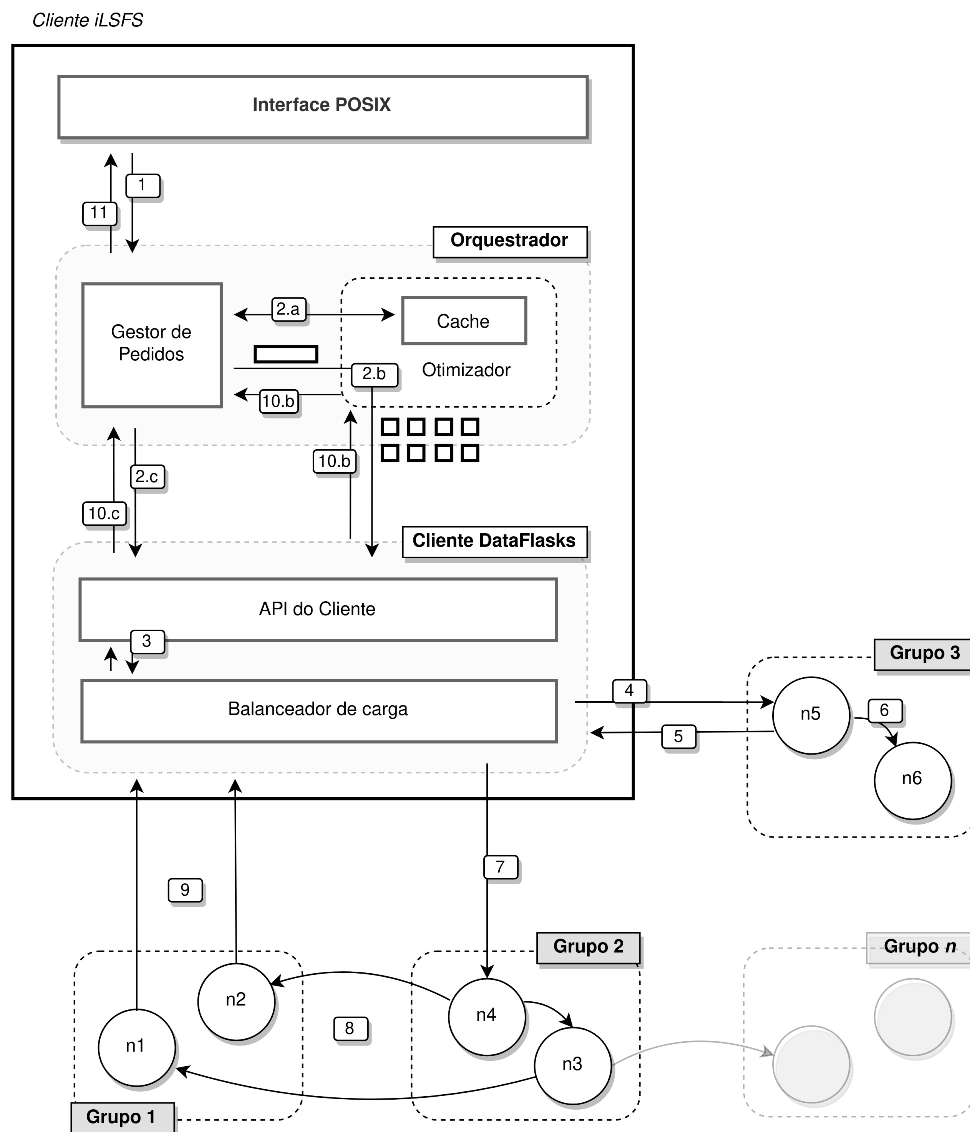


Figura 7: Fluxo de pedidos no iLSFS

Qualquer aplicação que utilize o iLSFS, fá-lo realizando um conjunto de chamadas ao sistema de ficheiros (de acordo com as operações disponibilizadas pela Interface POSIX) que são prontamente intercetadas pela solução *userspace* FUSE. Estas operações, depois de capturadas são transmitidas (Etapa 1) para processamento no componente Orquestrador e traduzidas em pedidos lógicos percetíveis pelas restantes camadas do sistema. Este processo realizado no Gestor de Pedidos pode seguir três caminhos, de forma à obtenção de resposta ao pedido realizado:

a. No caso de o pedido consistir na consulta de metadados de ficheiros ou diretorias, é realizada uma verificação na Cache (Etapa 2.a) pela informação necessária. Se esta informação se encontrar armazenada em cache do cliente, pode ser utilizada para responder ao utilizador (Etapa 11) consoante o tipo de pedido efetuado, caso contrário, o pedido terá de ser redirecionado para consulta remota nos nodos de armazenamento DataFlasks, através da API do Cliente. Este é um fluxo bastante utilizado em operações como *getattr*, *readdir*, *open*, ou outras que necessitem da consulta de metadados.

b. Quando o pedido corresponde à leitura (*read*) ou escrita (*write*) de um ficheiro, o processamento realizado segue uma lógica bem definida. De modo a clarificar o protocolo seguido, iremos acompanhar o exemplo da escrita de um bloco de tamanho 32k a partir do *offset* 8k (Etapa 2.b). A primeira ação efetuada é a verificação da versão que o ficheiro tem, de modo a marcar os blocos que serão escritos com versões superiores. Para isso é consultada a Cache, que terá sempre o valor que procuramos, uma vez que uma operação de escrita tem sempre associada uma operação de *open*, que povoa a cache com os metadados do respetivo ficheiro, na qual é adicionada a versão do mesmo. De seguida, os dados a serem escritos são divididos em 8 blocos de tamanho 4k na componente Otimizador, onde é aplicado o mecanismo de paralelização, tal como explicado na Secção 2.3.5. Os blocos agrupados são depois enviados, paralelamente com recurso à operação *put_batch* da API do Cliente (Etapa 3), ficando a aguardar a resposta aos pedidos efetuados. Quando, e se, uma resposta positiva for recebida por cada um dos pedidos efetuados, então esta é transmitida de volta ao Otimizador (Etapa 10.b), que por sua vez retorna o número de *bytes* escritos ao Gestor de Pedidos para que possa ser exteriorizado como o resultado final da operação (Etapa 11). Evidentemente, se tivéssemos recorrido ao exemplo de uma leitura de um ficheiro o procedimento a realizar seria bastante semelhante ao descrito anteriormente, com a exceção do primeiro passo realizado. Neste pedido seria na mesma necessário contactar a Cache, mas apenas para obter conhecimento do tamanho exato do ficheiro, de modo a verificar se o *offset* pedido se encontra dentro dos parâmetros válidos.

c. Por último, um pedido pode simplesmente seguir uma abordagem mais direta, onde é submetido ao processamento necessário no Gestor de Pedidos e direcionado para API do Cliente (Etapa 2.c), para que a resposta seja obtida remotamente (Etapa 10.c). Operações como *mkdir*, *rmdir*, *create*, *unlink*, entre outras, seguem esta abordagem.

Independentemente do percurso que um pedido faça no Orquestrador do iLSFS, quando este chega ao Cliente DataFlasks, ou mais concretamente à API do Cliente, o fluxo seguido para a obtenção de resposta é sempre idêntico. Cada um dos pedidos é disseminado com recurso ao balanceador de carga empregue (Etapa 4 e 7), até que este atinja um ou mais elementos do grupo de replicação capaz de o processar e de responder ao utilizador.

Durante a disseminação de pedidos no sistema, existem dois exemplos de comunicação com os nodos, para a obtenção de uma resposta. O primeiro acontece regularmente quando é empregue o *Smart load balancer* e corresponde a uma ligação direta com um nodo que consegue processar e responder ao pedido efetuado pelo cliente. Neste caso (Etapa 4), ao atingir o nodo *n5*, o pedido efetuado é processado e uma resposta é gerada para o cliente (Etapa 5), uma vez que o grupo no qual o nodo se inclui se encontra responsável por gerir a chave associada àquele pedido. Para além disso, e como forma de manutenção da coerência no sistema, a mensagem é ainda propagada para os restantes elementos do grupo (Etapa 6), que neste exemplo corresponde ao nodo *n6*, não havendo a necessidade de fornecer mais nenhuma resposta ao cliente.

Por outro lado, o nodo que o balanceador de carga seleciona nem sempre corresponde a um que pertence ao grupo capaz de satisfazer o pedido efetuado pelo cliente. Quando um pedido (Etapa 7) atinge um nodo que não o consegue processar, no exemplo, o nodo *n4*, o pedido é prontamente descartado, mas disseminado pelos restantes nodos da sua vista, *n1* e *n2* (Etapa 8). Neste caso específico, a propagação acontece apenas para este conjunto de nodos, uma vez que este processo se encontra otimizado, priorizando sempre o envio para um nodo que consiga processar o pedido, se tal existir na sua vista. Desta forma, quando o pedido é recebido pelos nodos *n1* e *n2*, estes são capazes de o processar e prontamente respondem ao cliente (Etapa 9). De salientar que a mensagem que ambos os nodos receberam não apresenta referência a qualquer resposta já entregue ao cliente, logo ambos os nodos geram uma resposta para o pedido e enviam-na para o respetivo cliente. Para além disso, posteriormente, ambos os nodos enviam o pedido para os restantes nodos do grupo, *n1* para *n2* e *n2* para *n1*, uma vez que desconhecem que o outro já observou aquela mensagem, porém, descartam-na se esta já tiver sido processada.

No final, quando o cliente receber a resposta ao seu pedido, esta é prontamente redirecionada para o Gestor de Pedidos (Etapa 10.c), onde é traduzida e apresentada como o resultado da operação (Etapa 11) inicialmente requerida.

4

Versionamento, Otimizações e Protótipo

Depois de exposta a arquitetura e fluxo de pedidos no iLSFS, é ainda necessário apresentar as diversas modificações realizadas ao sistema original, bem como as decisões seguidas que nos permitem melhorar a usabilidade no sistema de ficheiros.

4.1 Versionamento

O iLSFS é um sistema que necessita de versionamento para a gestão dos dados presentes no sistema de ficheiros. Desde o sistema de armazenamento chave-valor DataFlasks, no qual o LSFS se baseou, que o recurso a um versionamento, constituído apenas por um inteiro, permite assinalar modificações num objeto de dados (i.e., através do incremento do mesmo para sinalizar uma nova versão de uma chave). Contudo, a abordagem seguida não realizava nenhum controlo de concorrência em possíveis inserções divergentes sobre a mesma chave, atribuindo essa gestão para um utilizador do sistema.

O LSFS ao introduzir uma camada de gestão de ficheiros, baseou-se neste tipo de versionamento para conseguir controlar as modificações dos dados no sistema de ficheiros. Porém, o versionamento utilizado impedia o sistema de modificar ficheiros na presença de vários utilizadores, já que alterações efetuadas por utilizadores diferentes do criador do ficheiro, conduziriam à perda de atualizações e à não convergência para um estado coerente. Deste modo, houve necessidade de implementar um tipo de versionamento que permitisse obter uma relação de causalidade no sistema de ficheiros.

Em sistemas distribuídos, o recurso a um versionamento como *version vectors* [7] é uma solução amplamente utilizada para o estabelecimento de uma relação de causalidade no sistema. Sistemas de armazenamento como o DynamoDB [9] e o Riak [21] são alguns dos exemplos mais conhecidos que recorreram a este tipo de versionamento, ou variações, para versionar os seus dados.

Um ***Version vector*** é constituído por uma lista de pares (identificador, contador) através do qual se consegue estabelecer uma relação de causalidade entre objetos do sistema. Para isso, cada versão de um objeto terá de ter sempre um vetor associado, de modo que, pela análise deste, seja possível estabelecer se duas versões dum objeto têm origens concorrentes ou se apresentam uma ordem causal. Esta lógica só se torna possível se cada operação de atualização efetuada no sistema estiver sempre casualmente

relacionada com a leitura que a originou. Ou seja, quando é realizada uma atualização, tem que ser sempre fornecido um contexto para associar ao objeto, correspondente ao *version vector* do objeto a ser atualizado. Desta forma, cada entrada no vetor segue as seguintes regras:

A inicialização de uma entrada corresponde à criação de um par com o identificador do recurso x e com o contador a zero.

$$init(x) = (x, 0)$$

A atualização de uma entrada corresponde ao incremento em uma unidade do contador do par, se o x fornecido for igual ao *id* e a abstenção de qualquer atualização caso contrário.

$$atualiza(x) = \begin{cases} (id, c+1), & \text{se } x = \textit{id} \\ (id, c), & \text{se } x \neq \textit{id} \end{cases}$$

Porém, para a construção do vetor existem duas abordagens distintas no que diz respeito ao identificador a utilizar.

Version vector* com *id-per-server

A utilização de um identificador por cada servidor ou nodo do sistema permite que a gestão do versionamento de um objeto esteja sempre associada ao servidor que o processou. Quando um objeto é inserido por um cliente, o servidor que processar o pedido é responsável por marcar o objeto com o seu identificador e incrementar a sua entrada no contador, seguindo as regras definidas. Desta forma, cada atualização efetuada por aquele servidor é visível no vetor, o que permite comparar os vetores de forma a relacionar a sua história causal. Contudo, se atualizações concorrentes forem processadas pelo mesmo servidor, o vetor não conseguirá apresentar as duas versões como conflituosas, uma vez que a versão das duas atualizações será a mesma. Seguindo a lógica descrita de atualização do vetor, esta resultaria na sobreposição de dados e na perda de uma das atualizações. Este problema, nesta abordagem em específico, encontra-se resolvido por uma variação dos *Version vector* com *id-per-server*, denominado Dotted Version Vector [33].

Version vector* com *id-per-client

O vetor também pode utilizar os clientes do sistema como identificador único para gerir o versionamento dos objetos. Neste caso, antes de um objeto ser inserido, é marcado com o identificador do cliente e incrementado o contador, nessa mesma entrada, seguindo as regras definidas. Desta forma, cada atualização efetuada por aquele cliente é visível no vetor, permitindo relacionar a história causal das versões de um objeto. Este versionamento pode ocorrer tanto no cliente, antes do envio do pedido, como no servidor, ao ser processado, desde que este disponha do identificador do cliente para marcar a nova atualização.

Estas duas abordagens para além das características apresentadas, têm implicações diferentes no que ao tamanho do vetor se refere. Ao utilizarmos um abordagem *id-per-server*, o vetor pode conter,

no limite, o número total de nodos presentes no sistema distribuído, ou um subconjunto destes, i.e., os nodos capazes de processar o objeto. Já na abordagem *id-per-client*, o vetor terá tantas entradas quantos clientes manipularem o mesmo objeto, que no limite poderão ser todos os clientes do sistema.

O iLSFS é um sistema distribuído que privilegia a sua escalabilidade, estando desenhado para redes de muita larga escala, logo, não pode apresentar nenhum mecanismo que seja um entrave à sua escalabilidade. O recurso a *version vector* com *id-per-server* implica que o tamanho do vetor seja uma limitação ao crescimento do sistema, já que restringe o número de nodos em cada grupo de replicação (uma vez que cada grupo é responsável por um conjunto de dados). A mesma premissa pode ser utilizada para os *version vector* com *id-per-client*, no entanto, o iLSFS encontra-se desenvolvido para um ambiente bem específico de aproveitamento dos recursos dos dispositivos IoT, onde o número de utilizadores autorizados a usar o sistema de ficheiros é restrito e nunca maior que o número de nodos que o sistema poderá vir a apresentar.

Para além disso, a abordagem *id-per-server* no iLSFS levaria ao aparecimento de falsa concorrência, devido às características de disseminação deste sistema. Quando uma atualização é efetuada por um utilizador, esta não é processada apenas por um nodo, e depois replicada para outros elementos responsáveis pelo mesmo conjunto de dados, como acontece em muitos sistemas distribuídos. No iLSFS uma operação pode ser processada ao mesmo tempo por mais que um nodo, tal como foi descrito no fluxo de pedidos (Secção 3.4), o que levaria ao aparecimento de versões concorrentes, quando na realidade se estariam a referir à mesma operação. Este problema poderia ser resolvido através da introdução de metadados adicionais no versionamento que facilitariam a reconciliação das versões numa só, no entanto, estas informações suplementares iriam apenas acentuar o problema do tamanho do vetor nesta abordagem.

Posto isto, a abordagem que melhor se enquadra nas características deste sistema de ficheiros é o recurso a um versionamento, *version vector*, com *id-per-client*. Na Figura 8 é possível acompanhar um exemplo da interação de dois clientes, *c1* e *c2*, a realizarem atualizações sobre o mesmo objeto de dados no sistema e perceber como o *version vector* com *id-per-client* funciona no iLSFS.

Os dois clientes, no ponto *a*, começam por contactar o substrato de armazenamento de modo a obterem os dados relativos ao ficheiro *file*. No entanto, como o ficheiro não existe, recebem de volta um conjunto de dados vazio e também um vetor versão vazio. Posteriormente, no ponto *b*, ambos efetuam a escrita de dados no ficheiro, marcando a inserção com a versão relativa ao cliente - o cliente *c1* adiciona no vetor a entrada *{(c1,1)}*, enquanto que o cliente *c2* adiciona *{(c2,1)}*, uma vez que ambos partiram de uma vista inexistente do ficheiro */file*. Apesar da operação do cliente *c2* chegar ao nodo primeiro que a operação do cliente *c1*, o nodo é capaz de perceber, ao analisar o vetor, que as versões para aquele objeto são de facto concorrentes, e que tem de persistir as duas.

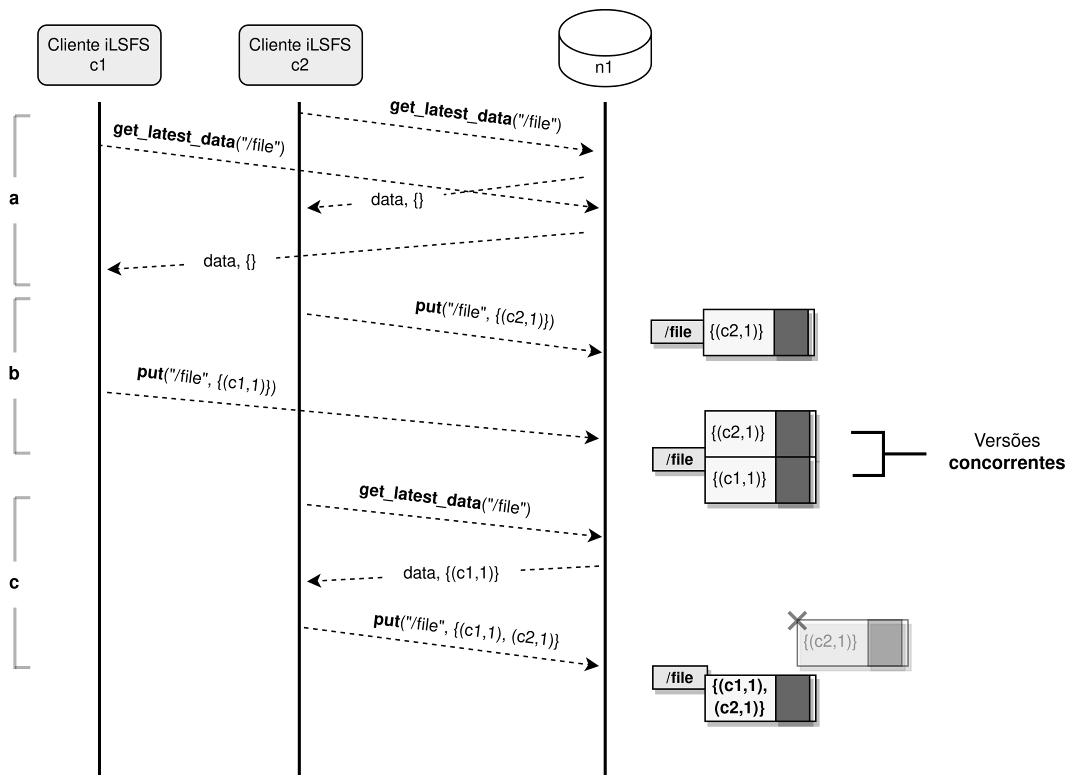


Figura 8: *Version vectors* com *id-per-client* no iLSFS [1]

4.2 Eliminação de dados

A introdução da operação de eliminação de dados foi um dos objetivos definidos para o alcance de uma melhor usabilidade no sistema.

Em sistemas distribuídos que aplicam uma coerência eventual, a introdução da operação de eliminação de dados apresenta um desafio adicional. Nestes sistemas, em virtude das inerentes partições da rede, não existem garantias de que as operações direcionadas a um determinado conjunto de dados, sejam realizadas em todos os nodos responsáveis por armazenar esse conjunto de dados. Na eliminação de dados, esta característica tem um especial destaque pela natureza da operação, que uma vez realizada, se nenhuma informação for guardada, não é possível distinguir se os dados foram eliminados, ou se nunca chegaram a ser inseridos no nodo. Para além disso, se ainda considerarmos a recuperação de dados, indispensável nestes sistemas, os dados uma vez eliminados podem acabar por reaparecer, por via de nodos que não tenham recebido a operação de eliminação, o que resultaria na anulação da mesma, já que não existiria diferença entre dados eliminados ou inexistentes. Posto isto, existe a necessidade de implementar uma estratégia capaz de persistir a operação de eliminação, para que possa ser propagada e nunca revertida.

A abordagem mais comum quando se tenta resolver este problema, em sistemas eventualmente coerentes, é o recurso a uma estratégia conhecida por *Tombstones*, empregue, por exemplo, no sistema

[1] Esquema simplificado do que ocorre no iLSFS.

de armazenamento Cassandra [22]. Uma ***Tombstone*** é responsável por armazenar ou assinalar, de forma persistente, o conjunto de dados eliminados do sistema. Assim, mesmo que um dos nodos não tenha recebido a operação de eliminação, os dados presentes como *tombstone* podem ser utilizados para que os outros nodos de replicação possam chegar a um estado coerente.

No iLSFS seguimos também esta estratégia para a implementação da operação de eliminação, no entanto, ao invés de marcamos apenas que uma determinada entrada se encontra eliminada, a cada nodo DataFlasks é adicionada uma nova tabela, responsável por armazenar as chaves e respetivas versões eliminadas do sistema. Esta implementação deve-se à necessidade de divulgação, de uma forma eficiente, do conjunto de chaves eliminadas entre todos os nodos do grupo de replicação, de forma a que seja possível uma fácil manipulação da quantidade de chaves eliminadas, difundidas durante o processo de recuperação de dados.

Posto isto, resta perceber como se sucede, de forma mais concreta, a operação no armazenamento do nodo DataFlasks. Quando é recebida uma operação de eliminação de dados, e se estes se encontrarem dentro do intervalo de dados pelos quais o nodo é responsável, é adicionada uma nova entrada na *tombstone*, persistindo a operação de eliminação. De igual modo, é também verificado se os dados a eliminar se encontram no armazenamento principal do nodo, e se for o caso, então é efetivada a eliminação dos mesmos, bem como de todas as versões que lhe antecedem na história causal. De realçar que, mesmo que o nodo não tenha armazenada a chave que a operação pretende apagar, a chave e a respetiva versão apagada são na mesma incorporadas na *tombstone*. Isto permite que, mesmo recebendo a chave, pelo mecanismo de recuperação de dados, o nodo saiba quando não a deve incorporar.

Como é espectável, o recurso a esta estratégia exige uma atualização do mecanismo de recuperação de dados, de modo a acomodar também as chaves presentes na *tombstone*. Assim, para além das chaves persistidas no armazenamento principal, parte do espaço de chaves disseminadas é agora dedicado às chaves eliminadas, permitindo que, eventualmente, todos os nodos dos grupos de replicação recebam também as operações de eliminação.

4.3 Reconciliação de conflitos

Em qualquer sistema eventualmente coerente, existe a necessidade de desenvolver mecanismos que possibilitem a resolução de conflitos provenientes da divergência do estado das réplicas do sistema distribuído. Durante o normal funcionamento do sistema, estas divergências de estado podem originar o aparecimento de versões conflituosas que requerem a introdução de resoluções automáticas ou manuais. Com a implementação de um novo tipo de versionamento, o iLSFS torna-se capaz de corretamente identificar a história causal de um objeto e a respetiva presença de concorrência entre versões. Perante isto, e tendo em conta as características de coerência do iLSFS, torna-se indispensável a implementação e atualização das estratégias de resolução destes conflitos, de modo que, independentemente das diferentes versões existentes para um objeto, o resultado final de consultas seja sempre o mesmo.

4.3.1 Seleção do ficheiro

Com a definição do novo modelo de acesso a dados, onde é considerada a manipulação de ficheiros por todos os utilizadores do sistema, temos de ter em consideração o surgimento de versões conflituosas. Como tal, existe a necessidade de implementar soluções que levem à resolução deste problema.

O iLSFS optou por tomar uma abordagem semelhante à Last-Write-Wins (LWW) [22] seguida pelo sistema de armazenamento Cassandra, quando este se depara com duas versões concorrentes. Porém, o iLSFS não utiliza *timestamps* para distinguir versões concorrentes, recorrendo ao identificador do cliente que realizou a atualização para decidir qual a versão que é apresentada. Assim, a versão selecionada corresponde sempre àquela com o menor identificador do cliente relativamente ao conjunto de versões concorrentes existentes.

Regressando ao exemplo da Figura 8, no ponto *c*, o cliente *c2* volta a consultar o substrato pelo ficheiro */file*. O nodo ao perceber que dispõe de duas versões concorrentes seleciona aquela com identificador do cliente menor - como $c1 < c2$ a versão enviada ao cliente é a *{(c1,1)}*. Posteriormente, o Cliente, baseando-se nos dados recebidos, realiza um conjunto de escritas no ficheiro que terão uma versão representatitva da ordem causal que originou a atualização. Desta forma, a versão desta atualização do ficheiro é *{(c1,1),(c2,1)}*. De notar ainda que o nodo ao receber esta atualização percebe que esta se sobrepõe à existente *{(c2,1)}* e, portanto, esta é eliminada do seu armazenamento.

4.3.2 Junção de diretorias

Uma das características do iLSFS é a liberdade de manipulação de dados entre utilizadores sobre a mesma diretoria. Isto torna-se possível devido à implementação de um mecanismo de junção de diretorias que permite a reconciliação de versões concorrentes recorrendo ao conceito de Conflict-free Replicated Data Types (CRDTs) [45]. A adoção deste mecanismo viabiliza a convergência das réplicas de dados para um estado coerente, garantindo que não existe perda de informação, através da união das estruturas que compõem uma diretoria.

Com a introdução da operação de eliminação, também a estrutura de referências de uma diretoria teve de exteriorizar possíveis eliminações das suas referências. À partida, uma simples eliminação das referências seria suficiente, no entanto, ao incluirmos na equação a estratégia de junção de diretorias, problemas semelhantes aos que nos levaram a optar pela abordagem de *tombstones* podem surgir. Consideremos o cenário presente na Figura 9, onde de uma versão da diretoria */home* são originadas duas versões concorrentes, por exemplo, devido a partições na rede. De um lado, o utilizador *c1* cria a diretoria */home/docs* e do outro o utilizador *c2* apaga a diretoria */home/tese*. É evidente que o resultado esperado da junção destas duas versões corresponde a uma estrutura de referências com o ficheiro */home/file* e a diretoria */home/docs*, no entanto, como não existe nenhuma dica, nesta estrutura, de que a diretoria */home/tese* foi apagada, o resultado da união das duas versões apresenta a diretoria que devia estar eliminada. Logo, existe a necessidade de adaptar a estratégia de junção de diretorias de modo que esta funcione corretamente perante a eliminação de ficheiros e diretorias.

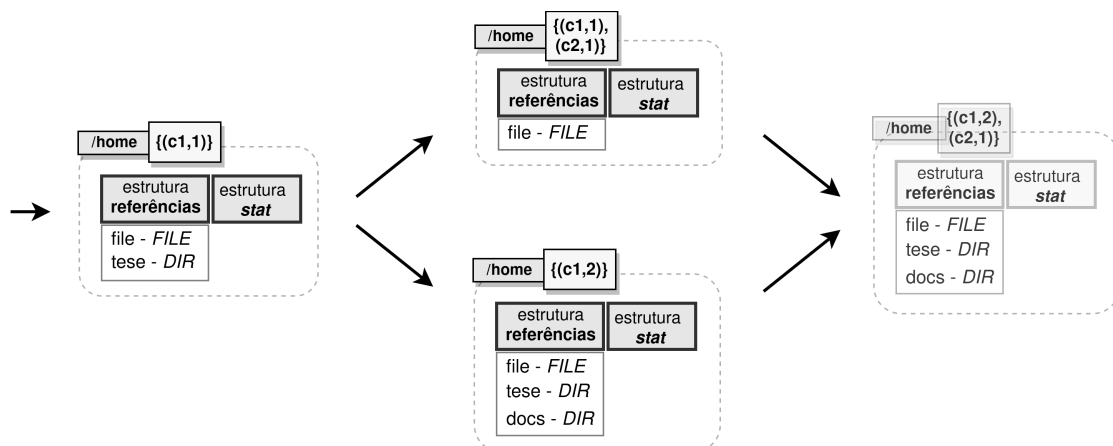


Figura 9: Problema na eliminação de referências de uma diretoria

Seguindo o conceito de CRDTs, foi adotada uma estratégia de persistência das referências eliminadas complementarmente às referências adicionadas já existentes. Esta estratégia, também conhecida por *2P-Set* [44], recorre a duas estruturas, uma *A* responsável por armazenar os dados adicionados, neste caso, as referências criadas (ficheiros ou diretorias), e outra *R* responsável por armazenar as referências eliminadas da diretoria, ou seja, uma *tombstone*. Desta forma, apenas os elementos (*e*) que se encontram na estrutura de referências adicionadas *A* e que ainda não foram eliminados podem ser considerados como fazendo parte da diretoria e expostos aos utilizadores.

$$\forall e \in A \setminus R$$

O recurso a esta estratégia permite que na eventualidade de aparecerem versões concorrentes da mesma diretoria, a junção destas possa ser realizada recorrendo à união dos elementos das duas estruturas. Na Figura 10 é possível visualizar o mesmo exemplo anterior, mas agora com a junção de diretorias a convergir para o estado pretendido. Neste cenário, a diretoria */home* apresenta como referências o ficheiro */home/file*, bem como a diretoria */home/tese*, estando como adicionados (*ADD*) na estrutura de referências. De seguida, por partições na rede, por exemplo, são originadas duas versões concorrentes, onde na primeira é eliminada a diretoria */home/tese* e na segunda é criada a diretoria */home/docs*. É possível observar que a diretoria eliminada pertence agora à estrutura de referências eliminadas (*DEL*). Quando o sistema volta a comunicar, percebe que existem duas versões concorrentes da mesma diretoria, acionando prontamente o mecanismo de junção de diretorias. Como resultado obtemos um nova versão da diretoria */home*, onde apresenta ao utilizador o ficheiro */home/file* e a diretoria */home/docs*.

Contudo, o recurso a esta abordagem apresenta algumas desvantagens. A lógica das estruturas apenas permite o seu crescimento, o que implica que tenham de ser armazenadas todas as referências alguma vez eliminados na diretoria. Do mesmo modo, quando uma referência passa para o estado de apagada já não pode voltar a ser criada e voltar a transitar para o estado de adicionada, visto que poderia originar convergências incorretas entre versões concorrentes, o que resulta no impedimento de voltar a criar um ficheiro ou diretoria que outrora tenha sido eliminado do iLSFS.

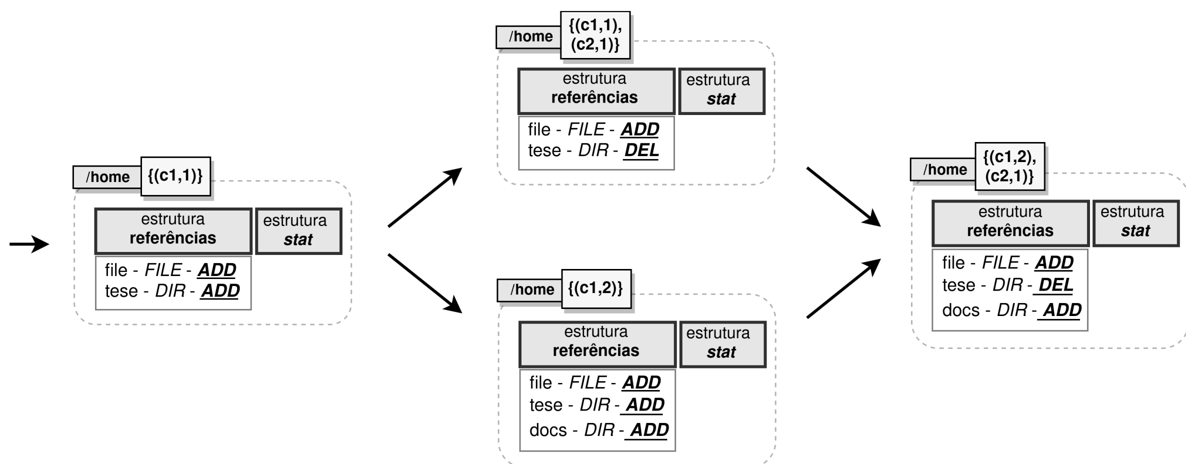


Figura 10: Junção de diretorias no iLSFS

4.4 Versão apresentada

Com a introdução de um versionamento capaz de estabelecer a ordem causal de atualizações de um objeto, e a introdução da operação de eliminação de dados num sistema eventualmente coerente, existe a necessidade de desenvolver um algoritmo capaz de apresentar ao utilizador do sistema de ficheiros a versão mais atual dos dados que pretende consultar.

Uma das operações de maior relevo e mais efetuada no iLSFS corresponde à procura da última versão de uma determinada chave. Este tipo de consultas, efetuadas a partir de operações da API do Cliente como *get_latest_version(key)* ou *get_latest(key)*, contactam um pré-definido número de nodos, de forma a selecionar a última versão de uma determinada chave requerida. Estas consultas possuem um papel fundamental no iLSFS, uma vez que, tratando-se de um sistema eventualmente coerente, o contacto de apenas um nodo pode não ser suficiente para a recolha da chave mais recente, o que, por conseguinte, impossibilita a apresentação da informação mais atual ao utilizador.

Com a incorporação da operação de eliminação de dados, também estas consultas necessitaram de começar a refletir a possibilidade da existência de versões apagadas no sistema. Para isso, para além de serem recolhidas as últimas versões concorrentes de uma determinada chave, são também enviadas, como resposta a um pedido, as últimas versões concorrentes e eliminadas do sistema de ficheiros. Esta nova informação, agora integrada nas respostas aos pedidos, requer o desenvolvimento de uma estratégia capaz de processar e selecionar qual a versão mais recente que não se encontra eliminada do sistema. Na Figura 11 é apresentado um exemplo, onde a consulta pela última versão do ficheiro */file*, é realizada através do contacto de quatro nodos diferentes. Neste cenário, por momentos de indisponibilidade da rede, os nodos do sistema exibem versões divergentes do ficheiro em questão, e ao serem agregadas, do lado do cliente, apenas uma é apresentada.

Desta forma foi desenvolvido o Algoritmo 1, capaz de filtrar as versões recebidas e retornar a que melhor representa o estado da chave pesquisada, no armazenamento remoto. O algoritmo recebe como variáveis de entrada, um conjunto de versões de dados, *versões_normais*, e um conjunto de versões

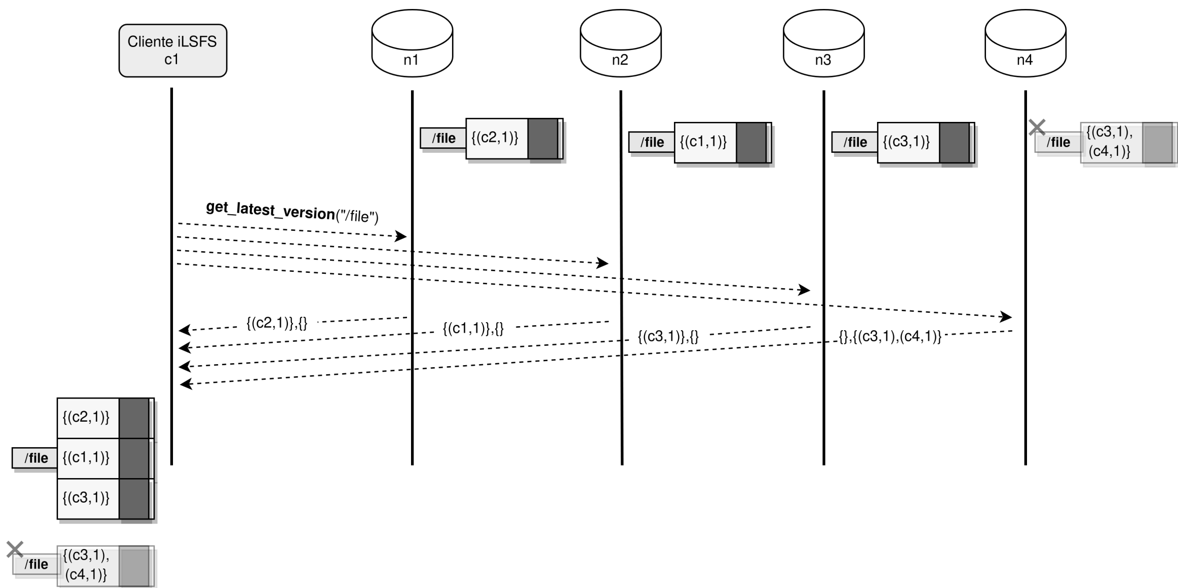


Figura 11: Consulta de última versão a múltiplos nodos

apagadas, *versões_apagadas*, representando aos dados obtidos e agregados dos nodos contactados. O primeiro passo tomado corresponde à inicialização de uma variável denominada *lista_últimas_versões*, e que armazenará todas as versões concorrentes que não foram apagadas. De seguida, são percorridas todas as versões presentes no conjunto de *versões_normais* (linha 2), para serem analisadas e avaliadas. Para isso, é mantida uma variável de controlo, inicializada a falso, denominada *apagado*, que permite memorizar se a versão em análise se encontra apagada ou não. Cada *versão*, é depois comparada com as presentes no conjunto de *versões_apagadas* (linha 4), e se existir alguma *versão_apagada* que seja superior ou igual à *versão* (linha 5), no que à história causal diz respeito, a variável *apagado* passa a tomar o valor de verdadeiro (linha 6), indicando que a *versão* é inválida, pois encontra-se eliminada. No caso de não existir nenhuma *versão_apagada* que inviabilize a *versão* em análise, o processo segue, de forma a verificar se a versão pode integrar a *lista_últimas_versões* (linha 10). Nesta fase, é criada a variável, *concorrente_contador*, responsável por contar quantas versões, presentes na *lista_últimas_versões*, são concorrentes à *versão* em análise, e a variável *i*, associada à iteração pela *lista_últimas_versões*. Com isto, é então percorrida a *lista_últimas_versões* (linha 13) e verificado se a variável *versão* sucede casualmente algum elemento da lista (linha 14). No caso de tal se verificar, o elemento em questão é substituido pela *versão*, uma vez que corresponde a uma versão superior. Caso contrário, e se a variável *versão* for concorrente ao elemento, *ult_versão* em análise (linha 17), o contador *concorrente_contador* é incrementado em uma unidade (linha 18). Posteriormente, ao percorrer a lista das últimas versões é verificado, ainda, se o contador tem o mesmo valor do tamanho da *lista_últimas_versões* (linha 22), o que, no caso de ser verdade, indica que a *versão* é concorrente a todos os elementos presentes na *lista_últimas_versões*, e então tem de ser adicionada à mesma lista (linha 23), por se tratar de uma versão atual e válida.

Algoritmo 1: Pseudo-código do processo de seleção da versão a apresentar

```
   Entrada: versões_normais, versões_apagadas
1  lista_últimas_versões ⟵ [];
2  for versão em versões_normais do
3  |  apagado ⟵ falso;
4  |  for versão_apagada em versões_apagadas do
5  |  |  if versão_apagada >= versão then
6  |  |  |  apagado ⟵ verdadeiro;
7  |  |  |  break;
8  |  |  end
9  |  end
10 |  if apagado = falso then
11 |  |  concorrente_contador ⟵ 0;
12 |  |  i ⟵ 0;
13 |  |  for ult_versão em lista_últimas_versões do
14 |  |  |  if versão > ult_versão then
15 |  |  |  |  lista_últimas_versões[i] ⟵ versão;
16 |  |  |  |  break;
17 |  |  |  else if versão concorrente a ult_versão then
18 |  |  |  |  concorrente_contador ⟵ concorrente_contador+1;
19 |  |  |  end
20 |  |  |  i ⟵ i+1;
21 |  |  end
22 |  |  if concorrente_contador = tamanho(lista_últimas_versões) then
23 |  |  |  adiciona versão a lista_últimas_versões;
24 |  |  end
25 |  end
26 end
27 escolher_última_versão(lista_últimas_versões);
```

Por fim, a variável *lista_últimas_versões* irá acomodar todas as versões concorrentes e válidas, contudo apenas uma será selecionada, através da função *escolher_última_versão*. A seleção de uma só versão segue a estratégia de reconciliação de conflitos, exposta na Secção 4.3.

Voltando ao exemplo da Figura 11, depois de executado o algoritmo, a versão apresentada ao cliente seria a *{(c1,1)}*. Isto aconteceria uma vez que a versão apagada *{(c3,1),(c4,1)}* invalidaria a versão *{(c3,1)}*, resultando em apenas duas versões válidas e concorrentes, *{(c1,1)}* e *{(c2,1)}*. Perante isso, e segundo a reconciliação de conflitos a versão escolhida corresponde ao cliente *c1*.

4.5 Cache

O LSFS é um sistema de ficheiros distribuído que beneficia de uma modelo não estruturado para a obtenção de uma elevada escalabilidade, no entanto, esta característica que o torna único limita-o na

obtenção de um desempenho competitivo. Como já foi descrito, um dos problemas deste sistema é a enorme quantidade de mensagens que têm de circular para a manutenção do seu normal funcionamento e que provocam uma saturação na rede de nodos. Desta forma, é essencial implementar mecanismos capazes de mitigar este problema. Uma das formas de atingir este objetivo passa pela diminuição das operações efetuadas remotamente, através do recurso, por exemplo a uma Cache, no Cliente iLSFS.

4.5.1 Ficheiros abertos

Num sistema de ficheiros, quando decorrem escritas ou leituras, desde a abertura até à libertação do descritor do ficheiro, ocorre um conjunto de operações, onde se destaca a consulta de metadados (chamada ao sistema *getattr*) pelas recorrentes invocações realizadas ao sistema. Claro que efetuar repetidamente esta operação ao armazenamento do sistema de ficheiros não só contribui para o problema de saturação da rede já existente, como também provoca um impacto negativo no desempenho das operações de leitura e escrita de ficheiros no LSFS. Por este motivo, o LSFS apresenta uma cache de ficheiros abertos capaz de evitar que constantes operações de consulta de metadados sejam realizadas aos nodos de armazenamento durante a manipulação de escritas e leituras em ficheiros.

Sempre que num ficheiro uma escrita ou leitura é efetuada, ocorre previamente uma operação de abertura (*open*) ou criação (*create*), que obtém ou inicializa os metadados do respetivo ficheiro. Deste modo, e aproveitando esta lógica presente num sistema de ficheiros, o LSFS armazena temporariamente os metadados provenientes destas operações para que possam ser utilizados sempre que uma consulta de metadados seja necessária. Por outras palavras, quando uma operação de abertura é realizada, os metadados requeridos remotamente são armazenados na estrutura de ficheiros abertos. Enquanto o ficheiro se encontrar aberto podem ser realizadas leituras ou escritas, bem como outras operações passíveis de alterar os metadados desse ficheiro. Quaisquer operações que necessitem de consultar os metadados do ficheiro são direcionadas para esta cache, evitando contacto com nodos de armazenamento. No final, quando a operação *close* ou *release* ocorrer, o ficheiro é removido da cache de ficheiros abertos e no caso de haver indicação da alteração nos metadados em relação aos originais, é atualizado o armazenamento remoto com o novo conjunto de metadados relativos ao ficheiro. É de realçar que operações de escrita e leitura são sempre direcionadas para a rede DataFlasks, porém, todos as outras que manipularem com os metadados do ficheiro realizam as alterações necessárias nos metadados armazenados temporariamente em cache e só no final são propagadas para armazenamento persistente. Em relação à operação de criação de ficheiros, a estratégia utilizada é igual, porém os metadados em memória terão os valores por omissão de criação, como tamanho do ficheiro vazio, ou atribuídos pela operação, como permissões, entre outras.

No entanto, nesta nova iteração, o iLSFS ao introduzir o novo tipo de versionamento existe a necessidade de estender e adaptar esta cache ao novo paradigma. Como foi descrito na secção sobre o versionamento, cada operação de modificação de dados requer um contexto de forma a manter a relação de causalidade no sistema, o que se aplica neste caso, à escrita de um ficheiro e dos seus metadados.

Existem duas estratégias de modo a obter este contexto. A primeira passa pela consulta da última versão antes da efetivação da escrita de cada um dos blocos que compõe a operação de escrita (*write*) num ficheiro. A segunda passa pelo armazenamento, em cache, da versão que o ficheiro tem aquando da sua abertura, ou seja, uma estratégia idêntica à descrita anteriormente. A opção com maior vantagem é a segunda, uma vez que a primeira levaria à repetição de operações de consulta da versão, consoante o número de blocos a serem escritos.

Desta forma, para além de serem armazenados os metadados do ficheiro aquando da abertura do mesmo, é também guardada a versão que o ficheiro apresenta. Em si a versão utilizada será a versão associada aos metadados (*iblock*) e não aos blocos de dados (*dblock*) a serem alterados. No entanto, a utilização desta estratégia funciona porque a última operação efetuada numa modificação de dados no ficheiro é sempre a atualização do *iblock* do ficheiro, o que imprime neste uma versão igual ou superior a todos os *dblock* associados, que compõem o ficheiro. Deste modo, todas as modificações a ocorrer serão subsequentes entre elas ao utilizar a versão em cache.

4.5.2 Cache de diretorias

Para além da necessidade da existência de uma cache direcionada a ficheiros é também importante que ao iLSFS seja adicionada uma cache de diretorias capaz de evitar acessos ao substrato de armazenamento. Em operações sobre metadados, com especial incidência na manipulação de várias diretorias, uma grande parte das operações realizadas são direcionadas a consultas recorrentes dos metadados de diretorias. Estas operações apesar de necessárias para a manutenção da lógica do sistema de ficheiros, sobrecarregam o sistema com constantes pedidos. Posto isto, foi desenhada uma cache de diretorias responsável por armazenar temporariamente o conjunto de metadados das diretorias mais acedidas pelo utilizador.

A cache implementada segue uma política Least Recently Used (LRU), ou seja, uma cache que proporciona a presença em memória dos dados recentemente acedidos pelo utilizador, uma vez que quando esta atinge o seu limite de armazenamento é aplicada uma política de remoção da entrada que há mais tempo não é utilizada pelo utilizador.

No entanto, o recurso a esta estratégia tem de ter em consideração o modelo de acesso a dados definido, nomeadamente, a possibilidade de vários utilizadores poderem realizar operações sobre a mesma diretoria. Consideremos o cenário presente na Figura 12, onde dois clientes, *c1* e *c2* estão a efetuar operações sobre a mesma diretoria, denominada */Desktop*, que contém o ficheiro *file1*. Nos dois utilizadores, durante o primeiro acesso à diretoria, os metadados seriam carregados para memória, de forma a que futuras invocações de consulta fossem feitas à cache, e não ao substrato de armazenamento. No caso do cliente *c2* efetuar a criação de um ficheiro *file2*, isto originaria uma alteração nos metadados da diretoria, de forma a que esta apontasse agora também para esse ficheiro, ficando a diretoria representada pelos ficheiros *file1* e *file2*. No entanto, apesar de esta alteração ocorrer tanto nos nodos de armazenamento

como nos metadados da cache do cliente *c2*, não se refletiria no que o cliente *c1* armazenaria em memória, uma vez que esses metadados continuavam a apontar apenas para o ficheiro *file1*. Isto quer dizer que os metadados presentes na cache de um cliente podem facilmente representar uma versão desatualizada do que os nodos DataFlasks armazenam.

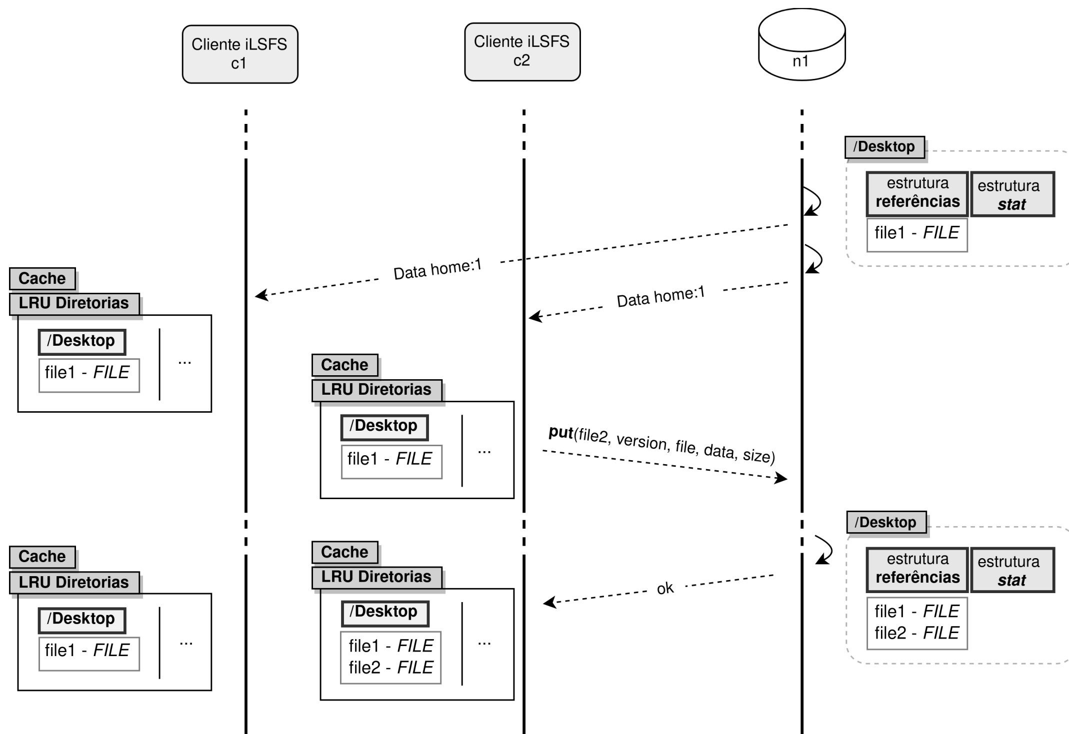


Figura 12: Cenário de Cache de diretorias sem atualização

Deste modo, existiu a necessidade de adicionar um tempo, parametrizável pelo utilizador, no fim do qual é considerado que a diretoria armazenada necessita de ser atualizada. Com isto tentamos sempre apresentar ao utilizador a informação mais correta possível. Claro que pode continuar a ocorrer o cenário descrito, mesmo com um tempo de atualização baixo, no entanto, é representativo de uma incoerência temporária, tal como é assumido que acontece nos nodos DataFlasks.

Desta forma, um utilizador pode definir um tempo de atualização consoante o número de utilizadores presentes no sistema e as operações que estes efetuam sobre diretorias. Para além disso, também consegue estabelecer um tamanho de cache que considere apropriado para a quantidade de diretorias que vai aceder.

4.6 Metadados de diretorias

Os metadados de diretorias possuem no iLSFS um papel essencial para a manutenção da lógica no sistema de ficheiros. Relembrando a organização de dados e metadados, Secção 2.3.3, cada diretoria é definida pelo conjunto dos seus metadados, onde se inclui referências para todos os ficheiros e/ou diretorias hierarquicamente abaixo. O recurso a esta estratégia permite ao iLSFS obter, de uma forma simples,

todos os dados relativos à diretoria mesmo num sistema totalmente descentralizada e não estruturado, onde não existem nodos dedicados a metadados ou dados.

No sistema LSFS, a forma como é realizada a gestão das operações e comunicações associadas à abordagem dos metadados de diretorias não é a mais adequada, nem a mais eficiente, fomentando o problema de saturação na rede. Neste sistema a manipulação das diretorias envolve os seguintes passos: Quando um ficheiro é criado no sistema de ficheiros, para além de despoletado o pedido para a criação do mesmo, também é gerado um pedido adicional para a atualização dos metadados da diretoria pai, de modo a que esta alteração se possa refletir nas referências da mesma. No entanto, para efetuar esta atualização, é necessário a consulta do conjunto total de metadados da diretoria (estrutura de referências + estrutura *stat*), para que sejam alterados na camada do cliente e, posteriormente, inseridos no substrato com o novo ficheiro como referência. É evidente que a constante comunicação do conjunto total de metadados, sempre que é necessária uma atualização dos mesmos tem um impacto negativo na saturação do sistema, quando a única alteração necessária apenas diz respeito à inserção de uma nova referência. Para além disso, a comunicação destes metadados ocorre de uma só vez, ou seja, independentemente do tamanho dos metadados da diretoria estes são enviados para o utilizador como se de um único bloco se tratasse. Esta abordagem torna-se um problema quando a quantidade de metadados associados à diretoria excede o tamanho máximo do pacote do protocolo UDP, dado que ao tratar estes metadados como um bloco é também utilizado apenas um pacote para o envio da totalidade de metadados. Esta superação do tamanho máximo do pacote UDP ocorre facilmente no LSFS, visto que os metadados de uma diretoria não têm definido nenhum limite no número de referências que podem ser guardadas e associadas à diretoria pai, o que significa que o tamanho dos metadados de diretorias aumenta consoante o número de ficheiros e diretorias que lhe são adicionadas.

Assim, existiu a necessidade de desenvolver uma nova estratégia para a comunicação dos metadados das diretorias, mantendo a forma como a organização de dados e metadados se encontra definida. Em primeiro lugar foi necessário evitar o envio de tantos dados, sempre que uma atualização dos metadados da diretoria se verificasse. Para isso foi construída a nova operação na API do Cliente - *put_child*. Esta operação permite adicionar aos metadados de uma diretoria uma nova referência na sua estrutura, transmitindo para os nodos DataFlasks a informação estritamente necessária sobre o "filho", que é agora associado a esta diretoria. A informação de um filho compreende atributos como o *path*, se este corresponde a um ficheiro ou diretoria e se a operação que originou o pedido diz respeito a uma eliminação ou adição da referência. Assim, em vez da alteração dos metadados ocorrer no lado do cliente, o cliente apenas indica que é necessário adicionar ou remover uma determinada referência aos metadados de uma diretoria e essa alteração é realizada no substrato de armazenamento.

Na Figura 13 é possível visualizar o fluxo de operações no iLSFS a partir do momento que é recebida a operação da Interface POSIX de criação de uma nova diretoria. Neste exemplo, o nodo DataFlasks *n1*, antes da operação, apresenta no seu armazenamento a diretoria */home* e o conjunto de metadados associados, no qual se inclui uma referência para um ficheiro denominado *file*. Quando é intercetada a operação *mkdir* é despoletado um pedido de criação da diretoria *docs* e posterior adição desta na

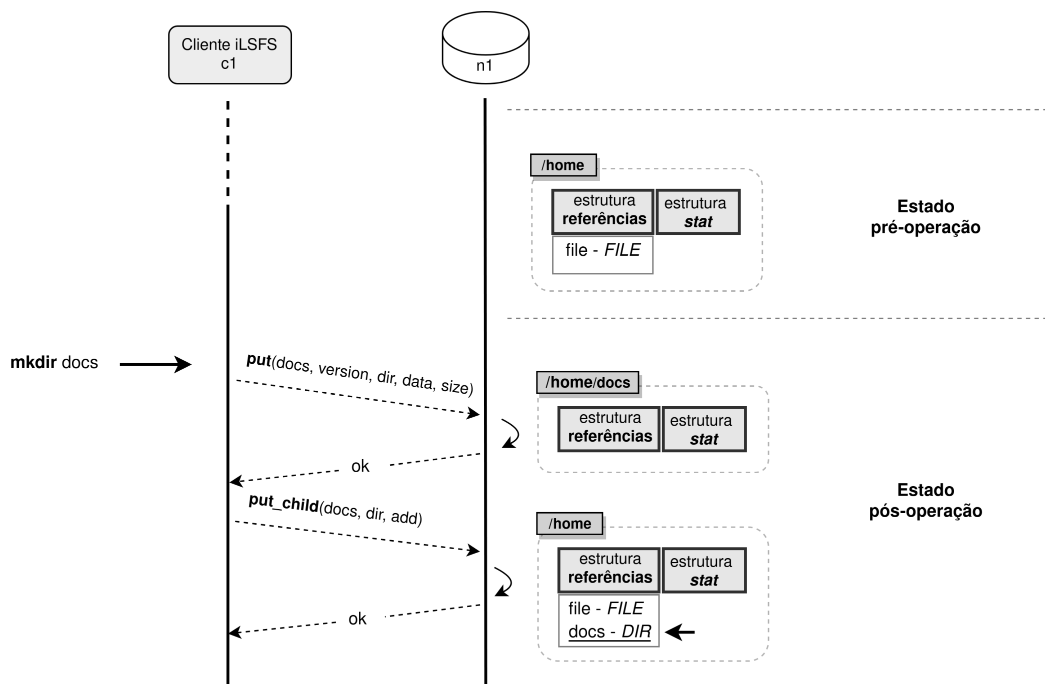


Figura 13: Fluxo de operações responsável pela criação de uma diretoria

diretoria pai, *home*, através da operação *put_child*. No final deste fluxo obtemos no armazenamento do nodo DataFlasks a nova diretoria */home/docs*, bem como a nova referência, */home/docs*, na diretoria */home*.

De seguida, existiu a necessidade da resolução do problema do tamanho dos metadados aquando da sua transmissão pela rede. Com a introdução da operação de atualização dos filhos de uma diretoria, este problema manteve-se apenas nas operações de consulta, uma vez que estas, dependendo da sua finalidade, podem necessitar da totalidade dos metadados. Como tal, e de forma a separar os metadados das diretorias de acordo com as necessidades do sistema de ficheiros, a consulta destes foi dividida em dois conjuntos de operações, um responsável pela solicitação do conjunto total dos metadados e outro pela estrutura *stat*.

Para a transmissão da totalidade do conjunto de metadados que compõem uma diretoria, foi seguida uma abordagem de fragmentação em blocos de tamanho fixo sempre que uma consulta deste gênero fosse requerida ao substrato. No entanto, devido à lógica da gestão dos pedidos implementada é exigido que seja o cliente a solicitar cada um dos fragmentos dos metadados. Por esta razão, esta consulta é composta por duas operações, a operação *get_latest_metadata_size*, seguida da operação *get_metadata_batch*. Na Figura 14 é exemplificado este fluxo necessário para a consulta dos metadados de uma diretoria. A primeira operação, *get_latest_metadata_size*, tem como função perguntar aos nodos de armazenamento o tamanho total do conjunto de metadados relativos à última versão armazenada de uma diretoria. Ao ser recebido este valor torna-se possível saber quantos fragmentos requisitar, através da divisão do valor recebido pelo tamanho fixo dos blocos. Desta forma, com recurso à operação *get_metadata_batch* são solicitados, de uma só vez, os blocos necessários para a reconstrução dos

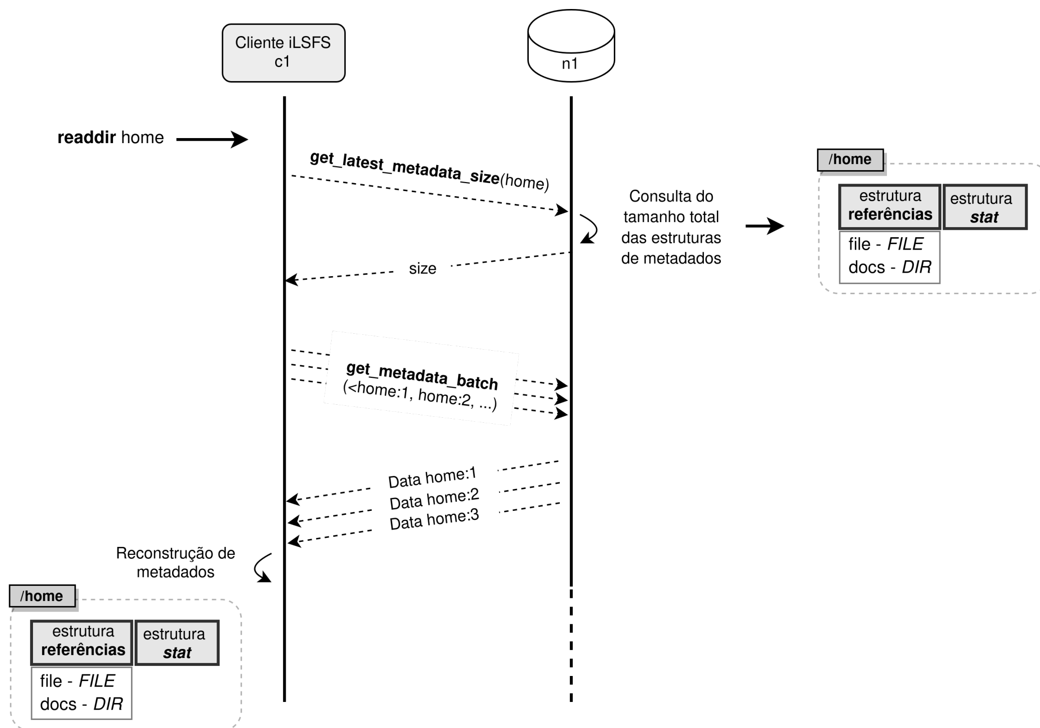


Figura 14: Fluxo de operações na consulta dos metadados de uma diretoria

metadados. Quando finalmente forem obtidos todos os fragmentos o utilizador tem à sua disposição os metadados da diretoria.

Para uniformização do tamanho máximo de blocos transmitidos no iLSFS, foi definido o tamanho máximo de cada fragmento como sendo 4096 Bytes, tal como estabelecido para os blocos de dados de um ficheiro.

Por outro lado existem operações que não necessitam deste conjunto de metadados na sua totalidade, mas de apenas parte, como acontece com a operação de consulta de metadados da interface POSIX - *getattr* - que para a conclusão da operação, basta a obtenção da estrutura *stat*. Com este intuito, foi desenvolvida a operação da API do Cliente *get_latest_metadata_stat* que entrega ao utilizador a estrutura relativa à última versão da diretoria em questão. Como neste pedido estamos apenas a consultar uma estrutura fixa, com tamanho sempre inferior ao tamanho máximo definido, é possível realizar a comunicação destes dados sem a necessidade de fragmentação.

4.7 Armazenamento persistente

A par da organização de dados e metadados seguida pelo iLSFS, também a compreensão da forma como esta informação se encontra organizada no armazenamento dos nodos DataFlasks é relevante. Tendo em conta o versionamento e a possibilidade de existirem versões concorrentes para a mesma chave armazenada, teve de ser seguida uma estratégia que pudesse agrupar todos os dados, aproveitando ao mesmo tempo as propriedades de procura que uma base de dados chave-valor apresenta. A escolha

da chave é, provavelmente, a decisão mais importante na implementação de uma boa estratégia para o alcance de consultas eficientes numa base de dados chave-valor. Como o iLSFS é um sistema de ficheiros, é razoável admitir que a chave deve ser constituída pelo *path* dos ficheiros ou diretorias do sistema. Efetivamente, em todas as procuras relevantes no sistema, a única característica comum diz respeito ao *path* dos dados, tal como é evidenciado nas operações da API do Cliente.

Para além da definição da chave, é também necessário estabelecer o conteúdo que se encontra associado a cada *path*. Com a introdução das operações específicas para diretorias, como a adição ou remoção de uma referência, e a existência de algoritmos, como a junção de diretorias, é útil a separação entre o tipo de dados existentes no sistema de ficheiro. Desta forma, cada *path* tem associado o tipo de dados que representa, conforme seja uma diretoria ou um ficheiro, tal como é apresentado na Figura 15. Esta associação leva-nos também a uma representação diferenciada no que ao objeto de dados armazenado diz respeito.

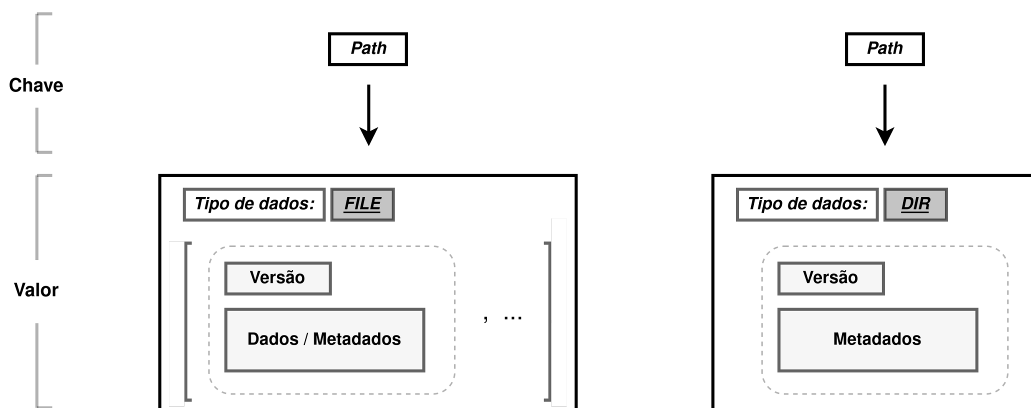


Figura 15: Organização de dados no armazenamento do nodo DataFlasks

Ficheiros

Com a definição de um novo modelo de acesso a dados, cada *path* do sistema de ficheiros pode dispor de mais do que uma versão, sejam concorrentes ou consecutivas. Assim, e de forma a facilitar a escolha de quais os dados a apresentar a um utilizador, as versões de um determinado ficheiro *(File)* são agrupadas no mesmo objeto de dados, em forma de lista. Isto permite que apenas com uma procura seja possível aceder a todas as versões daquela chave e realizar as operações necessárias com um único acesso à base de dados. Adicionalmente, nesta lista de pares versão - dados/metadados, apenas são mantidas em memória persistente as últimas versões concorrentes existentes de cada chave. Ou seja, sempre que uma nova versão chega ao nodo, esta é comparada às presentes e se coincidir com uma versão superior substitui a existente, mas se corresponder a uma versão concorrente, esta é adicionada à lista.

Diretorias

Quando o tipo de dados corresponde a uma diretoria *(Dir)*, apenas existe, em cada nodo, um único par versão - metadados correspondente ao *path* em questão. Em diretorias, versões concorrentes são prontamente unidas através do mecanismo de junção de diretorias, ou apenas atualizadas consoante as operações realizadas sobre elas, logo é suficiente a manutenção de apenas uma versão para representar a diretoria em questão.

4.8 Anti-Entropia

O componente de Anti-Entropia detém uma grande importância na manutenção da coerência do sistema de ficheiros. Com a adição de novas operações e mecanismos no iLSFS, também este teve de se adaptar ao novo paradigma.

Recordando o mecanismo de recuperação de dados do sistema, este encontra-se dividido em duas fases: A fase de operação, correspondente às frequentes comunicações das chaves, que o nodo armazena, aos restantes elementos do seu grupo de replicação, e a fase de recuperação, ativada por um nodo que necessite de obter uma base de dados coerente, quando entra no sistema.

Com a introdução da operação de eliminação de dados, e a respetiva estratégia de *Tombstone* empregue, o mecanismo de recuperação de dados necessita de sofrer alterações de modo a poder incorporar os dados eliminados. Durante a fase de operação, existe um conjunto arbitrário de chaves selecionadas para serem transmitidas pela rede. Este conjunto de chaves, outrora composto na sua totalidade por chaves do armazenamento principal, é agora fragmentado de forma a que parte seja destinado às chaves presentes na Tombstone. Assim, em cada mensagem de anti-entropia enviada, parte do espaço ocupado pelas chaves é reservado para o conjunto de chaves apagadas do nodo. Esta estratégia de designar, em cada mensagem, parte do espaço disponível para anunciar chaves eliminadas permite agilizar o processo de recuperação de dados, no que aos dados apagados se refere, e assim diminuir a probabilidade de que sejam apresentados dados já excluídos aos utilizadores do sistema. No entanto, em qualquer sistema, as operações mais frequentes dizem respeito a inserções ou atualizações dos dados presentes, sendo a eliminação de dados representativa de uma pequena parte do total de operações efetuadas. Deste modo, a divisão do espaço de chaves enviadas não é realizada igualitariamente, mas sim consoante uma percentagem parametrizável, que foi definida em 80% e 20%. Ou seja, em cada iteração do mecanismo de anti-entropia, 80% do conjunto de chaves enviadas é reservado para as chaves do armazenamento principal, enquanto os restantes 20% são referentes às chaves apagadas. Contudo, se a quantidade de chaves normais não preencher a totalidade de espaço reservado, este passa a ser ocupado pelas restantes chaves apagadas, se disponíveis.

Para além disso, realizaram-se também alterações à quantidade de chaves enviadas durante a fase de operação. Em cada iteração do mecanismo de anti-entropia é agora enviada uma percentagem, parametrizável, do total de chaves existentes no sistema, o que permite manipular a quantidade de chaves

enviadas, e avaliar o impacto deste mecanismo na saturação da rede, através da análise do desempenho obtido. Adicionalmente, é também uniformizado o tamanho máximo do pacote do protocolo de comunicação, para as mensagens de anti-entropia, estabelecendo o valor em 4 KiB. Isto significa que o conjunto de chaves selecionado necessita de ser dividido em vários pacotes, em oposição à abordagem tomada pelo LSFS, de anunciar o máximo de chaves possível de uma só vez e num único pacote. O recurso a esta abordagem, é semelhante à seguida nos *dblocks* para a organização de dados no sistema, e tem em conta as características da rede em que o sistema se insere, onde a falha de mensagens é a norma, mas também as propriedades do protocolo UDP utilizado, que no caso de perda de um fragmento do pacote, todo este fica inutilizado. Desta forma, ao diminuirmos a quantidade de dados por mensagem, pretendemos minimizar a perda de mensagens e por conseguinte, que o processo de anti-entropia seja mais eficaz.

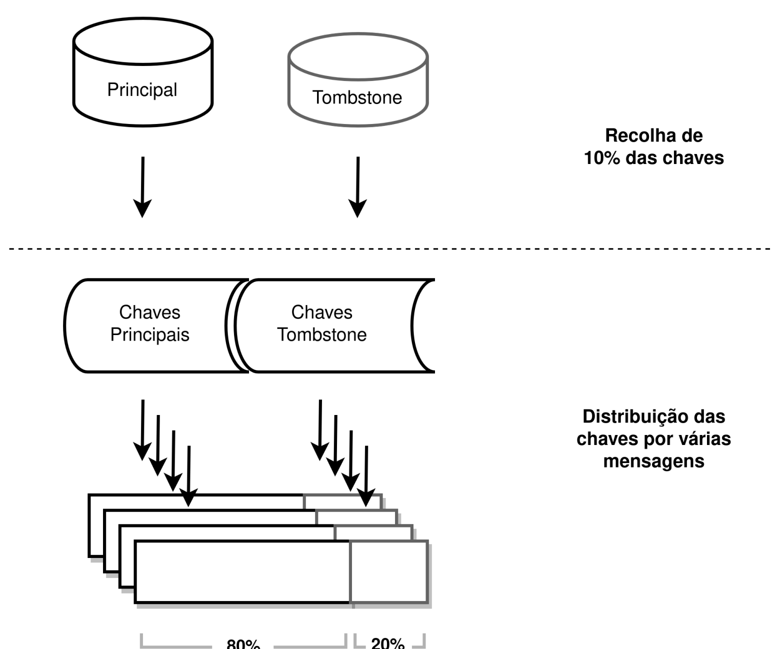


Figura 16: Mecanismo de Anti-Entropia no iLSFS

Na Figura 16 é esquematizado o processo tomada pelo mecanismo de recuperação de dados, desde, a título de exemplo, a seleção de 10% de cada uma das base de dados, até à divisão das chaves selecionadas por várias mensagens de anti-entropia, onde é realizada a divisão descrita anteriormente. Em primeiro lugar, são selecionadas, de cada uma das base de dados (Principal e Tombstone), 10% das chaves, e guardadas temporariamente em dois conjuntos diferentes, para serem disseminadas. De seguida, cada mensagem de anti-entropia, com 4 KiB, retira chaves do conjunto "Chaves Principais", até perfazer, aproximadamente, 80% do tamanho do pacote ($\approx$3277 Bytes), e o restante espaço disponível é ocupado por "Chaves Tombstone". Por fim, cada mensagem é enviada para todos os nodos da vista que fazem parte do grupo de replicação. No caso de um dos nodos não possuir uma das chaves que recebeu por anti-entropia, é realizado uma requisição e posterior envio dos dados associados à chave, até que

a totalidade dos dados da chave seja incorporado. Este processo é repetido consoante um intervalo de tempo parametrizável, por exemplo, a cada 60 segundos.

4.9 Protótipo iLSFS

A implementação do protótipo iLSFS seguiu a base definida pelo sistema de ficheiros LSFS, mantendo como linguagem de desenvolvimento, para todos os seus componentes, o C++. Como camada de indireção das chamadas ao sistema de ficheiros recorre-se à solução em espaço de utilizador FUSE, versão 3.9.1, através da qual é possível a exportação de uma interface POSIX.

No que se refere às comunicações entre as entidades do sistema de ficheiros (Cliente iLSFS, nodo DataFlasks e Bootstrapper), estas são realizadas maioritariamente com recurso ao protocolo da camada de transporte UDP. Assim, cada uma destas entidades dispõe de um servidor assíncrono UDP, implementado com recurso à biblioteca *boost* [5], versão 1.71.0, e que recorre a uma *thread pool* para a gestão e otimização das mensagens processadas. Para as comunicações TCP recorre-se à criação de *sockets* TCP da biblioteca C/C++ *standard library*, sempre que é necessária a realização de comunicações temporárias entre nodos DataFlasks, para a transferência de chaves durante o processo de anti-entropia. Em todas as comunicações pela rede no sistema utiliza-se, na camada de apresentação de ambos os protocolos, o mecanismo de serialização *Protocol Buffers* [36], versão 3.11.4, para o envio de diferentes estruturas de dados. Adicionalmente, é também adotada uma serialização binária nas estruturas de dados armazenadas persistentemente, mediante o recurso à biblioteca *boost*.

Para o armazenamento persistente de dados mantem-se o recurso à base de dados *embedded* chave-valor LevelDB, versão 1.23.0.

5

Avaliação

Definida a arquitetura e apresentadas as principais decisões de implementação no protótipo, é agora relevante a realização de uma avaliação detalhada ao iLSFS. Assim, este capítulo começa por efetuar uma análise abrangente do desempenho a diferentes configurações do sistema, comparando o sistema iLSFS com o sistema que o originou e avaliando cada uma das extensões introduzidas num ambiente de testes mais isolado. No final, e passando para um cenário realista, é examinada a resiliência e escalabilidade do sistema de ficheiros, numa rede de larga escala.

5.1 Desempenho

O iLSFS é um sistema que apresenta diversas configurações e parâmetros que o dotam de uma adaptação a diferentes ambientes de utilização. No cenário em estudo, pretendemos um sistema que possua características de alta disponibilidade, resiliência e escalabilidade, capaz de suportar uma infraestrutura de muita larga escala. No entanto, e como em qualquer sistema totalmente distribuído, é ainda fundamental perceber o desempenho do mesmo. Deste modo, torna-se necessário sujeitar o sistema a um conjunto de cargas de trabalho, com o objetivo de avaliar qual o desempenho máximo que o iLSFS consegue atingir.

5.1.1 Metodologia

De maneira a compreender o conjunto de testes realizados para aferir o desempenho do sistema, é necessário, primeiramente, a apresentação da metodologia seguida. Como tal, é detalhado o ambiente experimental para a execução das experiências realizadas, seguido da especificação do *benchmark* e respetivas cargas de trabalho definidas.

5.1.1.1 Ambiente Experimental

O ambiente experimental para a avaliação de um sistema de ficheiros, como o iLSFS, necessita de apresentar características que lhe permitam simular o contexto real, de uma rede de larga escala, capaz de

suportar uma grande quantidade de nodos de armazenamento. Para o conjunto de testes e configurações definidas na avaliação preliminar de desempenho, recorreu-se a 5 servidores, num ambiente de *cloud* privada. Por forma de uniformizar o ambiente onde os testes são realizados, em cada um dos servidores foi instalado o sistema operativo Ubuntu 20.04. Em relação ao hardware, este é distinto entre os servidores, contudo, podemos agrupá-los em dois conjuntos. O primeiro conjunto compreende quatro servidores, sendo todos estes idênticos, equipados com um processador Intel i5-9500, com 6 CPUs e 16 GiB de RAM. Como armazenamento foi montado um disco NVMe - Samsung SSD 970 Evo Plus, de alta performance, com capacidade de 250 GiB. O segundo conjunto é composto apenas por um servidor com capacidade de processamento superior aos restantes, já que possui um processador Intel i5-10505, com 12 CPUs e 16 GiB de RAM. O armazenamento ficou a cargo de um disco NVMe - PC711 SK hynix, com 256 GiB de memória. Os 5 servidores comunicam entre si através de uma rede interna, sendo a ligação disponibilizada por cada servidor de 10 Gb/s.

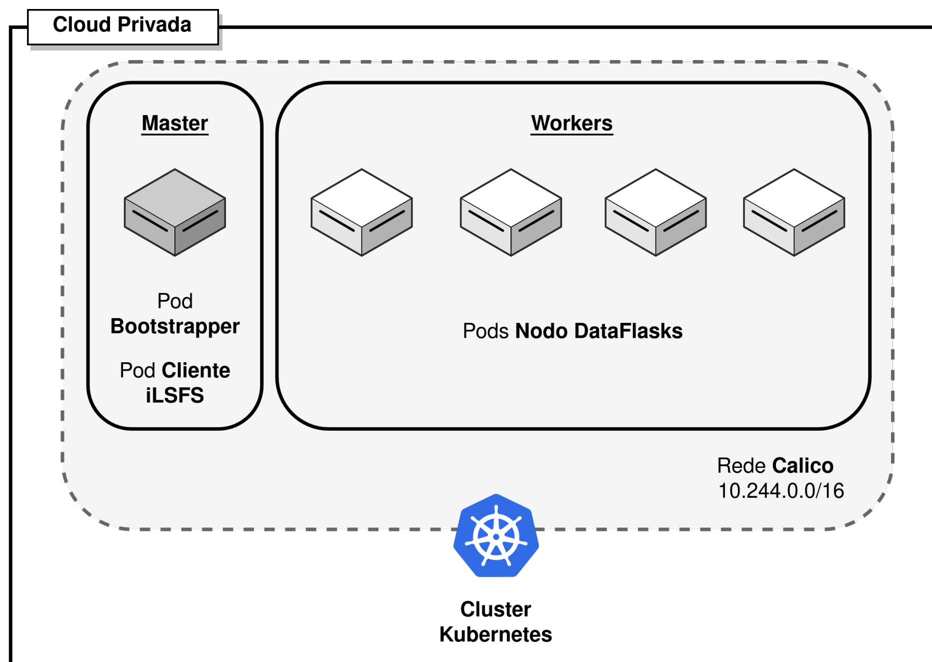


Figura 17: Ambiente experimental - Cloud Privada

Para a gestão dos componentes que compõe o iLSFS recorreu-se à ferramenta de virtualização Docker [2], que possibilita o isolamento através da containerização dos três componentes (Cliente iLSFS, Nodo DataFlasks e Bootstrapper), facilitando todo o processo de instalação e manutenção destes, pela uniformização e portabilidade das imagens criadas. Adicionalmente, toda a orquestração dos containers, automatização de diversos processos manuais e controlo sobre a infraestrutura, como as comunicações entre os componentes na rede, é realizada mediante a implementação de um cluster de Kubernetes [34]. Este cluster é constituído por um conjunto de servidores, ou nodos, que conseguimos agrupar de acordo com as responsabilidades que apresentam. Um dos nodos possui o papel de *Master*, uma vez que tem como função principal a gestão e manutenção do cluster e de todos os seus mecanismos associados. Os

restantes nodos identificam-se como *Workers*, e são apenas responsáveis pelo acolhimento e execução de um conjunto de *Pods* [31], que pretendemos em funcionamento no cluster. Uma *Pod* correspondente à unidade de computação mais simples existente nos Kubernetes, e tem ao seu encargo a gestão de um ou mais containers, tais como os containers Docker que encapsulam os componentes do sistema. Como tal, neste cluster Kubernetes existem três tipos de *Pods*, nomeadamente, a *Pod* Cliente iLSFS, a *Pod* Nodo DataFlasks e a *Pod* Bootstrapper. Na Figura 17 é esquematizado o cluster Kubernetes criado, e a respetiva divisão de *Pods* pelos nodos. O nodo *Master* foi alocado ao servidor que apresenta uma capacidade de processamento superior, entre os disponíveis. Já os restantes nodos *Workers* correspondem aos 4 servidores que possuem características de hardware iguais. De forma a libertar recursos e possibilitar a melhor divisão possível das *Pods* pelos nodos do cluster alteramos a configuração base do nodo *Master*, de modo a que, para além das suas funções principais, pudesse também encontrar-se disponível para acomodor *Pods*. Assim, o nodo *Master* ficou responsável pelas *Pod* Cliente iLSFS e *Pod* Bootstrapper, possibilitando a libertação de recursos para os *Workers*, que acomodam as várias *Pods* Nodo DataFlasks.

Cada *Pod* no cluster é uma abstração de uma máquina e, como tal, necessita de realizar comunicações com as restantes *Pods* existentes. Desta forma, no cluster é ainda adicionada uma rede sobreposta Calico [35], que permite a uniformização e atribuição de endereços IP virtual a todas as *Pods*, viabilizando a sua comunicação, mesmo quando as *Pods* se encontram alocadas em nodos diferentes, através do roteamento do tráfego entre as máquinas da rede no cluster.

Em cada uma das *Pods* criadas neste ambiente é ainda realizada uma monitorização aos recursos consumidos, através da ferramenta *dstat*, que permite a recolha de métricas relevantes, como a utilização de CPU, RAM ou rede. Esta informação é depois utilizada, quando pertinente, na análise de resultados para auxiliar a avaliação do sistema.

Para a automação, a simplificação e a organização de diversas tarefas, como a construção do cluster, monitorização das *Pods*, execução dos testes de carga e respetiva recolha de resultados, recorre-se a uma combinação de *scripts* Ansible [4] e *Bash*.

5.1.1.2 Cargas de trabalho

Na avaliação de desempenho do iLSFS definiram-se dois conjuntos de testes (micro e macro), responsáveis por analisar diferentes parâmetros do sistema, onde se incluir as diversas otimizações introduzidas para conseguir atingir os objetivos definidos. A realização de uma análise a estes parâmetros permite que seja possível encontrar qual a conjugação de valores que melhor contribuem para a extração do desempenho máximo no sistema de ficheiros.

Benchmark Para este conjunto de experiências recorreu-se à ferramenta de *benchmark*, Filebench [50], que nos permite definir um conjunto de cargas de trabalho, especificando e isolando as operações que pretendemos analisar, através da sua linguagem *Workload Model Language*. Esta ferramenta é vastamente utilizada na injeção de operações em sistemas de ficheiros, permitindo simular todo o tipo de

operações I/O, com uma grande customização, seja na quantidade de processos ou *threads* que pretendemos em execução, ou na quantidade e tamanho de ficheiros ou diretorias presentes na carga de trabalho. Desta forma, conseguimos, facilmente, manipular os níveis de stress a que o sistema de ficheiros pode estar sujeito.

Micro testes O conjunto de testes micro, corresponde a experiências que possibilitam a realização de uma avaliação singular às operações efetuadas, isolando-as, para uma análise individual em cada um dos testes. Assim, foi definido um conjunto de operações vastamente executadas em qualquer sistema de ficheiros. Começamos por estabelecer as operações mais recorrentes sobre dados, onde se inclui a escrita e leitura de dados em ficheiros. Estas operações, num sistema de ficheiros, podem ocorrer tanto de uma forma sequencial, como aleatória, tendo diferentes implicações na lógica seguida pelo sistema. No cenário de IoT, em que o iLSFS se insere, as escritas de dados decorrem maioritariamente de forma sequencial, em oposição às leituras, que se realizam tanto de modo sequencial como aleatória pelo ficheiro. Assim definiram-se três testes sobre dados: escrita sequencial, leitura sequencial e leitura aleatória. Para estas cargas de trabalho analisou-se o impacto do tamanho do bloco no desempenho geral do sistema, variando este valor entre 4 KiB e 128 KiB. A escolha do valor de 4 KiB deve-se à sua correspondência com o tamanho máximo de bloco definido para o iLSFS, enquanto o valor 128 KiB corresponde ao tamanho máximo de bloco que é possível requerer ou inserir, de uma só vez, pela camada FUSE.

Para além da realização de testes sobre dados, também as operações sobre metadados necessitam de ser alvo de avaliação. Definiu-se então um conjunto de micro testes correspondentes à criação e eliminação de ficheiros, mas também à consulta dos metadados de ficheiros. A análise destas operações permite perceber qual o comportamento do sistema, a nível do seu desempenho, quando operações sobre metadados de diretorias e ficheiros são realizadas, com especial destaque para a nova operação de eliminação de dados introduzida no iLSFS. Este conjunto de testes apresenta um número de operações, previamente definido e estabelecido na criação, eliminação ou consulta de 75000 ficheiros, distribuído por 300 diretorias. Com estas cargas de trabalho, analisou-se qual a influência que o recurso, ou não, a uma cache de metadados, tem no desempenho do iLSFS.

Macro testes O conjunto de testes macro, corresponde à inserção do sistema num cenário mais realista, onde o conjunto de operações a que este é sujeito se aproxima do caso de estudo. Como tal, recorreu-se a um conjunto de testes vastamente utilizados na avaliação de sistemas de ficheiros, simulando cargas realistas de sistemas, como é o exemplo de um servidor de ficheiros (operações, maioritariamente, de escrita de dados), um servidor web (operações, maioritariamente, de leitura de dados) e um servidor de e-mail (um número idêntico de operações de escrita e leitura). No contexto em que o iLSFS se insere, a carga de trabalho que mais se aproxima de um cenário real é a de um servidor web, no qual é injetada uma carga intensiva de leituras sobre ficheiros, simulando a leitura para processamento dos dados recolhidos por dispositivos IoT.

De realçar que determinadas cargas de trabalho, como a leitura de dados, eliminação e consulta de metadados de ficheiros e a servidor web, são sempre precedidas de um povoamento de dados. No entanto, este povoamento não tem qualquer influência na avaliação de desempenho apresentada, sendo apenas motivo de análise as operações que ocorrem à posteriori.

As cargas de trabalho, quanto ao seu período de execução, dividem-se em dois grupos: Os micro testes sobre dados e os testes macro, ocorrem durante um período de 15 minutos, realizando, de forma exaustiva, o maior número de operações possível; Já os testes micro sobre metadados decorrem até à conclusão do número de operações definido, ou até ser ultrapassado o tempo máximo de 15 minutos. Em todas as cargas de trabalho recorre-se a apenas uma *thread* para a execução dos testes.

5.1.1.3 Configurações em análise

Complementarmente a esta avaliação de desempenho é ainda realizada uma análise comparativa à estruturação do sistema e ao desempenho que cada uma das configurações definidas apresenta, expondo vantagens e limitações de cada. Esta variação na estruturação ocorre ao nível dos nodos DataFlasks, ou seja, tanto na quantidade de nodos existentes no sistema, como na configuração de grupos, existindo sempre, no entanto, um Cliente iLSFS e um Bootstrapper. Para além disso, e de forma a avaliarmos o impacto que fatores externos têm no sistema, é ainda realizada uma comparação entre o sistema, onde todos os seus componentes se encontram em execução numa só máquina e comunicam através de uma rede *loopback* local (*localhost*), e o sistema num modelo distribuído, onde todos os componentes comunicam pela rede.

Assim, definimos as seguintes configurações para análise:

Cenário Micro

<u>Local</u> ⟶ 1 nodo DataFlasks - 1 grupo

<u>Distribuído</u> ⟶ 1 nodo DataFlasks - 1 grupo

<u>Distribuído</u> ⟶ 2 nodos DataFlasks - 1 grupo

<u>Distribuído</u> ⟶ 2 nodos DataFlasks - 2 grupos

Cenário Macro

<u>Distribuído</u> ⟶ 50 nodos DataFlasks - 16 grupos

A análise da evolução destas configurações do iLSFS, permite-nos avaliar como a divisão em grupos e a distribuição da carga por mais do que um nodo influencia o desempenho do sistema. As configurações são divididas consoante o cenário em que se inserem, no que à quantidade de nodos se refere, podendo ser encaixadas no cenário micro ou no cenário macro. De realçar que a configuração "Distribuído ⟶ 1 nodo - 1 grupo", apesar de apenas apresentar 1 nodo de armazenamento é considerada distribuída porque

a comunicação entre o cliente e o armazenamento é realizada pela rede remotamente. A configuração do cenário macro apresenta-se como a mais realista, em termos de escalabilidade, permitindo-nos obter resultados mais objetivos e próximos do que o iLSFS pode atingir num cenário real.

5.1.2 Resultados

Nesta secção são apresentados os resultados obtidos nas diversas cargas de trabalho e variações de configurações, às quais o sistema de ficheiros iLSFS foi sujeito, e que nos permite aferir o seu desempenho.

Os resultados obtidos são uma combinação da execução de cada carga de trabalho 3 vezes, sendo apresentado como resultado o débito médio alcançado. A redundância nos testes pretende evitar que determinadas combinações de fatores externos influenciem positiva ou negativamente os resultados exibidos. Como tal, é também apresentado o desvio padrão, resultante dos testes realizados.

Todas as cargas de trabalho efetuadas são comparadas com um sistema de ficheiros local, o *Passthrough*, que recorre ao FUSE, como camada de indireção, para copiar o sistema de ficheiros nativo e repetir as operações que nele são efetuadas. Os resultados obtidos servem como referência do melhor desempenho que podemos alcançar, ao utilizarmos a ferramenta *userspace* FUSE, para a interceção das chamadas ao sistema de ficheiros. Complementarmente, é ainda realizada uma comparação entre o sistema desenvolvido e o sistema LSFS, a fim de compreender o impacto que as funcionalidades introduzidas têm no desempenho do sistema de ficheiros. Esta comparação é apenas realizada num conjunto restrito de cargas de trabalho, devido às limitações presentes no LSFS, que o impedem de suportar testes intensivos sobre metadados de diretorias, como é o caso das cargas de trabalho de criação de ficheiros ou de consulta de metadados de ficheiros, que na sua composição requerem a manipulação de uma grande quantidade de metadados de diretorias. Para além disso, a eliminação de ficheiros também não pode ser avaliada no LSFS, pois a operação de eliminação não se encontra disponível. Deste modo, apenas nas cargas de trabalhos micro de leitura e escrita de ficheiros é possível efetuar a análise comparativa com o LSFS.

Adicionalmente, todas as cargas de trabalho em análise realizam-se com a diretiva *"Direct I/O"* ligada. Esta diretiva permite evitar que as operações de leitura e escrita sobre um ficheiro utilizem a *page cache* do sistema operativo, obrigando a que todas estas sejam direcionadas da aplicação para o armazenamento do sistema. Recorre-se a esta diretiva para percebermos o impacto de todas as cargas de trabalho definidas no iLSFS, sem a influência positiva das caches do sistema operativo.

5.1.2.1 Cenário Micro

Antes de procedermos à análise das diversas experiências efetuadas para a avaliação do desempenho do iLSFS, é relevante enunciar os diferentes parâmetros e configurações dos componentes utilizados durante a execução das cargas de trabalho. Alguns destes parâmetros serão ativados ou desligados em algumas experiências, de modo a percebermos o impacto que estes têm no desempenho do sistema e nas configurações estruturais, de menores dimensões.

Neste conjunto de **micro testes** estabelecemos que o Cliente iLSFS recorre ao balanceador de carga *Random* para a distribuição dos pedidos pelos nodos de armazenamento DataFlasks, e a uma paralelização de escritas e leituras, descrita na Secção 2.3.5, igual ao tamanho do bloco no sistema, que se encontra fixado em 4 KiB. Já para a disseminação de mensagens na rede, por intermédio do mecanismo de SPS, é definido um intervalo de troca de mensagens de 15 segundos entre os nodos de armazenamento, e de 5 segundos no cliente. Adicionalmente, para este conjunto de testes de menor escala, não é ativado o mecanismo de anti-entropia nos nodos DataFlasks, permitindo uma análise mais isolada, sem a influência deste mecanismo no desempenho geral do iLSFS. De notar que a mesma configuração foi usada também no sistema LSFS.

Escrita de ficheiros Começamos por analisar o comportamento das diversas configurações definidas, quando são realizadas operações sobre dados ao sistema de ficheiros. De realçar que nestas experiências, definiu-se o tamanho de bloco escrito por cada operação, também denominado Tamanho I/O nas cargas de trabalho do Filebench, com o valor de 4 KiB, igual ao tamanho de bloco no sistema desenvolvido.

Na Tabela 2 é possível visualizar o débito obtido, quando os sistemas, em diversas configurações, são sujeitos a uma carga de trabalho de escritas sequencias.

<table>
<tr><th colspan="3" rowspan="2"></th><th colspan="2">Débito (MiB/s)</th></tr>
<tr><th>Média ($\overline{x}$)</th><th>Desvio padrão (σ)</th></tr>
<tr><td colspan="3">Passthrough</td><td>136.33</td><td>29.56</td></tr>
<tr><td rowspan="4">LSFS</td><td colspan="2">Local</td><td>32.96</td><td>0.40</td></tr>
<tr><td rowspan="3">Distribuído</td><td>1 nodo - 1 grupo</td><td>9.5</td><td>1.15</td></tr>
<tr><td>2 nodos - 1 grupo</td><td>18.13</td><td>1.55</td></tr>
<tr><td>2 nodos - 2 grupos</td><td>7.33</td><td>0.32</td></tr>
<tr><td rowspan="4">iLSFS</td><td colspan="2">Local</td><td>17.53</td><td>0.06</td></tr>
<tr><td rowspan="3">Distribuído</td><td>1 nodo - 1 grupo</td><td>6.53</td><td>0.47</td></tr>
<tr><td>2 nodos - 1 grupo</td><td>12.63</td><td>1.28</td></tr>
<tr><td>2 nodos - 2 grupos</td><td>5.07</td><td>0.06</td></tr>
</table>

Tabela 2: Resultados Filebench - Escritas sequenciais em diversas configurações

Comparando os valores de desempenho, entre o sistema de referência *Passthrough* e a configuração local do iLSFS, percebemos que o nosso sistema de ficheiros apresenta apenas 13% do desempenho máximo que é possível obter, em escritas, num sistema de ficheiros local. Claro que esta comparação não é a mais adequada, uma vez que o iLSFS é constituído por diversas camadas, preparadas e otimizadas para um cenário distribuído, existindo mudanças de contexto que prolongam a execução das tarefas. Porém, através desta conseguimos perceber o impacto, de uma forma superficial, dos mecanismos presentes no iLSFS.

Focando-nos apenas no sistema desenvolvido, conseguimos acompanhar o comportamento do mesmo quando testadas diversas configurações na sua estrutura de grupos. Ao analisarmos o débito obtido, entre a configuração local do sistema e a sua correspondente, no modelo distribuído, é possível notar a existência de um decréscimo de mais de metade no desempenho, de 17.53 MiB/s para 6.53 MiB/s. Esta diferença de desempenho, já era espectável e pode ser explicada pela introdução da rede ao passarmos para o modelo distribuído. No modelo local, as comunicações nunca chegam a sair para um ambiente externo, uma vez que recorrem à rede *loopback*, para o efeito. No caso do modelo distribuído, todas as comunicações, entre componentes, são efetuadas de forma remota, necessitando de transitar pela rede, entre máquinas, o que para as escritas sequenciais em ficheiros significa uma redução de 63% no débito obtido.

Analisando o cenário distribuído do iLSFS, foram realizadas experiências para 3 configurações diferentes, que nos permitem analisar diferentes aspetos e características do sistema. Entre a configuração com "1 nodo e 1 grupo", e a de "2 nodos e 1 grupo", encontramos um aumento de desempenho, que praticamente dobra o débito de dados escritos, e pode ser explicado, naturalmente, pela existência de mais um nodo capaz de atender pedidos. Existir dois nodos do mesmo grupo proporciona uma distribuição da carga que evita que pedidos fiquem em fila de espera, agilizando o seu processamento, já que cada nodo tem uma carga menor. Contudo, mais um nodo capaz de responder a pedidos do cliente, nem sempre significa um melhor desempenho. Comparando a configuração "2 nodos e 1 grupo", com a "2 nodos e 2 grupos", é possível visualizar um decréscimo acentuado do débito de escritas na configuração com 2 grupos, passando dos 12.63 MiB/s para os 5.07 MiB/s. Nesta configuração existe uma divisão do espaço de endereçamento em metade, reduzindo a quantidade de chaves que cada nodo pode armazenar. Para além disso, apenas um nodo pode responder corretamente ao pedido efetuado, tal como acontece na configuração de "1 nodo e 1 grupo", porém possuirmos dois grupos diferentes, com um balanceador de carga *Random*, faz com que não haja uma distribuição correta dos pedidos para os nodos capazes de os processar. Desta forma, grande parte dos pedidos são enviados para o nodo errado, que por sua vez tem de os redirecionar para o nodo certo, o que aumenta o número de saltos na rede, e aumenta o tempo de resposta ao pedido, impactando negativamente o desempenho no sistema.

Comparando ainda o sistema LSFS, com o protótipo iLSFS, é visível um decréscimo do débito de escritas entre os dois sistemas, em todas as configurações. Focando-nos na configuração local, o iLSFS consegue obter 53% do desempenho alcançado pelo LSFS na mesma configuração. Esta diminuição para metade do desempenho, nas escritas, é justificada com as funcionalidades introduzidas no iLSFS, que adicionam uma camada de complexidade às operações realizadas. O maior impacto encontra-se no nodo DataFlasks: No iLSFS a escrita de dados é efetuada recorrendo a uma estruturação bem definida, já detalhada na Secção 4.7, do Armazenamento persistente, e a uma posterior serialização de dados para armazenamento; Já no LSFS, este processo é muito mais simples e consiste apenas na inserção direta dos dados na base de dados. Esta complexidade no iLSFS é estritamente necessária e lógica, e permite uma organização dos dados mais eficiente pelas suas versões, contudo tem um certo impacto no tempo de processamento de cada operação, que influência negativamente o desempenho do sistema

nas escritas de ficheiros.

Ao analisarmos as restantes configurações, a diferença de desempenho entre os dois sistemas já não é tão elevada, uma vez que a rede passa a ter a maior influência no tempo de resposta aos pedidos. Aqui, o sistema desenvolvido passa a apresentar entre 69% e 70% do débito do sistema LSFS.

Leitura de ficheiros Depois de analisado o comportamento do sistema, no que se refere às escritas em ficheiros, é então importante a realização da mesma análise às leituras. Tal como as escritas sequencias, também nesta carga de trabalho foi estabelecido o tamanho de bloco pedido por cada operação de leitura, durante a carga de trabalho, em 4 KiB, igual ao tamanho do bloco no iLSFS.

Nas Tabelas 3 e 4 são apresentados os valores de desempenho alcançados pelo iLSFS, durante a leitura sequencial e aleatória de um ficheiro.

			Débito (MiB/s)	
			Média ($\overline{x}$)	**Desvio padrão** (σ)
Passthrough			512.1	5.86
LSFS	**Local**		29.87	2.96
	Distribuído	1 nodo - 1 grupo	12.9	3.39
		2 nodos - 1 grupo	22.0	0.8
		2 nodos - 2 grupos	8.43	1.33
iLSFS	**Local**		24.9	1.55
	Distribuído	1 nodo - 1 grupo	8.9	3.97
		2 nodos - 1 grupo	19.57	0.23
		2 nodos - 2 grupos	5.7	0.17

Tabela 3: Resultados Filebench - Leituras sequenciais em diversas configurações

Podemos observar que o sistema iLSFS local apresenta aproximadamente 5% do desempenho obtido pelo *Passthrough* nas leituras sequenciais e 4% nas leituras aleatórias. Olhando apenas para as configurações do iLSFS, observa-se também, um decréscimo acentuado no débito entre o cenário local e o distribuído "1 nodo e 1 grupo", em ambos os tipos de leituras, pela introdução da rede nas comunicações entre componentes.

Analisando as três configurações distribuídas é possível notar um padrão idêntico tanto na leituras sequenciais, como nas leituras aleatórias, e um desempenho também bastante similar. Usualmente, as leituras sequencias tendem a ter um desempenho bastante superior às leituras aleatórias, uma vez que o modelo sequencial toma vantagem de mecanismos como o *readahead* e a *page cache*. O *readahead* é um mecanismo bastante utilizado quando o sistema de ficheiros nota um padrão de acesso a dados linear, e ao invés de apenas requerer a quantidade de dados necessária, acede a dados subsequentes (no que aos blocos de dados diz respeito), prontamente guardados em caches do sistema operativo. Isto

			Débito (MiB/s)	
			Média ($\overline{x}$)	**Desvio padrão** (σ)
Passthrough			497.63	3.59
LSFS	**Local**		30.03	1.65
	Distribuído	1 nodo - 1 grupo	14.0	0.95
		2 nodos - 1 grupo	19.83	2.12
		2 nodos - 2 grupos	7.57	0.95
iLSFS	**Local**		22.23	0.35
	Distribuído	1 nodo - 1 grupo	7.17	1.50
		2 nodos - 1 grupo	19.23	0.21
		2 nodos - 2 grupos	5.97	0.06

Tabela 4: Resultados Filebench - Leituras aleatórias em diversas configurações

quer dizer que, se, posteriormente, forem pedidos dados, e se estes se encontrarem na *page cache*, os acessos realizados são extremamente rápidos, uma vez que evitamos acessos penalizadores a memória persistente. Nas leituras aleatórias, a utilização de mecanismos como os descritos, não trazem tantas vantagens, já que o acesso aos dados são realizados de forma totalmente aleatória, o que faz com que exista uma baixa probabilidade de aceder a dados previamente carregados para cache. Porém, nesta avaliação de desempenho, o uso da *page cache* é ignorado, por meio da diretiva *"Direct I/O"*, o que significa que não existem benefícios significativos entre a realização de leituras sequenciais comparativamente às aleatórias, e por esse motivo obtemos débitos bastante similares.

Verifica-se ainda um incremento acentuado quando passamos da configuração "1 nodo e 1 grupo", para "2 nodos e 1 grupo". Este incremento, próximo do dobro, pode ser explicado pela divisão de carga ao configurarmos o sistema com 2 nodos capazes de processar os mesmos pedidos do cliente, como já foi explicado nas escritas de dados, apresentando o sistema um comportamento idêntico tanto nas escritas como nas leituras de dados.

Ao compararmos o sistema iLSFS com o sistema LSFS, podemos observar uma diminuição ligeira do desempenho, tanto na configuração local, como na distribuída. Na configuração local, em escritas sequenciais, é alcançado um débito de 24.9 MiB/s no iLSFS, representativo de 83% do débito no sistema LSFS, com 29.87 MiB de dados lidos por segundo. Esta diferença, semelhantemente ao constatado nas escritas, deve-se às funcionalidades adicionadas no protótipo disponibilizado. Em leituras, o iLSFS, para a obtenção de uma resposta, necessita de realizar o processo contrário às escritas, e desserializar a informação da base de dados, para aceder aos dados e entregar ao utilizador a versão mais recente disponível. Já no LSFS, as leituras são realizadas com recurso a uma funcionalidade singular da *LevelDB*, de procura por prefixo da chave, o que, no caso de uma correspondência elevada entre as chaves e o prefixo, faz com que as procuras sejam demoradas, tornando-se um ponto crítico para o desempenho

do sistema. No caso desta carga de trabalho, o impacto da procura por prefixo não é elevado, por haver poucos dados em cada nodo, o que resulta na aparência de um maior impacto no débito pelo processo de desserializar, porém, é esperado que ao aumentarmos a quantidade de dados a nossa abordagem porporcione um desempenho superior nas leituras sobre dados, em relação ao sistema LSFS.

Ao avaliarmos o modelo distribuído, a diminuição no débito entre os dois sistemas é menos acentuada, já que a rede passa a ter uma maior perturbação no desempenho, e a complexidade introduzida no iLSFS deixa de ter tanta influência no tempo de resposta em cada pedido efetuado.

Criação e eliminação de ficheiros Após a realização de uma análise ao comportamento do sistema desenvolvido, quando sujeito a cargas de trabalhos sobre dados, é fundamental a concretização de uma avaliação semelhante a operações sobre os metadados do sistema, como a criação ou eliminação de ficheiros. As operações de criação de ficheiros, compreendem a realização de criação do ficheiro, propriamente dito, a escrita de dados para o mesmo, e o fecho do descritor, previamente aberto durante a criação do ficheiro. As operações de eliminação de ficheiros apresentam menos passos, e apenas chamam a operação de eliminação do ficheiro. Nas Tabelas 5 e 6 são apresentados os valores de débito, em operações por segundo (Ops/s), obtido durante a execução das cargas de trabalho de criação e eliminação de ficheiros, respetivamente. Ainda aqui é realizada uma análise comparativa entre o recurso ou não (on e off) à cache de metadados desenvolvida no sistema de ficheiros, de modo a compreender o impacto que esta tem no desempenho do sistema. Como foi descrito na arquitetura do iLSFS, a cache encontra-se dividida em duas, sendo uma a cache de ficheiros abertos e outra a cache de diretorias. Esta última apresenta dois parâmetros base, o tamanho da cache e o tempo de atualização. O tamanho da cache estabeleceu-se nas 1000 entradas de diretorias, recentemente acedidas pelo utilizador e o tempo de atualização definiu-se como sendo 1 segundo. Ambos os parâmetros foram definidos consoante valores realistas, especialmente o tempo de atualização, simulando um cenário intensivo de vários utilizadores a manipular na mesma diretoria, o que requer uma atualização da cache regular, para a apresentação dos metadados mais atuais.

À primeira vista, reparamos que a diferença de desempenho obtido, entre o sistema de referência *Passthrough* e ou iLSFS local, sem ou com cache ativada, é elevada. No iLSFS, as operações sobre metadados, como a criação ou eliminação de ficheiros apresentam uma maior complexidade e são mais intensivas, no que se refere ao processamento, do que as operações similares num sistema de ficheiros tradicional, ou mesmo, comparativamente às operações sobre dados no sistema desenvolvido. Isto deve-se, naturalmente, às características de coerência eventual que necessitam de mecanismos adaptados, como a lógica de metadados implementada, para o normal funcionamento do sistema de ficheiros. Assim, é normal que o débito atingido pelo sistema de referência seja bastante superior ao que o iLSFS consegue alcançar.

Analisando os resultados da carga de trabalho de criação de ficheiros, é possível perceber como a utilização de uma cache de metadados beneficia o desempenho no sistema, em qualquer das configurações definidas. No sistema local obtemos uma subida de 430% no débito de operações realizadas por

			Cache	Débito (Ops/s)	
				Média ($\overline{x}$)	Desvio padrão (σ)
Passthrough			-	8267.66	1234.23
iLSFS	Local		off	115.43	0.05
			on	496.67	4.39
	Distribuído	1 nodo - 1 grupo	off	152.97	1.82
			on	579.09	14.49
		2 nodos - 1 grupo	off	159.33	3.61
			on	676.08	21.83
		2 nodos - 2 grupos	off	163.63	2.61
			on	616.04	14.18

Tabela 5: Resultados Filebench - Criação de ficheiros em diversas configurações

			Cache	Débito (Ops/s)	
				Média ($\overline{x}$)	Desvio padrão (σ)
Passthrough			-	3032.44	159.54
iLSFS	Local		off	50.8	0.05
			on	151.3	0.77
	Distribuído	1 nodo - 1 grupo	off	71.83	0.15
			on	236.51	8.06
		2 nodos - 1 grupo	off	72.56	0.17
			on	259.8	10.82
		2 nodos - 2 grupos	off	73.22	0.41
			on	260.05	10.0

Tabela 6: Resultados Filebench - Eliminação de ficheiros em diversas configurações

segundo, de 115.43 Ops/s para 496.67 Ops/s, representado numa clara vantagem do recurso à cache. Nesta carga de trabalho, a maior parte das operações de consulta sobre metadados, tanto de diretorias, como de ficheiros, dirige-se apenas à cache para a recolha da informação pretendida, o que faz com que muitos dos pedidos efetuados nunca cheguem aos nodos de armazenamento. Isto significa que nunca existe uma mudança de contexto entre o Cliente iLSFS e o Nodo DataFlasks, nem penalizações da rede, aumentando amplamente o desempenho obtido, como é verificado. Observa-se o mesmo na carga de trabalho de eliminação de dados. Na configuração local, existe um incremento de aproximadamente 300%, quando a cache não é utilizada, comparativamente a quando é utilizada, e o mesmo se constata nas restantes configurações distribuídas.

Para além disso, verifica-se uma ligeira diferença no padrão do desempenho obtido em ambas as cargas de trabalho, quando comparamos o modelo iLSFS local, com o modelo iLSFS distribuído. Na avaliação das experiências sobre dados, obtivemos constantemente um débito na configuração local substancialmente superior à configuração distribuída. Em contrapartida, nas operações de criação e eliminação de ficheiros, existe um ligeiro aumento do débito no iLSFS distribuído, comparativamente ao local, quer quando a cache de metadados se encontra ligado ou desligada. Este incremento pode ser explicado pelas características da configuração local, juntamente com a natureza das operações sobre metadados no iLSFS. Durante a configuração local, a máquina responsável por alojar este cenário necessita de ter em execução, em todo o momento, todos os componentes que compõe o sistema, nomeadamente, um Cliente iLSFS e um Nodo DataFlasks, aos quais juntamos ainda a execução da carga de trabalho do Filebench. Este conjunto de processos em atividade ao mesmo tempo resulta numa carga maior exercida sobre a máquina, do que a configuração distribuída, onde existe uma clara divisão física dos componentes entre várias instâncias. Adicionalmente, as operações sobre metadados são mais prolongadas no seu processamento e intensas, do que as operações sobre dados, o que sujeita o sistema a um stress maior, impactando negativamente o desempenho obtido no modelo local. Desta forma, o desempenho atingido pelo modelo distribuído consegue superar o local.

Consulta de metadados de ficheiros A par da avaliação das operações que realizam alterações ao estado dos metadados no sistema de ficheiros, é necessário perceber o comportamento do sistema perante a consulta de metadados, onde o recurso a uma cache pode ter uma influência ainda mais elevada nos resultados de desempenho alcançados. Assim, na Tabela 7 são apresentados os resultados da carga de trabalho de consulta de metadados, sobre diversas configurações.

Observando o débito de operações por segundo, atingido pelo sistema desenvolvido no cenário local, encontramos um salto de desempenho muito superior, comparativamente às outras operações sobre metadados, quando analisamos a diferença entre a cache ligada ou desligada. Neste cenário, o recurso a uma cache de metadados permite ao iLSFS um aumento de 78.31 Ops/s para 3057.13 Ops/s, o que representa um aumento de 3900% nas operações realizadas a cada segundo. Este aumento substancial, deve-se maioritariamente ao tipo de carga de trabalho executada que tira o máximo proveito das caches de metadados implementadas no sistema. A consulta de metadados é uma carga de trabalho intensiva que requer, repetidamente, ao sistema de ficheiros os metadados, tanto de diretorias, como dos ficheiros existentes no sistema. Com a utilização das caches de metadados presentes no Orquestrador do Cliente iLSFS, a maioria dos pedidos realizados são prontamente respondidos, sem a necessidade de obtenção da resposta remotamente aos Nodos DataFlasks. Por outro lado, ao removermos as caches do cenário em teste, todos os pedidos efetuados, sem exceção, necessitam de chegar aos nodos de armazenamento, para a obtenção de uma resposta, o que implica uma penalização no desempenho, devido à rede e, consequentemente, um débito de operações baixo.

Comparando o modelo distribuído com o modelo local, é notória a dualidade de resultados que obtemos entre o recurso ou não à cache. Quando passamos do modelo local para distribuído, com a cache

desligada, é visível um incremento no desempenho do iLSFS, tal como ocorre nas operações de metadados de criação e eliminação, explicado pelas características do iLSFS local, e intensidade das operações sobre metadados. De maneira oposta, existe um decréscimo do débito obtido quando passamos do modelo local para distribuído e a cache se encontra ligada. Com a utilização da cache, e sendo a maioria dos pedidos satisfeitos ainda no cliente, procuramos tentar perceber o porquê de o desempenho diminuir. Uma das características da cache, mais concretamente da cache de diretorias do iLSFS, corresponde à necessidade de realização de uma atualização dos valores armazenados, para que, a todo o momento, se tenha uma vista dos metadados o mais atual possível. Definido um tempo de atualização de 1 segundo, todas as diretorias presentes na cache, são prontamente atualizadas de 1 em 1 segundo, no entanto, cada uma necessita de ficar temporariamente bloqueada enquanto a consulta é realizada no armazenamento remoto. Esta característica impede que, em determinados momentos, certos pedidos ao sistema obtenham uma resposta instantânea, sendo obrigados a permanecer em espera, uma vez que a diretoria em questão pode estar em processo de atualização. No modelo local, esta característica detém um impacto muito reduzido, uma vez que as atualizações são realizadas rapidamente, e as comunicações são efetuadas todas com recurso à rede *localhost*. Porém, no modelo distribuído, com todas as comunicações a necessitarem de aceder remotamente ao armazenamento, existe uma penalização da rede, que prolonga o tempo de atualização dos metadados da diretoria, provocando uma espera maior do que a obtida no iLSFS local, o que resulta num desempenho inferior nas configurações distribuídas.

<table>
<tr><th colspan="3" rowspan="2"></th><th rowspan="2">Cache</th><th colspan="2">Débito (Ops/s)</th></tr>
<tr><th>Média ($\overline{x}$)</th><th>Desvio padrão (σ)</th></tr>
<tr><td colspan="3">Passthrough</td><td>-</td><td>29276.40</td><td>168.74</td></tr>
<tr><td rowspan="8">iLSFS</td><td colspan="2" rowspan="2">Local</td><td>off</td><td>78.31</td><td>12.76</td></tr>
<tr><td>on</td><td>3057.13</td><td>0.32</td></tr>
<tr><td rowspan="6">Distribuído</td><td rowspan="2">1 nodo - 1 grupo</td><td>off</td><td>106.93</td><td>9.34</td></tr>
<tr><td>on</td><td>1229.99</td><td>0.82</td></tr>
<tr><td rowspan="2">2 nodos - 1 grupo</td><td>off</td><td>112.49</td><td>8.87</td></tr>
<tr><td>on</td><td>1920.61</td><td>0.52</td></tr>
<tr><td rowspan="2">2 nodos - 2 grupos</td><td>off</td><td>108.54</td><td>9.19</td></tr>
<tr><td>on</td><td>1844.14</td><td>0.54</td></tr>
</table>

Tabela 7: Resultados Filebench - Consulta dos metadados de ficheiros em diversas configurações

Dentro do modelo distribuído, existe uma configuração que se destaca das restantes, nomeadamente, a configuração "2 nodos e 1 grupo", com um débito de 112.49 operações por segundo, quando não se encontra a cache em utilização, e 1920.61 Ops/s, quando a cache se apresenta ligada. O destaque desta configuração distribuída, nesta carga de trabalho, é similar ao observado nas leituras de ficheiros, onde esta configuração apresenta um desempenho bastante superior. Isto leva-nos a inferir que as operações de

consulta, seja de dados ou metadados, tiram mais proveito de uma configuração de 2 nodos a responder a pedidos do cliente, do que uma divisão das chaves por 2 nodos, no cenário em que as experiências decorreram.

Depois da análise às operações sobre metadados no sistema de ficheiros, conseguimos concluir que o uso da cache é muito vantajoso, permitindo-nos aumentar o desempenho do iLSFS.

5.1.2.2 Cenário Macro

Dada uma intuição de como o sistema iLSFS se comporta, e como diferentes configurações estruturais impactam o desempenho que este consegue atingir, é fundamental a introdução do mesmo num cenário macro, onde conseguimos obter uma melhor perceção do seu desempenho máximo. Como tal foi definido um conjunto de micro e macro testes, nos quais avaliamos, faseadamente, o impacto de diferentes parâmetros no sistema desenvolvido, tais como: a paralelização de blocos, o recurso ou não à cache de metadados, o tipo de balanceador empregue e variações nas definições de anti-entropia. Esta análise tem como objetivo principal a compreensão de como o sistema reage a diferentes parâmetros, de modo a selecionar os que proporcionam um melhor desempenho.

Para estes testes recorre-se à implementação do iLSFS com 50 nodos de armazenamento DataFlasks, e um fator de replicação mínimo de 3 nodos e máximo de 5, o que nos permite a construção de 16 grupos. Isto significa que cada grupo tem em média 3 elementos, capazes de responder a pedidos do cliente sobre o conjunto de chaves que armazenam. A disseminação de mensagens de SPS ocorre segundo um intervalo de 15 segundos nos nodos DataFlasks, e de 5 segundos no Cliente iLSFS, mais concretamente no balanceador de carga, para conhecimento e manutenção da vista de nodos. Cada nodo mantém uma vista com 8 elementos, e a cada troca de mensagens é realizada uma substituição de 7 dos elementos da vista. Em todas as cargas de trabalho é utilizado o balanceador de carga *Smart*, uma vez que sabemos que este é a solução ótima para o desempenho do sistema. No entanto, uma análise comparativa aos dois balanceadores de carga disponíveis é realizada com recurso aos testes macro, posteriormente.

Micro Testes

O iLSFS detém alguns mecanismos, como a paralelização de blocos, que permitem ao sistema uma otimização na gestão das operações de escrita e leitura de ficheiros. Este mecanismo tem o objetivo de diminuir o tempo de espera pelos pedidos efetuados, e consequentemente aumentar o desempenho que o sistema de ficheiros consegue alcançar nas operações sobre dados. Recorda-se que, este mecanismo permite a realização de um conjunto de pedidos, paralelamente, para a camada de armazenamento, de modo a evitar a lógica de envio e espera ativa de resposta por apenas um pedido singular. A par do envio paralelo dos pedidos realizados, estes são também distribuídos, aleatoriamente, pelos nodos do grupo de replicação, evitando que todos sejam delegados a apenas um nodo. Para a definição do número pedidos efetuados de cada vez, é utilizada uma notação que tem como base o tamanho do bloco estipulado no

iLSFS. Quando mencionamos uma paralelização de 16 KiB, durante uma escrita de um ficheiro, referimo-nos ao envio, de uma só vez, de 4 pedidos de inserção de blocos de 4 KiB de dados para a camada de armazenamento. Se a paralelização for de 4 KiB, não existe qualquer paralelismo efetuado, uma vez que este valor é igual ao tamanho base do bloco no iLSFS.

Assim, perante as cargas de trabalho de escritas sequenciais e de leituras sequenciais e aleatórias, são recolhidos os débitos de desempenho atingidos pelo sistema de ficheiros desenvolvido, avaliando diferentes configurações do mecanismo de paralelização, que nos permitirão aferir quais os valores de paralelização com desempenho mais elevado no sistema. Tal como na análise do cenário micro sobre estas cargas de trabalho, também aqui foram recolhidos os valores de desempenho alcançados pelo sistema LSFS.

Nestas experiências sobre dados, são também relacionados os valores de paralelismo com o tamanho do bloco (Tamanho I/O) pedido em cada operação do Filebench. De notar que apenas é relevante efetuar uma paralelização até ao tamanho I/O da carga de trabalho, já que paralelizações superiores não trazem qualquer benefício.

Escrita de ficheiros Na Tabela 8 apresentam-se os resultados alcançados pelo iLSFS e pelo LSFS, em cargas de trabalho de escritas sequenciais, perante diferentes valores de paralelismo.

Quando apenas são efetuadas escritas de blocos de tamanho 4 KiB, correspondente ao tamanho base de um bloco no iLSFS, o débito obtido pelo sistema é de 9.33 MiB/s. Este valor é próximo ao atingido pelo iLSFS quando configurado com "2 nodos e 1 grupo", com parâmetros iguais, o que nos permite admitir que o crescimento horizontal estabelecido, neste cenário de 50 nodos, não influenciou negativamente o seu desempenho. Contudo, nesta circunstância, não se consegue aproveitar o mecanismo de paralelização, dado que o bloco de 4 KiB, associado à operação, só pode ser escrito de uma só vez.

De forma similar, quando definido um tamanho I/O de 128 KiB, os valores de desempenho obtidos alcançam os 9.73 MiB/s. O incremento existente, em relação ao mesmo cenário com um tamanho I/O de 4 KiB, é explicado pelas menores mudanças de contexto requeridas quando é manipulado um bloco de maiores dimensões. Neste caso, o pedido de escrita de 128 KiB é realizado em apenas uma operação ao iLSFS, enquanto que para o mesmo cenário, com tamanho I/O de 4 KiB, seria necessário a realização de 32 diferentes pedidos e o mesmo número de mudanças de contexto, que naturalmente, detêm um pequeno impacto no débito obtido.

A partir do momento em que o paralelismo de escritas no iLSFS começa a crescer, e a ter influência no processamento dos blocos, é possível visualizar um aumento gradual do desempenho no sistema de ficheiros. Este aumento atinge o seu máximo quando é utilizada uma paralelização de 64 KiB ou 96 KiB, correspondente a um débito de aproximadamente 56 MiB/s. O recurso a estas paralelizações significa que em cada operação de escrita de um bloco de 128 KiB, que é dividido em 32 pedidos com tamanho de bloco de 4 KiB, são enviados paralelamente 16 ou 24 pedidos para os nodos de armazenamento, até que seja finalizada a operação. O aumento do paralelismo, para além dos 96 KiB, não traz qualquer benefício ao desempenho do iLSFS, indicando-nos que, para as escritas de ficheiros, aos 128 KiB é atingido o nível

	Tamanho I/O	Paralelização	Débito (MiB/s)	
			Média ($\overline{x}$)	**Desvio padrão** (σ)
Passthrough	128 KiB	-	133.07	20.55
LSFS	4 KiB	4 KiB	9.03	0.05
	128 KiB	4 KiB	9.9	0.36
		8 KiB	31.7	2.86
		16 KiB	44.83	1.01
		32 KiB	43.8	9.22
		64 KiB	59.0	2.61
		96 KiB	16.1	20.79
		128 KiB	0.36	0.38
iLSFS	4 KiB	4 KiB	9.33	0.15
	128 KiB	4 KiB	9.73	0.45
		8 KiB	19.43	2.05
		16 KiB	33.55	2.89
		32 KiB	47.7	1.93
		64 KiB	55.5	0.26
		96 KiB	56.23	1.06
		128 KiB	41.77	1.79

Tabela 8: Resultados Filebench - Escrita sequencial no cenário macro

de saturação da rede, que provoca perda de mensagens e atraso no normal funcionamento e execução das operações.

Ao incluirmos o mecanismo de paralelização na avaliação do iLSFS, é possível visualizar um aumento significativo do desempenho nas escritas sequenciais, quantificado em 42% do débito do sistema de referência *Passthrough*. Já ao compararmos o sistema iLSFS com o sistema que o originou, é observada uma diferença residual no desempenho alcançado. O LSFS, atinge o seu débito máximo de escritas com uma paralelização de 64 KiB, alcançando os 59.0 MiB/s, comparativamente aos 55.5 MiB/s atingidos pelo iLSFS com a mesma paralelização (máximo de 56.23 MiB/s com 96 KiB de paralelismo). Naturalmente, e como já foi evidenciado na análise às configurações de cenários micro, as novas funcionalidades do iLSFS afetam negativamente o desempenho do sistema, porém quando passamos para uma versão distribuída de maiores dimensões, a diferença existente passa a ser mínima, conseguindo o iLSFS aproximar-se do débito obtido pelo LSFS.

Leitura de ficheiros Ao realizarmos a mesma análise para as leituras sobre dados, é possível perceber um padrão semelhante ao observado nas escritas de ficheiros. Nas Tabelas 9 e 10 são divulgados os débitos de dados em leituras sequenciais e aleatórias, respetivamente, na presença de variações no paralelismo do iLSFS.

O desempenho máximo atingido pelo sistema desenvolvido é alcançado quando realizado um paralelismo de 32 KiB, o qual nos permite obter um débito de leituras de aproximadamente 58 MiB/s.

	Tamanho I/O	**Paralelização**	**Débito** (MiB/s)	
			Média ($\overline{x}$)	**Desvio padrão** (σ)
Passthrough	128 KiB	-	515.1	0.46
LSFS	4 KiB	4 KiB	6.97	0.06
	128 KiB	4 KiB	7.26	0.06
		8 KiB	22.23	1.70
		16 KiB	45.6	0.76
		32 KiB	63.67	0.21
		64 KiB	20.57	2.44
		128 KiB	1.9	0.1
iLSFS	4 KiB	4 KiB	8.9	1.11
	128 KiB	4 KiB	9.06	0.38
		8 KiB	18.83	2.74
		16 KiB	37.5	0.85
		32 KiB	57.83	1.76
		64 KiB	26.9	4.45
		128 KiB	13.23	2.63

Tabela 9: Resultados Filebench - Leitura sequencial no cenário macro

Em ambos os tipos de leituras, é possível observar que o recurso a um tamanho I/O de 4 KiB, apresenta sempre um débito inferior, quando no mesmo cenário, de ausência de paralelismo, se recorre a um tamanho do bloco lido pelo Filebench, de 128 KiB. Esta diferença, evidenciada com maior relevo nas leituras aleatórias, reforça, tal como aconteceu nas escritas, a nossa teoria, de que menores mudanças de contexto, por via de realização de apenas uma operação a requerer um bloco de dados maior, são vantajosas no iLSFS. Naturalmente, nem sempre existe a necessidade de utilizar toda a quantidade de dados pedida num bloco de 128 KiB, no entanto, num cenário diferente do constituído para estas experiências, onde existe o recurso às estratégias de *readahead* e da *page cache*, o sistema operativo tentará sempre pedir ao sistema de ficheiros uma quantidade de dados superior àquela que é requerida pela operação da carga de trabalho. No caso do FUSE, o valor máximo de bloco encontra-se limitado a 128 KiB, querendo

isto dizer que independentemente do tamanho do bloco efetivamente pedido, ao FUSE poderá sempre chegar um pedido de leitura de um bloco de 128 KiB. É claro que este cenário só acontece quando não é ativada a diretiva *"Direct I/O"*, contudo, não sendo esse o caso, continua a ser importante perceber qual o comportamento do iLSFS quando é requerido um bloco de tamanho máximo, e qual o paralelismo que melhor estimula o seu desempenho.

	Tamanho I/O	**Paralelização**	**Débito** (MiB/s)	
			Média ($\overline{x}$)	**Desvio padrão** (σ)
Passthrough	128 KiB	-	495.67	1.62
LSFS	4 KiB	4 KiB	7.0	0.01
	128 KiB	4 KiB	7.23	0.06
		8 KiB	19.53	1.03
		16 KiB	44.43	2.56
		32 KiB	63.63	0.95
		64 KiB	22.47	2.16
		128 KiB	1.7	0.14
iLSFS	4 KiB	4 KiB	8.17	0.32
	128 KiB	4 KiB	9.13	0.49
		8 KiB	17.0	0.82
		16 KiB	35.7	0.3
		32 KiB	57.77	1.10
		64 KiB	30.33	2.78
		128 KiB	13.16	0.51

Tabela 10: Resultados Filebench - Leitura aleatória no cenário macro

Para além disso, podemos também relacionar o valor máximo de desempenho atingido pelo iLSFS em leituras, com o sistema de referência, sendo que o sistema desenvolvido consegue alcançar entre 11% e 12% do débito de leituras obtido pelo *Passthrough*. Já comparando com o sistema LSFS, as conclusões tiradas, são em tudo similares à escrita de ficheiros. Também aqui, a diferença de desempenho entre os dois sistemas é bastante pequena. O LSFS atinge o seu débito de leituras máximo com uma paralelização de 32 KiB, alcançando, aproximadamente, os 64 MiB/s em ambos os tipos de leituras. Isto significa que o iLSFS, com todas as funcionalidades adicionadas apresenta 91% do débito de leituras apresentadas pelo sistema LSFS, representando uma diferença baixa, de 9%, no desempenho entre os dois sistemas.

Operações sobre metadados Análogo ao estudo realizado nos cenários micro, realizou-se a mesma bateria de micro testes, de operações sobre metadados, para a configuração do iLSFS com 50 nodos.

Nas Tabelas 11, 12 e 13 são apresentados os débitos de desempenho do sistema de ficheiros, para as operações de criação, eliminação e consulta de metadados de ficheiros, perante a desconexão e ativação das caches desenvolvidas.

Analisando os resultados obtidos, mais uma vez é comprovado que o recurso às caches de metadados no iLSFS promove um crescimento no débito de operações realizadas por segundo, nas diversas cargas de trabalho. Neste cenário existe um incremento de 524% na criação de ficheiros, 437% na eliminação de ficheiros e 1219% na consulta de metadados de ficheiros, quando ocorre uma ativação das caches.

	Cache	**Débito** (Ops/s)	
		Média ($\overline{x}$)	**Desvio padrão** (σ)
iLSFS	off	101.13	3.04
	on	530.21	2.61

Tabela 11: Resultados Filebench - Criação de ficheiros no cenário macro

	Cache	**Débito** (Ops/s)	
		Média ($\overline{x}$)	**Desvio padrão** (σ)
iLSFS	off	41.79	1.09
	on	182.80	3.43

Tabela 12: Resultados Filebench - Eliminação de ficheiros no cenário macro

	Cache	**Débito** (Ops/s)	
		Média ($\overline{x}$)	**Desvio padrão** (σ)
iLSFS	off	85.68	5.20
	on	1044.64	102.73

Tabela 13: Resultados Filebench - Consulta dos metadados de ficheiros no cenário macro

Na Tabela 14, através da recolha dos recursos utilizados pelo iLSFS durante a consulta de metadados de ficheiros, em que as operações realizadas são unicamente sobre metadados, é bem evidenciado o impacto que a utilização das caches tem no sistema como um todo. Por um lado, o uso da cache aumenta ligeiramente os recursos consumidos no Cliente do sistema, no que à percentagem de CPU se refere, devido à utilização recorrente da cache, que promove um consumo mais intensivo de recursos no cliente, mas por outro lado, no mesmo cenário, é observável um decréscimo considerável dos recursos consumidos nos Nodos de armazenamento. Esta diminuição deve-se à menor quantidade de pedidos efetuados remotamente, sendo estes, na sua maioria, provenientes do processo de atualização da cache de metadados de diretorias, enquanto os pedidos realizados pelo cliente são servidos pelos dados em cache.

	Cache	Cliente		Nodos de armazenamento	
		CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
iLSFS	off	2.3	1477.49	33.35	867.02
	on	4.0	1479.68	21.85	864.94

Tabela 14: Recursos utilizados - Consulta de metadados de ficheiros no cenário macro

Macro testes

Mecanismos como o balanceamento de carga e a anti-entropia contribuem para o normal funcionamento do iLSFS, detendo um papel crucial, quer seja na disseminação das mensagens para os nodos DataFlasks, quer na manutenção da coerência dos dados. Desta forma, também estes necessitam de ser testados, de modo a encontrar os parâmetros que colaboram positivamente no alcance de um melhor desempenho do sistema de ficheiros. Assim, mediante uma carga de trabalho macro, que simula um servidor web, é analisado o comportamento destes mecanismos num cenário realista.

Na Tabela 15 são apresentados os débitos de desempenho obtidos pelo iLSFS quando experimentadas as duas opções de balanceamento da carga no sistema. Facilmente conseguimos perceber que o recurso a um balanceador de carga *Smart* é a estratégia que melhor desempenho consegue extrair, ao apresentar 648.22 operações por segundo, um aumento de 558%, em relação ao balanceador concorrente, que apenas chega às 116.15 operações por segundo. Esta diferença significativa é explicada pela natureza dos dois tipos de balanceadores: O *Random* segue uma abordagem totalmente aleatória, onde um pedido efetuado pode ter de passar por diversos nodos até que chegue a um que consiga processar a mensagem, o que, logicamente, aumenta o tempo de resposta a cada pedido e a utilização de recursos da rede. Já o *Smart*, segue uma estratégia de disseminação eficiente, adquirindo um conhecimento da composição e divisão de grupos, que lhe permite efetuar uma ligação direta com nodos capazes de processar as mensagens.

	Balanceador de Carga	**Débito** (Ops/s)	
		Média ($\overline{x}$)	**Desvio padrão** (σ)
Passthrough	-	31184.63	1366.2
iLSFS	Random	116.15	30.27
	Smart	648.22	22.61

Tabela 15: Resultados Filebench - Servidor web no cenário macro - Avaliação ao balanceador de carga

As estratégias distintas de cada um dos balanceadores não só têm impacto no desempenho, como também apresentam repercussões nos recursos consumidos pelo sistema. Na Tabela 16, é possível observar a diferença na utilização de CPU em cada uma das abordagens empregue. Quando se recorre ao balanceador *Random* são consumidos, 3 vezes mais recursos, nos nodos de armazenamento, do que quando se recorre à estratégia *Smart*, para o balanceamento da carga. As características de disseminação

aleatórias do balanceador *Random* promovem um constante processamento e envio de mensagens pelos diversos nodos da rede, resultando numa inundação destes com pedidos, que naturalmente, se traduz num consumo elevado de recursos.

	Balanceador de carga	**Cliente**		**Nodos de armazenamento**	
		CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
iLSFS	Random	3.87	1392.92	75.95	845.24
	Smart	4.28	1398.21	24.51	869.2

Tabela 16: Recursos utilizados - Servidor web no cenário macro - Avaliação ao balanceador de carga

No iLSFS, a Anti-entropia, pode ser configurada através de parâmetros como a quantidade de chaves disseminadas, ou o intervalo de espera entre cada ativação do mecanismo. Estas configurações detêm um papel importante na eficácia e rapidez da recuperação de dados entre os nodos do sistema, contudo, podem também promover uma sobreutilização dos recursos dos nodos e da rede. Assim, de modo a analisar o impacto que estes parâmetros têm no desempenho do sistema de ficheiros foi ativado o mecanismo de Anti-entropia e definiu-se: Uma quantidade de chaves disseminadas em cada iteração do mecanismo como 50% e 100% da totalidade de chaves presentes na base de dados, representativa de uma divulgação intensiva dos dados detidos por cada nodo; Um intervalo entre mensagens de Anti-entropia de 30, 60 e 300 segundos.

Na Tabela 17 são expostos os resultados recolhidos destas experiências. Como teste de controlo recorremos ao desempenho obtido na Tabela 15, perante um balanceador de carga *Smart*. Nesse cenário ainda não se encontra ligado o mecanismo de recuperação de dados, o que possibilita a obtenção de um débito de 648.22 operações por segundo.

	Quantidade de chaves	**Intervalo de Anti-Entropia**	**Débito** (Ops/s)	
			Média ($\overline{x}$)	**Desvio padrão** (σ)
iLSFS	50%	30 s	635.92	33.17
		60 s	606.95	35.87
		300 s	643.01	12.58
	100%	30 s	661.42	56.76
		60 s	608.59	57.66
		300 s	640.08	35.56

Tabela 17: Resultados Filebench - Servidor web no cenário macro - Avaliação à anti-entropia

Analisando o desempenho alcançado quando é disseminado 50% da base de dados, o recurso a um intervalo de Anti-entropia de 300 segundos é mais vantajoso, permitindo o alcance de um débito ligeiramente superior ao observado num intervalo menor. Porém, seria de esperar que um intervalo de 60

segundos obtivesse um desempenho superior ao de 30 segundos, pela menor frequência de disseminação de mensagens, mas tal não é evidenciado. Na verdade, é verificado um comportamento similar durante a transmissão de 100% das chaves armazenadas, uma vez que o intervalo de maior frequência, 30 segundos, apresenta um desempenho superior aos restantes e ainda superior ao teste de controlo. Nestas experiências, constata-se ainda uma elevada variância nos resultados obtidos, que dificulta a retirada de conclusões objetivas. Contudo, podemos afirmar que o impacto da anti-entropia, no desempenho geral do sistema, quando sujeito à carga de trabalho de servidor web, é mínimo, dado que obtemos resultados muito próximos do atingido pelo teste de controlo.

5.1.3 Discussão

Os resultados da avaliação de desempenho do iLSFS retratam um sistema capaz de suportar o mais variado tipo de cargas de trabalho micro e macro, tanto sobre dados como metadados. Depois de realizado um estudo aos diversos parâmetros e combinações dos mesmos, o iLSFS, conseguiu alcançar um débito máximo, em operações sobre dados, de 56.23 MiB/s para as escritas e de 57.83 MiB/s para as leituras de ficheiros. Em operações sobre metadados, foi atingido, no cenário macro, um desempenho de 530.21 Ops/s durante a criação de ficheiros, 182.80 Ops/s na eliminação de ficheiros e 1044.64 Ops/s na consulta de metadados de ficheiros. Já para a carga de trabalho realista, representativa de um servidor web, o sistema desenvolvido apresentou um débito máximo de 661.42 Ops/s. Para atingir estes valores de desempenho o sistema de ficheiros foi configurado com uma paralelização de 96 KiB para as escritas de dados, e 32 KiB para as leituras. Verificou-se ainda que o balanceador de carga *Smart* é uma mais valia no roteamento eficiente dos pedidos pela rede, melhorando substancialmente o desempenho do sistema. Para além disso, confirmou-se que as otimizações introduzidas à cache de metadados, bem como a sua expansão, permitiram a obtenção de um aumento considerável no desempenho do sistema, em todas as cargas de trabalho onde foi avaliado.

Constatou-se ainda que o sistema iLSFS, apesar de possuir funcionalidades que introduzem uma maior complexidade ao sistema e afetam desfavoravelmente o seu desempenho (como evidenciado nos cenários micro), consegue manter um débito muito próximo ao alcançado pelo LSFS, um sistema mais simplista, quando ambos são introduzidos num cenário com maior distribuição de dados, e maior quantidade de nodos. Ou seja, podemos inferir, que quanto maior a dimensão do sistema, em termos de escala, mais próximo será o desempenho do iLSFS, em relação ao LSFS, tornando o impacto da complexidade introduzida desprezável.

Com esta análise aos diversos componentes do iLSFS, temos agora um conhecimento alargado de quais os parâmetros que proporcionam a extração do melhor desempenho possível para este sistema de ficheiros totalmente descentralizado, bem como uma perceção do impacto das melhorias e funcionalidades adicionadas no sistema.

5.2 Escalabilidade e Resiliência

Um dos aspetos de maior relevo no iLSFS remete-se às suas propriedades de escalabilidade e resiliência, bem destacadas ao longo deste documento. Desde o recurso a um protocolo epidémico, até ao versionamento de dados selecionado, toda a arquitetura do sistema e respetivos mecanismos encontram-se desenhados de forma que nenhum seja um entrave ao seu crescimento em redes de muita larga escala. Como tal, é essencial perceber como o sistema de ficheiros se comporta quando sujeito a um caso de estudo real, numa rede de grandes dimensões, onde a ocorrência de falhas nos seus componentes é uma constante. Deste modo procedeu-se à criação de uma infraestrutura iLSFS com 500 nodos de armazenamento, sendo introduzida instabilidade artificial na rede, por intermédio da remoção e adição de nodos (*churn*), durante o treino de uma rede neuronal de classificação de imagens.

5.2.1 Metodologia

Antes de expostos os resultados obtidos nesta avaliação, é útil a compreensão da metodologia seguida para a obtenção dos mesmos. Primeiro, é detalhado o ambiente experimental de larga escala construído, bem como as ferramentas utilizadas para o efeito. De seguida, é descrita a carga de trabalho selecionada e delineada a estratégia de injeção de faltas, para avaliar a resiliência do sistema, e por fim é apresentada a configuração do iLSFS.

5.2.1.1 Ambiente Experimental

O ambiente experimental para a avaliação da escalabilidade e resiliência do iLSFS segue uma metodologia díspar da adotada na avaliação de desempenho. Para nos aproximarmos o mais possível de um cenário realista de maiores dimensões, e pela escassez de recursos que a abordagem anterior apresentava em acomodar tal cenário, optamos por migrar o desenvolvimento para uma nova infraestrutura *cloud* na Google Cloud Platform (GCP) [16]. Esta abordagem permitiu-nos facilmente simular uma rede de larga escala, conseguindo a sua expansão até aos 500 nodos de armazenamento pretendidos.

Para este cenário recorremos a máquinas de computação representativas de cada componente do iLSFS, consoante os recursos necessários. Foi tida em particular consideração a máquina selecionada para representar o Nodo DataFlasks, uma vez que pretendemos que esta se assemelhe a dispositivos IoT, no que às características de baixo poder de processamento e reduzido espaço de armazenamento diz respeito. Deste modo, e seguindo a nomenclatura definida pela *Google*, a máquina selecionada para os nodos DataFlasks é do tipo *g1-small* e dispõe de 1 vCPU e 1,7 GB de RAM. Como armazenamento é-lhe atribuído um disco de inicialização SSD, com 25 GB de memória.

Ao contrário do que é definido para os nodos de armazenamento, a máquina selecionada para o cliente do sistema de ficheiros necessita de apresentar características muito superiores. Esta máquina, para além de se encontrar responsável por acomodar uma instância do Cliente iLSFS, tem como função a execução do conjunto de cargas de trabalho, na qual é necessária a instalação de uma unidade de

processamento gráfico (GPU). Assim o cliente do sistema é introduzido numa máquina designada por *n1-standard-4*, que apresenta 4 vCPUs e 15 GB de RAM, com um disco de inicialização SSD definido nos 50 GB de armazenamento. Para além do disco de inicialização, é ainda adicionado um disco externo SSD, com 200 GB, capaz de armazenar o *dataset* de classificação de imagens escolhido. No que diz respeito à capacidade de processamento gráfico foi adicionada uma *Nvidia Tesla T4*.

Para a gestão e orquestração de todas as máquinas presentes na infraestrutura, manteve-se a abordagem de um cluster Kubernetes, tal como é definido no ambiente experimental para a avaliação de desempenho. Cada *Pod* executa uma instância de um container Docker, relativo aos componentes do iLSFS, sendo a sua alocação feita sempre numa máquina independente, onde os recursos disponibilizados pela mesma se encontram todos ao seu dispor. Posto isto, o cluster criado para este ambiente experimental necessita de 502 *workers* - 500 nodos de armazenamento, 1 cliente e 1 bootstrapper.

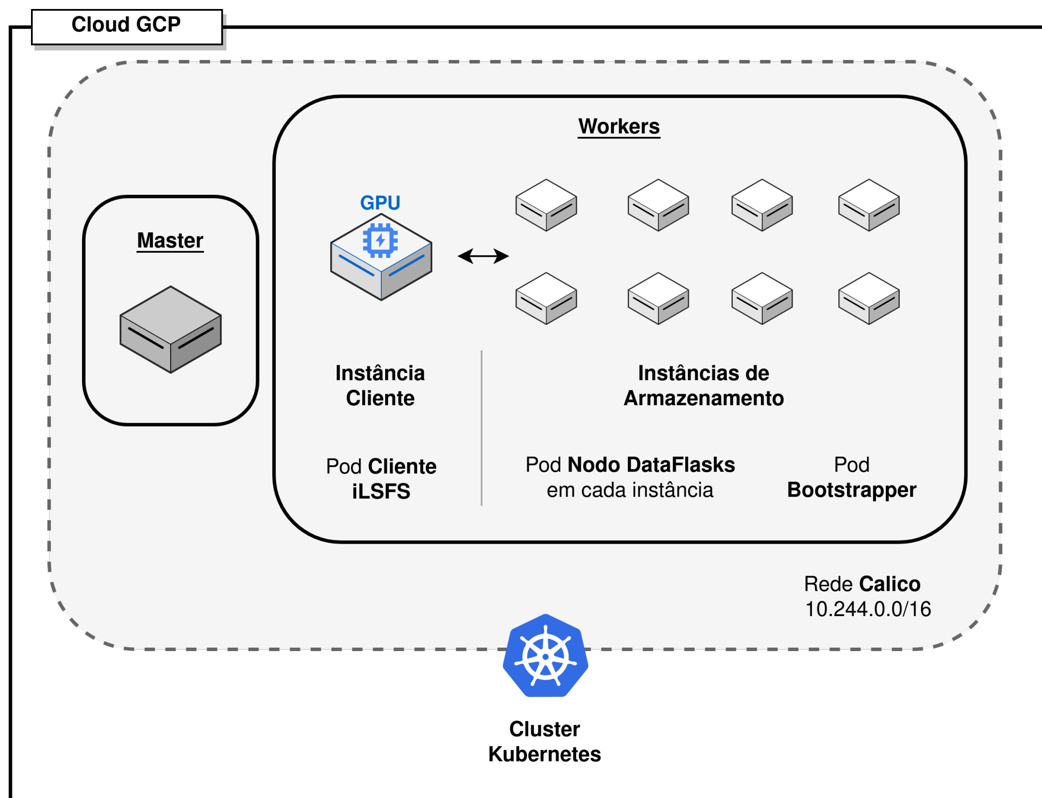


Figura 18: Ambiente experimental - Cloud GCP

Apresentada a arquitetura dos nodos que realizam o trabalho no cluster, resta agora perceber qual a configuração do nodo *Master* deste cluster Kubernetes. Para um cluster de tal dimensão é necessário que esta máquina seja capaz de gerir e manter operacional todos os nodos que fazem parte do mesmo. Como tal, a máquina de referência para um cluster desta dimensão é do tipo *n1-standard-16* e dispõe de 16 vCPUs e 60 GB de RAM. Como disco de inicialização são disponibilizados 15 GB de armazenamento SDD. Na Figura 18 é possível ver a arquitetura do cluster Kubernetes descrita.

Para a construção desta infraestrutura optamos pela utilização da ferramenta Terraform [52], pela

sua total integração com o provedor de *cloud* da *Google*. O recurso a esta ferramenta permite a criação da infraestrutura em código, automatizando todo esse processo, mediante a especificação dos diversos componentes na sua linguagem *HashiCorp Configuration Language*. Através desta ferramenta facilmente conseguimos proceder à criação e destruição da infraestrutura sempre que necessário. Para além disso, e tal como é realizado no ambiente experimental onde avaliamos o desempenho do sistema, também aqui procedemos à utilização de *scripts* Ansible e *Bash* para automação do mais variado conjunto de tarefas, como a preparação das máquinas para a receção do cluster, ou a alocação de cada *Pod* à sua respetiva máquina.

Em todas as máquinas presentes no cluster é efetuada uma monitorização de diversos aspetos do sistema durante a realização dos testes definidos, através da ferramenta *dstat*. Complementarmente é ainda monitorizada a unidade de processamento gráfico presente na máquina do cliente, com recurso à ferramenta *nvidia-smi*, o que nos permite o estudo de informação como a percentagem de GPU em uso, ou o consumo de memória, durante o período de teste. A recolha e processamento destes dados serve de auxílio na análise de resultados desenvolvida.

5.2.1.2 Carga de trabalho

A carga de trabalho definida para a avaliação da escalabilidade e resiliência do iLSFS seguiu a mesma lógica de contexto realista, já definido para o ambiente experimental do sistema.

Hoje em dia, com o crescimento da inteligência artificial, o recurso a tecnologia de treino para aprendizagem máquina é vastamente utilizada, com diversas aplicabilidades, nas mais variadas áreas. No contexto IoT, no qual o sistema se insere, os dados produzidos ou agregados por estes dispositivos são muitas vezes utilizados como fonte de informação para o desenvolvimento de redes neuronais, na aprendizagem ou reconhecimento de padrões. Posto isto, consideramos importante, perceber o comportamento do sistema de ficheiros desenvolvido num caso de estudo atual e real, como o da aprendizagem máquina. Assim, recorremos à *framework* Tensorflow, para a criação e execução do treino de uma rede neuronal de classificação de imagens.

O treino de uma rede neuronal corresponde, em exclusivo, a um conjunto de operações de leituras sobre um determinado conjunto de dados, que através do seu processamento, consegue, progressivamente, construir conhecimento sobre esse mesmo conjunto de dados. Para o efeito, necessitamos de escolher qual a rede neuronal e o respetivo *dataset*, que melhor se enquadram no contexto de classificação de imagens. Como modelo de rede neuronal, foi selecionada a rede convolucional LeNet, vastamente utilizada e conhecida no processamento e análise de imagens. Para o conjunto de dados utilizado no treino da rede neuronal selecionou-se o *dataset* ILSVRC [40], versão ILSVRC2012, disponibilizado pela base de dados de imagens ImageNet [10]. O *dataset* empregue correspondeu a um subconjunto do mesmo, prefazendo 105 GiB em amostras de imagens. Como configurações para o treino foi empregue um *shuffle buffer* de 10000 e um *batch size* de 64 KiB, a decorrer, no máximo, durante uma iteração (*epoch*) pela amostra de dados.

5.2.1.3 Injeção de falhas

Uma parte integral do sistema de ficheiros iLSFS é a sua redundância no armazenamento de dados e estrutura adaptativa, que o dota de uma grande resiliência a falhas na rede. Por esta razão, pretendemos aferir o comportamento do sistema, tanto a nível da sua persistência de dados, como na sua capacidade de recuperação, quando sujeito a instabilidade provocada pela falha de nodos do sistema. Definiu-se, então, um conjunto de testes de falhas, que juntamente com a carga de trabalho, nos permitem a realização de uma avaliação completa à resiliência do iLSFS. De modo a compreender o impacto que a inserção de instabilidade tem no sistema, recorre-se à análise do desempenho obtido pelo sistema durante o treino da rede neuronal, comparando-o com o momento em que as falhas são introduzidas.

Os testes de falhas decorrem por intermédio da injeção de falhas artificiais, simuladas através da introdução de *churn* no sistema. Mais concretamente, é realizado um conjunto de 4 iterações de falhas durante um período de treino de 2 horas e 30 minutos, onde a cada 30 minutos é injetado no sistema um conjunto de falhas totalmente aleatórias. Nos primeiros 30 minutos do teste não é introduzida nenhuma instabilidade no sistema, permitindo-nos usar os valores de desempenho obtidos, como referência de débito estável para o restante período em análise. A metodologia da falha consiste na seleção e remoção de vários nodos DataFlasks do sistema, sendo estes, posteriormente, substituídos por nodos indênticos com uma base de dados vazia. O período entre a remoção dos nodos e a sua substituição encontra-se definido nos 5 minutos, tempo suficiente para analisarmos o comportamento do sistema quando a falha acontece e percebermos como este reage à adição de novos nodos. É ainda de realçar, que quando os nodos entram no sistema, estes necessitam de realizar uma recuperação total da base de dados e para isso, recorrem à fase de recuperação do mecanismo de anti-entropia.

A instabilidade introduzida no sistema pretende ser o mais próximo possível do que poderia acontecer num cenário real. Posto isto, é necessário que a seleção dos nodos a remover, em cada iteração de injeção de falhas, seja realizada aleatoriamente pelos grupos de nodos, sem quaisquer restrições adjacentes. Para o efeito desenvolveu-se um *script* Python, que em cada iteração seleciona, de forma aleatória, um conjunto de nodos distintos, de entre os 500 nodos de armazenamento no sistema, que representa o volume de injeção de falhas realizado. A quantidade de nodos selecionada varia, e corresponde a 1%, 3% e 5% da totalidade de nodos presentes no iLSFS.

5.2.1.4 Configuração iLSFS

Para o cenário estipulado, o sistema desenvolvido necessitou de ser configurado de maneira a apresentar uma elevada resiliência, que possibilitasse a sustentação do seu normal funcionamento, mesma na presença de elevados níveis de agitação. Com isto em mente, começou-se por estruturar o sistema, no que aos grupos de nodos diz respeito, através da especificação do fator de replicação pretendido. Estabeleceu-se então um tamanho de grupo, segundo um intervalo mínimo de 25, e máximo de 35, o que permitiu uma convergência para 16 grupos de replicação, possuindo cada um destes, em média, 31 elementos. Quanto aos parâmetros dos diversos mecanismos do sistema, foram utilizados, maioritariamente, como

base, os valores que nos permitiram alcançar o melhor débito possível, durante a avaliação de desempenho. Assim, o mecanismo de paralelização foi fixado com um paralelismo de escritas de 64 KiB e leituras de 32 KiB. De notar, que, para as escritas, o valor de paralelismo difere do valor que nos permitiu obter o desempenho máximo, já que consideramos benéfico, para a saturação da rede, a disseminação de uma menor quantidade de mensagens, em cada iteração do mecanismo, quando o desempenho alcançado entre um paralelismo de 64 KiB e 96 KiB é tão próximo. Para o roteamento dos pedidos pela rede, recorreu-se ao balanceador de carga *Smart*, onde foi estipulada uma troca de mensagens de SPS a cada 5 segundos, de modo a viabilizar uma rápida e ágil adaptação a qualquer mudança ou instabilidade nos nodos da rede. Definiu-se ainda, que cada pedido efetuado pelo cliente teria de ser enviado pelo balanceador para 3 nodos distintos. Durante os testes preliminares, foi constatado que o envio dos pedidos, para apenas um nodo resultava numa penalização no desempenho obtido, pela ocorrência de *timeouts* nas mensagens enviadas para nodos que se encontravam temporariamente indisponíveis. Assim, decidiu-se introduzir um certo nível de redundância no processo de disseminação, de forma a aumentar a probabilidade de resposta em tempo útil. Quanto aos nodos DataFlasks, o mecanismo de SPS, foi configurado com uma vista máxima de 20 elementos e uma troca de mensagens a cada 15 segundos. Em cada uma das mensagens disseminadas estabeleceu-se o envio da informação sobre 9 nodos, prontamente incorporada pelo nodo recetor. No mecanismo de anti-entropia, definiu-se um ciclo operacional com início a cada 60 segundos, sendo enviadas, em cada iteração, 5% das chaves armazenadas pelo nodo.

5.2.2 Resultados

Nesta secção são apresentados os débitos obtidos, resultantes do treino de uma rede neuronal de classificação de imagens, que nos permitirão comprovar, juntamente com a injeção de falhas, a escalabilidade e resiliência do iLSFS.

Posto isto, a análise efetuada encontra-se dividida em dois conjuntos de testes, com objetivos bem definidos, sobre a mesma carga de trabalho, e que nos permitem analisar diferentes características do sistema.

1º Conjunto de testes - Cenário estável

O primeiro conjunto de experiências tem como objetivo validar a escalabilidade do sistema de ficheiros desenvolvido, num cenário estável e sem perturbações externas. Para isso, recorre-se à análise do desempenho obtido pelo iLSFS durante um período de 30 minutos de treino da rede neuronal. Estas experiências repetiram-se três vezes, de modo a assegurar a consistência e solidez dos resultados obtidos, o que permitiu a recolha do débito médio, e o respetivo desvio padrão sobre a quantidade de imagens processadas por segundo pela rede neuronal.

Complementarmente, este primeiro conjunto de testes são ainda reproduzidos num sistema de ficheiros local, regularmente utilizado no contexto de aprendizagem máquina, e que servirá como termo de comparação para o desempenho obtido pelo iLSFS durante o treino da rede neuronal. As experiências

realizadas sobre este sistema, recorrem à execução da carga de trabalho numa máquina igual à usado pelo Cliente iLSFS, sendo o treino da rede neuronal sustentado pelo *dataset* de imagens, armazenado no disco externo instalado.

Na Tabela 18 são apresentados os valores de débito alcançados por cada sistema, durante o treino da rede neuronal.

	Débito (Imagens/s)	
	Média ($\overline{x}$)	**Desvio padrão** (σ)
Local	974.43	5.4
iLSFS	99.32	1.26

Tabela 18: Treino LeNet - Comparação entre o sistema de ficheiros Local e iLSFS

Os resultados obtidos demonstram que o iLSFS, numa configuração resiliente de 500 nodos, permite que a rede neuronal LeNet processe em média 99.32 imagens por cada segundo de treino realizado. Este débito, é representativo de aproximadamente 10% do desempenho que o sistema de ficheiros local consegue proporcionar à rede neuronal, atingindo um débito médio, nas três experiências, de 974.43 imagens por segundo. Evidentemente, a configuração de armazenamento local, pode atingir um débito de leituras substancialmente superior, devido às próprias características de acesso local ao *dataset*, ao contrário do que acontece no iLSFS que oferece uma solução de armazenamento remoto. Contudo, o sistema de ficheiros local, e todo o ecossistema onde reside, é altamente vulnerável por representar um ponto único de falha, algo não existente no iLSFS.

Ao analisarmos os valores presentes nas Tabelas 19 e 20, representativas dos recursos consumidos por cada um dos sistemas, encontramos também alguns dados de relevo, que salientam o comportamento dos sistemas durante a execução da carga de trabalho. O sistema de ficheiros local apresenta uma utilização de recursos intensiva, aproximando-se dos 90% de utilização de CPU e 60% da unidade de processamento gráfica, crucial no contexto de aprendizagem máquina. Este facto está diretamente relacionado com o trabalho realizado na máquina pelo cenário local, que exerce um processamento de imagens elevado, como exposto no desempenho obtido durante o treino da rede neuronal. Em contraste, o iLSFS apresenta um consumo de recursos baixo, de aproximadamente 35% de utilização de CPU e 8% da GPU. Com isto, conseguimos deduzir que a máquina cliente tem capacidade para realizar um treino mais rápido e intenso, contudo, a leitura de dados aos nodos DataFlasks limita o desempenho do treino, como era espectável.

	GPU (%)	CPU (%)	RAM (MiB)
Local	58.65	88.15	5122.5

Tabela 19: Treino LeNet - Recursos consumidos pelo sistema de ficheiros local

	Cliente			**Nodos de armazenamento**	
	GPU (%)	CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
iLSFS	7.59	34.41	5396.96	19.64	523.2

Tabela 20: Treino LeNet - Recursos consumidos pelo iLSFS

2º Conjunto de testes - Injeção de falhas

O segundo conjunto de testes tem como objetivo averiguar a resiliência que o sistema de ficheiros desenvolvido apresenta, por intermédio da injeção de falhas artificiais que simulam um cenário de grande instabilidade na rede. Assim, tal como é detalhado na Secção 5.2.1.3, o iLSFS é submetido a um conjunto de falhas, distribuídas de forma totalmente aleatória pelos nodos de armazenamento. Começa-se por introduzir na configuração resiliente do iLSFS um nível de instabilidade correspondente à falha de 1% dos 500 nodos existentes, seguido da falha de 3% e de 5%. Como métrica para a análise ao comportamento do sistema, é acompanhado ao longo do período de teste o desempenho do treino da rede neuronal, tal como é apresentado nas Figuras 19, 20 e 21. Em cada um dos gráficos são apontados, através de uma linha vertical a tracejado, os momentos de relevo na experiência. O primeiro, representado a cor vermelha, corresponde ao momento de introdução da falha, ou seja, à saída ou remoção de um conjunto de nodos do sistema, consoante o nível de instabilidade definido. Já o segundo, com cor verde, simboliza a entrada de nodos, mais concretamente a altura em que estes já se encontram com uma participação ativa na disseminação de mensagens de SPS, e se encontram em processo de recuperação de chaves, pelo mecanismo de anti-entropia. De modo a perceber o impacto acumulado que a instabilidade provoca no sistema de ficheiros, a injeção de falhas é introduzida em 4 instantes distintos, espaçados no tempo.

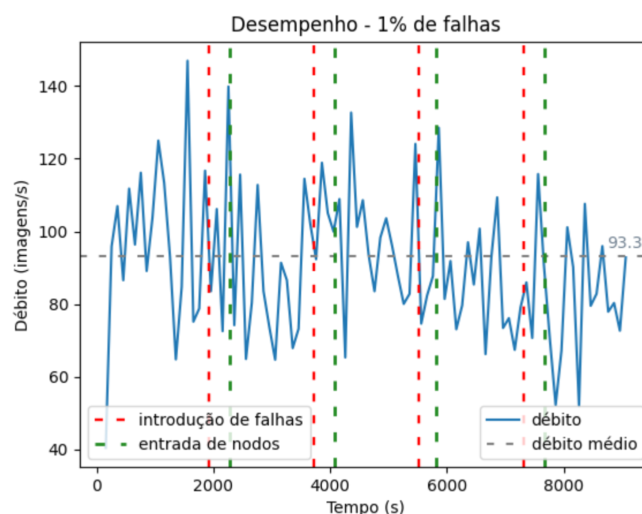


Figura 19: Treino LeNet - Injeção de 1% de falhas no iLSFS

Num primeiro momento é possível perceber, nos três gráficos, que os resultados recolhidos, ao longo

do período de teste, apresentam uma grande variabilidade no desempenho, mesmo durante o intervalo inicial de 30 minutos, onde não ocorre nenhuma saída de nodos, e o sistema se encontra num estado estável. Tomando como exemplo o gráfico da Figura 19, é possível observar oscilações no débito obtido, que atingem um máximo de 149 imagens/s, até um mínimo de 56 imagens/s. Esta variabilidade no desempenho, pode ser explicada, pela natureza do iLSFS, que devido às necessárias e constantes comunicações entre nodos, seja através do mecanismo de SPS, ou de Anti-Entropia, provocam, momentaneamente, uma maior quantidade de mensagens na rede e nos nodos, originando a sua saturação. Desta forma, se as operações realizadas ao sistema ocorrerem durante um período de menor troca de mensagens na rede, o sistema de ficheiros consegue proporcionar um débito de leituras mais elevado. Por outro lado, se estas operações se sucederem durante um período de maior intensidade, o débito será, consequentemente, menor, provocando o atraso ou até a perda dos pedidos realizados. Ainda assim, observando cada um dos 4 momentos de injeção de falhas conseguimos identificar alguns padrões que nos permitem perceber o comportamento do iLSFS nos instantes seguintes à remoção e entrada de nodos no sistema. Estes padrões, são evidenciados, com maior destaque, durante a injeção de 3% e 5% de falhas, enquanto que na falha de 1% de nodos, os resultados obtidos variam de iteração para iteração, não apresentando um padrão consistente.

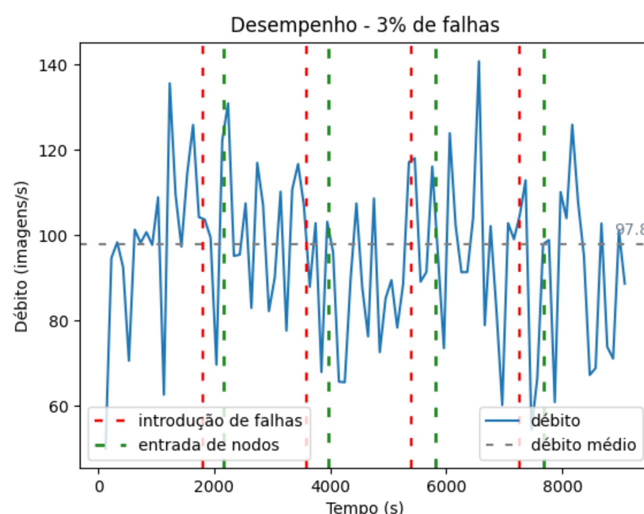


Figura 20: Treino LeNet - Injeção de 3% de falhas no iLSFS

Na injeção de 3% de falhas (15 nodos), após a remoção dos nodos de armazenamento, é visível, um decréscimo no desempenho do treino, causado pela forçosa adaptação dos nodos, e dos respetivos mecanismos, à saída destes elementos dos seus grupos. Contudo, rapidamente, e ainda antes da entrada de novos elementos, é observado que o débito iguala ou até supera os valores verificados antes da injeção de falhas. Isto acontece, porque, durante um curto período de tempo, após a saída de nodos, os elementos que permanecem no sistema, ainda têm a perceção de que os nodos removidos se encontram em funcionamento, e por isso, continuam a enviar-lhes mensagens de SPS e Anti-Entropia, até que estes

saiam, efetivamente, das suas vistas. Assim, gradualmente, e à medida que os nodos atualizam as suas vistas, temos menos mensagens na rede, e o débito cresce. Quando ocorre a entrada de nodos no sistema, é possível visualizar o mesmo padrão inicial de decréscimo do débito de treino. Na realidade, com a entrada de nodos no sistema, para além dos necessários ajustes nos mecanismos, aos novos elementos, passa a existir uma maior circulação de mensagens na rede, correspondente a comunicações com nodos recentemente adicionados, que, como é observado, tem repercussões no treino.

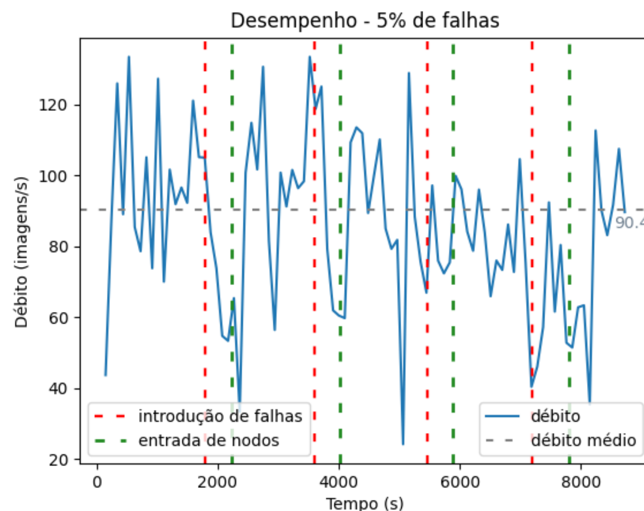


Figura 21: Treino LeNet - Injeção de 5% de falhas no iLSFS

Na injeção de 5% de falhas (25 nodos), o sistema demonstra o mesmo comportamento de picos e vales descrito anteriormente. A primeira iteração de falhas realça bem este padrão. Nesta, é observado um decréscimo do débito do treino no momento em que são removidos do sistema 25 nodos de armazenamento de forma aleatória. A partir desse instante, os nodos DataFlasks, têm de ajustar os seus mecanismos, de forma a retirar das suas vistas nodos que já não se encontram na rede. Da mesma forma, após a entrada de nodos no sistema, uma diminuição no desempenho é constatada, até que o débito retorna a valores previamente observados.

Para além do natural processo de adaptação dos mecanismos ao *churn* no sistema nos nodos de armazenamento, também o mesmo processo acontece no Cliente iLSFS, uma vez que se recorre ao balanceador de carga *Smart* para o roteamento dos pedidos. Ao guardar um conjunto de nodos, de cada grupo, com os quais é possível comunicar, passa a ser necessário que o armazenamento da topologia se ajuste ao *churn* no sistema, de maneira a mitigar o envio de pedidos para nodos que já não se encontrem disponíveis. Todavia, durante o treino da rede neuronal constatamos que a ocorrência de *timeouts*, originada pela não receção de uma resposta em tempo útil, não ocorria apenas durante a injeção de falhas, mas ao longo de todo o treino, levando-nos a questionar a razão de tal acontecer. Nesta carga de trabalho, a grande quantidade de dados continuamente transmitidas pela rede refere-se às respostas das operações de leitura. Num cenário de uma configuração resiliente do sistema, estas comunicações com

o cliente têm um impacto particular, quando incluímos na equação a camada de redundância adicionada no envio dos pedidos para os nodos de armazenamento. Esta requer que por cada pedido efetuado, 3 mensagens sejam enviadas do cliente para os nodos, despoletando 3 respostas, para o cliente. Este tráfego constante de mensagens, provoca a saturação da largura de banda do cliente, limitando o seu processamento de pedidos, o que impede, por vezes, a receção de respostas e a sua posterior entrega ao utilizador. Logo, a saturação da rede do cliente, provoca diversos *timeouts* para os pedidos realizados, não estando estes totalmente relacionados com a injeção de falhas.

Distribuição de falhas pelos grupos De modo a termos uma melhor perceção do impacto que a introdução aleatória de falhas provocou nos grupos de replicação de dados, monitorizamos os nodos selecionados em cada iteração de instabilidade, e identificamos o grupo a que pertenciam. Como resultado obtemos a Tabela 21, onde podemos verificar como se distribui a seleção aleatória de nodos pelos 16 grupos. É possível perceber, que apesar de termos realizado uma escolha totalmente aleatória dos nodos a serem removidos, em cada iteração de falhas, a seleção de nodos por grupos aconteceu de forma relativamente dispersa, não havendo uma aglomeração elevada que originasse a remoção da maioria dos elementos de um só grupo de replicação. Por exemplo, quando injetado 5% de instabilidade no iLSFS, verificou-se a remoção aproximada de 2 nodos de armazenamento de cada grupo.

	Iterações							
	1ª		**2ª**		**3ª**		**4ª**	
	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
1% - 5 nodos	0.31	0.48	0.31	0.60	0.31	0.60	0.31	0.48
3% - 15 nodos	0.94	0.93	0.94	0.93	0.94	0.68	0.94	1.06
5% - 25 nodos	1.56	1.26	1.56	1.03	1.56	1.21	1.56	0.81

Tabela 21: Distribuição de *churn* pelos 16 grupos de replicação, em cada iteração

Depois de adquirida uma perceção do comportamento do iLSFS, perante a injeção de falhas, resta comparar o desempenho médio do sistema em diferentes níveis de instabilidade, com o cenário livre de perturbações, apresentado no primeiro conjunto de testes. Na Tabela 22 são exibidos os valores médios de desempenho relativo a cada teste de injeção de falhas. Quando é introduzido 1% de falhas, o débito médio do treino fixa-se nas 93.3 imagens/s processadas, com 3% de falhas nas 97.77 imagens/s e com 5% de falhas, nas 90.37 imagens/s processadas. Nos três casos, o valor médio do desempenho, representa um ligeiro decréscimo, em comparação, com o débito médio de 99.32 imagens/s processadas no contexto estável. Contudo, esse valor não representa nenhum padrão que nos permita admitir que quanto maior for a instabilidade no sistema, menor é o desempenho que o sistema proporciona ao treino da rede neuronal. Assim, conseguimos objetivamente concluir, que no panorama geral do iLSFS, a introdução de instabilidade, em forma de *churn* de diferentes níveis, tem um impacto residual no desempenho geral do sistema de ficheiros, sendo esta diferença explicada, em parte, pela variabilidade do treino. O mesmo

pode ser verificado ao analisarmos os recursos consumidos no cenário de injeção de falhas, presentes na Tabela 23, em comparação com os medidos no contexto estável (Tabela 20). Em todos os parâmetros medidos obtemos valores de utilização idênticos, que são representativos do débito de imagens obtido no treino da rede neuronal.

	Débito (Imagens/s)
1%	93.3
3%	97.77
5%	90.37

Tabela 22: Treino LeNet - Débito médio do treino no iLSFS durante a injeção de falhas

	Cliente			**Nodos DataFlasks**	
	GPU (%)	CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
1%	8.87	38.56	5419.81	19.76	524.22
3%	9.44	32.22	5726.55	19.6	523.2
5%	7.59	34.35	5421.02	19.82	523.16

Tabela 23: Treino LeNet - Recursos consumidos pelo iLSFS durante a injeção de falhas

5.2.3 Discussão

A avaliação de escalabilidade e resiliência ao sistema de ficheiros permitiu-nos comprovar, num contexto real, a capacidade que o iLSFS tem em suportar uma rede de larga escala, e sustentar o seu normal funcionamento, mesmo quando submetido a diferentes níveis de instabilidade na camada de armazenamento.

Com uma configuração bastante resiliente, e introduzido numa rede com 500 nodos, o iLSFS conseguiu exibir uma grande escalabilidade, mostrando-se capaz de servir o treino de uma rede neuronal e gerir centenas de nodos de armazenamento ao mesmo tempo. Neste cenário, o sistema desenvolvido atingiu, durante o treino da rede neuronal de classificação de imagens, 10% do desempenho alcançado por um sistema de ficheiro local, sobre a mesma carga de trabalho. Esta diferença estabelece-se como o compromisso de desempenho esperado quando utilizamos o iLSFS numa configuração que priorize uma elevada resiliência e disponibilidade dos dados que armazena. Todavia, é quando o sistema é testado perante falhas de nodos, que transparecem os seus pontos fortes. Com 20% de falhas acumuladas durante a sessão de treino, o iLSFS demonstrou uma enorme resiliência, mantendo-se capaz de continuar a responder a pedidos do cliente, mesmo quando, em vários momentos, 5% do total de nodos se encontravam indisponíveis.

6 Conclusão

Os sistemas de armazenamento distribuídos, hoje em dia, procuram oferecer garantias de elevada resiliência e disponibilidade, tendo em vista dispor de uma capacidade de tolerância a faltas que lhes permita permanecer operacionais durante períodos de elevada instabilidade, bem como garantir uma grande escalabilidade, de forma a acomodar redes de grandes dimensões, sem restrições. No estado de arte, apenas o sistema LSFS consegue manifestar todas as características mencionadas, e ao mesmo tempo exportar uma interface POSIX, capaz de proporcionar a interação do sistema de armazenamento como um sistema de ficheiros. Baseado em protocolos epidémicos *peer-to-peer*, este sistema recorre a algoritmos não estruturados e descentralizados para conseguir atingir a alta escalabilidade que o caracteriza. Contudo, limitações como a falta de operações essenciais, gestão de metadados restritiva, ou a grande carga que é exercida sobre a rede, e que provoca a sua saturação, têm um impacto negativo na usabilidade, inviabilizando a sua aplicação em ambientes para os quais foi proposto.

Esta dissertação propôs assim o iLSFS, uma extensão ao sistema de ficheiros LSFS, onde o objetivo principal se fixou na aplicação e desenvolvimento de técnicas que permitem melhorar a usabilidade do mesmo.

No sistema desenvolvido todas as funcionalidades introduzidas tiveram em consideração as características não estruturadas e de escalabilidade, únicas deste sistema de ficheiros, de modo que nenhuma pudesse ser um entrave ao crescimento do mesmo. Como tal, começou-se por definir um novo modelo de acesso a dados que proporcionasse aos utilizadores do sistema de ficheiros a criação, a modificação, a leitura e a eliminação de ficheiros ou diretorias no iLSFS, sendo que, até aqui, apenas era permitido a um utilizador a criação e a leitura de dados. Deste modo, tornou-se fundamental a implementação de um mecanismo que permitisse a obtenção de uma relação de causalidade entre atualizações de dados no sistema, e que viabilizasse o controlo de concorrência entre as diversas atualizações efetuadas pelos utilizadores. Os *Version Vectors* com *id-per-client*, mostraram-se a melhor opção para este propósito, tendo em conta as características de disseminação das operações pelos nodos de armazenamento, e a importância de não existirem obstáculos à escalabilidade do sistema. Assim, através deste mecanismo, foi possível disponibilizar, a qualquer utilizador, a modificação de dados e identificar conflitos provocados

por atualizações concorrentes. Adicionalmente, e com o mesmo propósito, também a introdução da operação de eliminação foi objeto de estudo para o alcance de uma melhor usabilidade no iLSFS. Tendo em conta o modelo de coerência eventual, próprio deste sistema, para a introdução desta operação foi necessário optar por uma solução que oferecesse a persistência e disseminação da operação de eliminação por todos os nodos DataFlasks do sistema, de forma que a mesma pudesse perdurar no tempo, e nunca ser anulada. Esta exigência levou à implementação de um mecanismo de *tombstones*, que, mediante o armazenamento dos dados eliminados, possibilitou a introdução da operação de eliminação de ficheiros e diretorias no iLSFS.

Para além da necessidade da apresentação de um modelo de acesso aos dados mais completo, tornou-se também inevitável a expansão e alteração da gestão de metadados de diretorias, de maneira a que esta pudesse ir ao encontro dos padrões de usabilidade que pretendíamos para este sistema. Para isso, os metadados de diretorias sofreram um total redesenho na forma como operacionalizavam no sistema, retirando as limitações de tamanho existentes, e ao mesmo tempo, reduzindo a informação disseminada, ao estritamente necessário, de modo que esta tivesse um menor impacto na saturação da rede. Procedeu-se ainda ao desenvolvimento de uma cache de metadados de diretorias, com o objetivo de diminuir a quantidade de pedidos efetuados à camada de armazenamento, providenciando um melhor desempenho nas operações sobre metadados e um tempo de resposta menor para o cliente.

Efetuou-se uma avaliação experimental detalhada ao iLSFS, na qual se analisou o desempenho, a escalabilidade e a resiliência do sistema de ficheiros. Mostrou-se, por intermédio de um conjunto de micro e macro testes intensivos, que as novas funcionalidades no iLSFS contemplaram uma melhor usabilidade, em troca de uma ligeira redução do desempenho do sistema, comparativamente ao alcançado pelo LSFS num cenário e configurações idênticas. Mostrou-se ainda que a cache de metadados representa uma mais valia em operações sobre metadados, evidenciando um aumento substancial do desempenho do sistema, quando esta se encontra operacional. Para a avaliação da escalabilidade e resiliência, o sistema foi submetido a um cenário atual e realista, enquadrado na aprendizagem máquina, como é o treino de uma rede neuronal para a classificação de imagens. Configurado com 500 nodos DataFlasks e características altamente resilientes, o iLSFS demonstrou ter uma elevada escalabilidade, capaz de proporcionar a leitura de dados para o treino ininterruptamente. O mesmo foi constatado quando se simulou instabilidade no sistema, através da injeção de falhas totalmente aleatórias, em que o iLSFS, perante a falha de 20% de nodos de armazenamento, no período do treino, manteve um desempenho contínuo, demonstrando uma enorme resiliência, capaz de se adaptar a elevados níveis de *churn* no sistema.

6.1 Trabalho Futuro

Nesta dissertação são realizados contributos importantes que permitem ao iLSFS tornar-se um sistema de ficheiros mais competitivo, aproximando-se daquilo que sistemas de armazenamento mais conhecidos na comunidade oferecem, no que às funcionalidades se refere. Ainda assim, o desenvolvimento deste

sistema é um trabalho em progresso, existindo espaço para melhorias, quer a nível do Cliente iLSFS, quer a nível dos Nodos DataFlasks.

A distribuição de chaves pelos nodos de armazenamento, é uma característica que, de entre muitas vantagens, permite ao sistema uma divisão da carga e um processamento de pedidos mais rápido, quando a disseminação é realizada eficientemente pelo balanceador de carga. Para isso, o balanceador necessita de possuir um conhecimento da totalidade da rede de nodos, tal como acontece no balanceador *Smart*. Todavia, ter este conhecimento representa uma limitação à escalabilidade do sistema, uma vez que, a partir de uma certa dimensão da rede, torna-se incomportável o armazenamento, no Cliente iLSFS, da informação sobre todos os nodos e configuração de grupos existentes, e no caso de pretendermos este armazenamento, uma restrição à escalabilidade do iLSFS teria de ser posta em prática. Deste modo, sugere-se o desenvolvimento de uma solução de disseminação de pedidos mais equilibrada, que realize uma seleção inteligente dos nodos que armazena, sem comprometer a latência dos pedidos, e que, ao mesmo tempo, não se estabeleça como uma restrição à escalabilidade do iLSFS. Complementarmente, uma abordagem mais arrojada pode introduzir métricas como tempo de resposta ou sucesso de resposta para ordenar ou classificar os nodos dos grupos armazenados pelo balanceador, de forma a privilegiar o desempenho do sistema.

Para além disso, um estudo pormenorizado ao mecanismo de gestão de pedidos no Cliente iLSFS ajudará a mitigar ainda mais a saturação da rede do cliente, permitindo a introdução de melhorias que tenham um impacto benéfico no desempenho do sistema, seja através da diminuição de pedidos que não cheguem ao destino, como de respostas que, devido à saturação da largura de banda do cliente não conseguem ser processados com sucesso. De forma similar, um estudo ao mecanismo de anti-entropia que permita encontrar a melhor configuração para diversos cenários é pertinente, ajudando a perceber a influência que diferentes parâmetros têm na convergência de estados e na saturação da rede.

Bibliografia

[1] M. Abadi et al. "Tensorflow: A system for large-scale machine learning". Em: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283 (ver p. 4).

[2] *Accelerated, containerized application development*. url: `https://www.docker.com/` (acedido em 21/09/2022) (ver p. 53).

[3] *Amazon Web Services IoT*. url: `https://aws.amazon.com/pt/iot/` (acedido em 28/12/2021) (ver p. 1).

[4] R. H. Ansible. *Ansible is simple it automation*. url: `https://www.ansible.com/` (acedido em 12/10/2022) (ver p. 54).

[5] *Boost C++ libraries*. url: `https://www.boost.org/` (acedido em 05/03/2023) (ver p. 51).

[6] J.-M. Busca, F. Picconi e P. Sens. "Pastis: a Highly-Scalable Multi-User Peer-to-Peer File System". Em: ago. de 2005, pp. 1173–1182 (ver p. 9).

[7] A. Cunha et al. *Version Vectors are not Vector Clocks*. 2011. url: `https://haslab.wordpress.com/2011/07/08/version-vectors-are-not-vector-clocks/` (acedido em 17/07/2022) (ver pp. 4, 32).

[8] F. Dabek et al. "Wide-Area Cooperative Storage with CFS". Em: *SIGOPS Oper. Syst. Rev.* 35.5 (out. de 2001), pp. 202–215 (ver p. 10).

[9] G. DeCandia et al. "Dynamo: Amazon's highly available key-value store". Em: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220 (ver pp. 7, 32).

[10] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". Em: *CVPR09*. 2009 (ver p. 77).

[11] *Differences from POSIX*. url: `https://docs.ceph.com/en/latest/cephfs/posix/` (acedido em 20/01/2022) (ver p. 11).

[12] P. Druschel e A. Rowstron. "PAST: a large-scale, persistent peer-to-peer storage utility". Em: *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. 2001, pp. 75–80 (ver p. 7).

[13] S. Ghemawat, H. Gobioff e S.-T. Leung. “The Google file system”. Em: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, pp. 29–43 (ver p. 10).

[14] S. Gilbert e N. Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. Em: *SIGACT News* 33.2 (jun. de 2002), pp. 51–59 (ver p. 7).

[15] Google. *LevelDB*. url: `https://github.com/google/leveldb` (acedido em 12/03/2023) (ver p. 13).

[16] *Google Cloud: Cloud Computing Services*. url: `https://cloud.google.com/` (acedido em 01/12/2022) (ver p. 75).

[17] J. H. Howard et al. *An overview of the andrew file system*. Vol. 17. Carnegie Mellon University, Information Technology Center, 1988 (ver p. 9).

[18] “IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7”. Em: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018), pp. 1–3951 (ver p. 2).

[19] *Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025*. url: `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/` (acedido em 20/12/2021) (ver p. 1).

[20] *Internet of Things on IBM Cloud*. url: `https://www.ibm.com/cloud/internet-of-things` (acedido em 28/12/2021) (ver p. 1).

[21] R. Klophaus. “Riak Core: Building Distributed Applications without Shared State”. Em: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP ’10. Baltimore, Maryland: Association for Computing Machinery, 2010 (ver pp. 7, 32).

[22] A. Lakshman e P. Malik. “Cassandra: A Decentralized Structured Storage System”. Em: *SIGOPS Oper. Syst. Rev.* 44.2 (abr. de 2010), pp. 35–40 (ver pp. 7, 36, 37).

[23] J. Li et al. “Comparing the performance of distributed hash tables under churn”. Em: *Peer-to-Peer Systems III: Third International Workshop, IPTPS 2004, La Jolla, CA, USA, February 26-27, 2004, Revised Selected Papers 3*. Springer. 2005, pp. 87–99 (ver p. 8).

[24] *Linux FUSE (Filesystem in Userspace) interface*. url: `https://github.com/libfuse/libfuse` (acedido em 12/03/2023) (ver p. 13).

[25] F. Maia et al. “DATAFLASKS: Epidemic Store for Massive Scale Systems”. Em: *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. 2014, pp. 79–88 (ver pp. 8, 20).

[26] P. Maymounkov e D. Eres. “Kademlia: A Peer-to-peer Information System Based on the XOR Metric”. Em: *Peer-to-Peer Systems*. Abr. de 2002 (ver p. 7).

[27] *MooseFS*. url: `https://moosefs.com/` (acedido em 07/01/2022) (ver p. 8).

[28] M. Muntjir, M. Rahul e H. A. Alhumyani. “An Analysis of Internet of Things (IoT): Novel Architectures, Modern Applications, Security Aspects and Future Scope with Latest Case Studies”. Em: *International Journal of Engineering Research and Technology* 6 (2017). issn: 2278-0181 (ver p. 1).

[29] A. Muthitacharoen et al. “Ivy: A Read/Write Peer-to-Peer File System”. Em: *SIGOPS Oper. Syst. Rev.* 36.SI (dez. de 2003), pp. 31–44 (ver p. 9).

[30] *MySQL*. url: `https://www.mysql.com/` (acedido em 05/01/2022) (ver p. 7).

[31] *Pods*. url: `https://kubernetes.io/docs/concepts/workloads/pods/` (acedido em 08/10/2022) (ver p. 54).

[32] *PostgreSQL*. url: `https://www.postgresql.org/` (acedido em 05/01/2022) (ver p. 7).

[33] N. Preguiça et al. “Dotted version vectors: Logical clocks for optimistic replication”. Em: *arXiv preprint arXiv:1011.5808* (2010) (ver p. 33).

[34] *Production-grade container orchestration*. url: `https://kubernetes.io/` (acedido em 08/10/2022) (ver p. 53).

[35] Projectcalico. *Calico: Cloud native networking and network security*. url: `https://github.com/projectcalico/calico` (acedido em 10/10/2022) (ver p. 54).

[36] *Protocol buffers*. url: `https://protobuf.dev/` (acedido em 05/03/2023) (ver p. 51).

[37] S. Ratnasamy et al. “A Scalable Content-Addressable Network”. Em: SIGCOMM ’01. San Diego, California, USA: Association for Computing Machinery, 2001, pp. 161–172 (ver p. 7).

[38] S. Rhea et al. “Handling Churn in a DHT”. Em: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’04. Boston, MA: USENIX Association, 2004 (ver p. 8).

[39] A. Rowstron. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. Em: vol. 2218. Out. de 2002 (ver p. 7).

[40] O. Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. Em: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. doi: `10.1007/s11263-015-0816-y` (ver p. 77).

[41] R. Sandberg. “The Sun network file system: Design, implementation and experience”. Em: *in Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*. 1986 (ver p. 9).

[42] M. Satyanarayanan et al. “Coda: a highly available file system for a distributed workstation environment”. Em: *IEEE Transactions on Computers* 39.4 (1990), pp. 447–459 (ver p. 9).

[43] *ScyllaDB - The Real-Time Big Data Database*. url: `https://www.scylladb.com/` (acedido em 12/03/2023) (ver p. 7).

[44] M. Shapiro et al. “A comprehensive study of convergent and commutative replicated data types”. Tese de doutoramento. Inria–Centre Paris-Rocquencourt; INRIA, 2011 (ver p. 38).

[45] M. Shapiro et al. *Conflict-free Replicated Data Types*. Research Report RR-7687. Jul. de 2011, p. 18. url: `https://hal.inria.fr/inria-00609399` (ver p. 37).

[46] K. Shvachko et al. "The Hadoop Distributed File System". Em: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10 (ver p. 8).

[47] D. da Silva Fernandes. *"LSFS: Sistema de ficheiros tolerante a faltas para armazenamento em larga escala"*. 2020 (ver pp. 2, 11, 22).

[48] K. Sonbol et al. "EdgeKV: Distributed Key-Value Store for the Network Edge". Em: *2020 IEEE Symposium on Computers and Communications (ISCC)*. 2020, pp. 1–6 (ver p. 7).

[49] I. Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications". Em: *IEEE/ACM Trans. Netw.* 11.1 (fev. de 2003), pp. 17–32 (ver p. 7).

[50] V. Tarasov, E. Zadok e S. Shepler. "Filebench: A flexible framework for file system benchmarking". Em: *USENIX; login* 41.1 (2016), pp. 6–12 (ver p. 54).

[51] P. Taylor. *Total Data Volume Worldwide 2010-2025*. url: `https://www.statista.com/statistics/871513/worldwide-data-created/` (acedido em 20/12/2021) (ver p. 1).

[52] *Terraform*. url: `https://www.terraform.io/` (acedido em 15/12/2022) (ver p. 76).

[53] S. A. Weil et al. "Ceph: A scalable, high-performance distributed file system". Em: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 307–320 (ver pp. 8, 11).

[54] *What is IoT?* url: `https://www.oracle.com/pt/internet-of-things/what-is-iot/` (acedido em 05/12/2021) (ver p. 1).

[55] B. Zhang et al. "The Cloud is Not Enough: Saving Iot from the Cloud". Em: HotCloud'15. Santa Clara, CA: USENIX Association, 2015, p. 21 (ver p. 1).

[56] B. Zhao et al. "Tapestry: A Resilient Global-Scale Overlay for Service Deployment". Em: *IEEE Journal on Selected Areas in Communications* 22 (jul. de 2003) (ver p. 7).

A

Apêndice

A.1 Utilização de recursos

A.1.1 Avaliação de desempenho

A.1.1.1 Cenário Micro

			Cliente		Nodos de armazenamento	
			CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
LSFS	**Local**		15.08	657.3	-	-
	Distribuído	1 nodo - 1 grupo	2.11	1348.81	12.83	569.81
		2 nodos - 1 grupo	3.35	1360.89	15.99	551.95
		2 nodos - 2 grupos	2.05	1354.06	13.78	580.77
iLSFS	**Local**		15.79	320.12	-	-
	Distribuído	1 nodo - 1 grupo	2.14	1391.23	13.76	564.87
		2 nodos - 1 grupo	3.13	1385.96	23.35	589.18
		2 nodos - 2 grupos	1.77	1393.77	14.4	655.87

Tabela 24: Recursos utilizados - Escritas sequenciais em diversas configurações

			Cliente		Nodos de armazenamento	
			CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
LSFS	**Local**		16.27	636.27	-	-
	Distribuído	1 nodo - 1 grupo	2.33	1348.76	10.84	560.47
		2 nodos - 1 grupo	4.61	1361.13	9.8	549.44
		2 nodos - 2 grupos	2.38	1357.0	17.73	582.04
iLSFS	**Local**		16.36	313.15	-	-
	Distribuído	1 nodo - 1 grupo	2.53	1403.69	11.92	548.29
		2 nodos - 1 grupo	4.72	1393.12	10.43	572.37
		2 nodos - 2 grupos	2.34	1395.71	20.25	583.46

Tabela 25: Recursos utilizados - Leituras sequenciais em diversas configurações

			Cliente		Nodos de armazenamento	
			CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
LSFS	**Local**		15.89	632.43	-	-
	Distribuído	1 nodo - 1 grupo	2.63	1339.17	11.35	559.18
		2 nodos - 1 grupo	4.55	1355.64	9.81	547.43
		2 nodos - 2 grupos	2.05	1360.61	12.65	611.09
iLSFS	**Local**		16.15	311.35	-	-
	Distribuído	1 nodo - 1 grupo	2.65	1397.61	10.87	548.76
		2 nodos - 1 grupo	4.72	1399.69	10.5	565.25
		2 nodos - 2 grupos	2.55	1404.05	16.22	567.71

Tabela 26: Recursos utilizados - Leituras aleatórias em diversas configurações

			Cache	Cliente		Nodos de armaz.	
				CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
iLSFS	**Local**		off	16.88	307.46	-	-
			on	21.2	315.69	-	-
	Distribuído	1 nodo - 1 grupo	off	2.32	1395.57	16.18	558.74
			on	2.65	1408.12	17.87	558.57
		2 nodos - 1 grupo	off	2.22	1417.61	28.69	564.93
			on	3.24	1408.44	19.22	645.56
		2 nodos - 2 grupos	off	2.62	1399.85	9.01	647.54
			on	2.52	1409.58	15.81	646.97

Tabela 27: Recursos utilizados - Criação de ficheiros em diversas configurações

			Cache	Cliente		Nodos de armaz.	
				CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
iLSFS	**Local**		off	17.82	310.43	-	-
			on	24.73	318.13	-	-
	Distribuído	1 nodo - 1 grupo	off	2.24	1397.32	14.63	544.27
			on	3.32	1404.26	16.77	545.93
		2 nodos - 1 grupo	off	2.27	1404.37	17.85	649.52
			on	4.14	1411.36	18.54	573.3
		2 nodos - 2 grupos	off	2.41	1404.46	10.71	570.74
			on	3.1	1401.5	14.2	572.81

Tabela 28: Recursos utilizados - Eliminação de ficheiros em diversas configurações

			Cache	Cliente		Nodos de armaz.	
				CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
iLSFS	**Local**		off	17.5	312.33	-	-
			on	19.67	326.46	-	-
	Distribuído	1 nodo - 1 grupo	off	2.54	1402.98	14.44	544.05
			on	3.85	1416.65	15.14	545.8
		2 nodos - 1 grupo	off	2.73	1408.65	29.23	568.55
			on	5.98	1411.83	11.13	581.89
		2 nodos - 2 grupos	off	2.41	1402.01	10.98	648.36
			on	4.55	1414.04	14.83	566.32

Tabela 29: Recursos utilizados - Consulta de metadados de ficheiros em diversas configurações

A.1.1.2 Cenário Macro

	Tamanho I/O	Paralelização	Cliente		Nodos de armazenamento	
			CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
LSFS	4 KiB	4 KiB	3.19	1405.39	19.12	862.46
	128 KiB	4 KiB	3.04	1413.12	19.59	862.78
		8 KiB	6.68	1408.25	23.12	886.52
		16 KiB	8.44	1410.93	35.11	890.9
		32 KiB	6.44	1400.76	22.51	901.54
		64 KiB	10.22	1416.75	41.64	902.08
		96 KiB	1.49	1409.22	3.36	864.37
		128 KiB	1.1	1396.72	1.52	853.93
iLSFS	4 KiB	4 KiB	4.59	1596.5	21.35	786.9
	128 KiB	4 KiB	2.2	1562.44	14.15	799.49
		8 KiB	7.08	1591.68	32.87	835.28
		16 KiB	11.68	1606.87	33.82	918.35
		32 KiB	17.29	1597.02	40.3	851.04
		64 KiB	21.86	1613.14	43.43	827.63
		96 KiB	20.23	1593.9	36.98	983.03
		128 KiB	12.03	1589.02	21.0	824.42

Tabela 30: Recursos utilizados - Escritas sequenciais no cenário macro

	Tamanho I/O	Paralelização	Cliente		Nodos de armazenamento	
			CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
LSFS	4 KiB	4 KiB	3.14	1400.38	17.37	856.29
	128 KiB	4 KiB	3.14	1392.88	17.4	857.78
		8 KiB	5.69	1407.0	32.06	858.98
		16 KiB	11.45	1415.67	35.36	858.11
		32 KiB	15.81	1401.08	40.57	858.54
		64 KiB	7.03	1418.34	12.35	864.05
		96 KiB	1.64	1403.46	4.18	864.83
		128 KiB	1.65	1409.56	3.71	865.22
iLSFS	4 KiB	4 KiB	5.11	1576.64	11.6	819.53
	128 KiB	4 KiB	4.69	1582.23	16.83	858.68
		8 KiB	7.4	1597.61	30.17	857.82
		16 KiB	13.89	1594.4	32.81	1015.07
		32 KiB	22.59	1584.05	35.28	827.55
		64 KiB	12.28	1596.82	12.04	947.69
		96 KiB	5.07	1607.96	8.47	931.55
		128 KiB	6.88	1606.15	10.26	888.18

Tabela 31: Recursos utilizados - Leituras sequenciais no cenário macro

	Tamanho I/O	Paralelização	Cliente		Nodos de armazenamento	
			CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
LSFS	4 KiB	4 KiB	3.23	1406.67	17.25	852.52
	128 KiB	4 KiB	3.4	1404.08	17.42	854.88
		8 KiB	5.55	1413.16	32.0	856.35
		16 KiB	11.85	1423.93	34.63	854.88
		32 KiB	15.73	1405.74	41.32	856.08
		64 KiB	6.61	1404.64	11.87	858.84
		96 KiB	1.83	1399.73	4.39	857.7
		128 KiB	1.51	1393.4	3.53	859.18
iLSFS	4 KiB	4 KiB	4.67	1581.33	16.7	884.54
	128 KiB	4 KiB	4.69	1575.79	19.08	809.1
		8 KiB	7.54	1578.1	30.08	816.02
		16 KiB	13.74	1582.53	38.89	826.31
		32 KiB	21.69	1586.87	38.43	957.1
		64 KiB	11.1	1592.17	13.1	922.53
		96 KiB	7.67	1569.61	9.47	830.29
		128 KiB	8.13	1579.42	8.38	845.14

Tabela 32: Recursos utilizados - Leituras aleatórias no cenário macro

	Cache	Cliente		Nodos de armazenamento	
		CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
iLSFS	off	2.49	1468.95	17.04	844.7
	on	2.82	1403.56	15.67	890.88

Tabela 33: Recursos utilizados - Criação de ficheiros no cenário macro

	Cache	Cliente		Nodos de armazenamento	
		CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
iLSFS	off	2.58	1482.16	16.31	868.94
	on	3.1	1486.36	23.09	800.17

Tabela 34: Recursos utilizados - Eliminação de ficheiros no cenário macro

	Quantidade de chaves	**Intervalo de Anti-Entropia**	**Cliente**		**Nodos de armaz.**	
			CPU (%)	RAM (MiB)	CPU (%)	RAM (MiB)
iLSFS	50 %	30 s	4.44	1388.51	24.43	869.68
		60 s	4.35	1403.43	24.65	870.43
		300 s	4.27	1400.61	24.77	872.75
	100 %	30 s	4.36	1402.78	24.35	874.31
		60 s	3.99	1401.68	24.32	873.7
		300 s	4.3	1413.89	24.16	877.48

Tabela 35: Recursos utilizados - Servidor web no cenário macro - Avaliação à anti-entropia