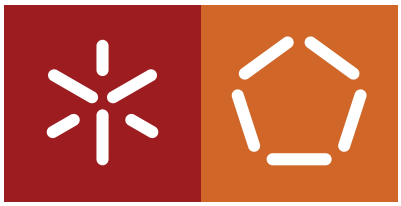
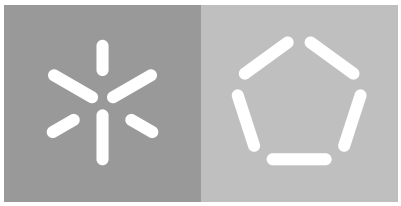


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel da Silva Fernandes

LSFS: Sistema de ficheiros tolerante a faltas para armazenamento em larga escala

Dezembro 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel da Silva Fernandes

LSFS: Sistema de ficheiros tolerante a faltas para armazenamento em larga escala

Dissertação de mestrado
Mestrado Integrado em Engenharia Informática

Trabalho realizado sob a orientação do Professor
João Tiago Medeiros Paulo
Rui Carlos Oliveira

Dezembro 2020

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

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.

AGRADECIMENTOS

Com este trabalho termina uma etapa importante da minha vida. Desta forma, gostaria de deixar uma mensagem especial de apreço e gratidão para todas as pessoas que de uma ou doutra forma contribuíram para a realização desta dissertação.

Começo por agradecer ao meu orientador, João Paulo, por toda a dedicação que colocou neste trabalho, pela disponibilidade em ajudar-me em qualquer situação e, sobretudo, por me aturar nos momentos de maior desespero sempre com uma posição otimista, encontrando soluções simples para todos os problemas. Agradeço ao meu co-orientador, Francisco Maia, por arranjar sempre tempo para ajudar mesmo quando ocupado pelo stress do seu trabalho, pela visão idealista na resolução de problemas e pelos conhecimentos transmitidos. A estes dois senhores, um muito obrigado do fundo do coração, foram impecáveis. Agradeço, ainda, ao Doutor Rui Oliveira, co-orientador, pelo apoio no decorrer deste projeto.

Um muito obrigado aos meus amigos, pelos momentos de convívio e diversão que me proporcionaram, aliviando-me a cabeça do trabalho e stress, aos quais também aproveito para pedir desculpa pelos convites recusados. Quero agradecer à minha família, principalmente aos meus pais, por todo o apoio, pelas mensagens de incentivo e por disponibilizarem o ambiente de estudo perfeito ao longo de todos os anos da minha passagem pela universidade. Por último, agradeço à minha namorada por ter estado sempre presente nos bons e maus momentos, por me conseguir aturar, e por suportar as alturas em que não lhe dava a atenção merecida.

RESUMO

A necessidade de armazenar quantidades de informação cada vez maiores tem vindo a acentuar-se nos dias correntes. Conceitos como Internet das Coisas (IoT) e *Big Data*, atualmente em voga, vêm normalmente associados a muito dados motivando a procura por novas formas de armazenar e aceder a informação. Atualmente, milhares de aplicações recorrem a interfaces de sistemas de ficheiros para assegurar a persistência e o rápido acesso aos dados que geram. No entanto, as soluções de sistemas de ficheiros existentes apresentam configurações centralizadas ou orientadas a poucos nodos em redes controladas que se refletem numa escala e disponibilidade limitada.

De forma a abordar estes desafios, esta dissertação propõe o sistema LSFS, *Large Scale Filesystem*, que se trata de um sistema de ficheiros distribuído, compatível com a interface POSIX, capaz de escalar para redes de centenas a milhares de nodos heterogéneos, e ainda garantir elevada resiliência à falha dos seus nodos. Estas propriedades decorrem da sua arquitetura completamente descentralizada, *peer-to-peer*, e da utilização de protocolos de natureza epidémica. A aplicação destes protocolos no contexto de um sistema de ficheiros é nova, constituindo a principal contribuição desta dissertação.

Como outras contribuições, propomos um protótipo do sistema e uma avaliação experimental extensa conduzida com um caso de estudo real e num ambiente com 500 nodos. Os resultados mostram que o sistema LSFS consegue tolerar falhas de carácter catastrófico (p.ex. que contemplam 25% dos nodos totais) mantendo um desempenho de armazenamento estável ao longo do tempo.

***Keywords*—** Sistema de ficheiros, *Peer-to-peer*, Epidémico, Resiliência a falhas

ABSTRACT

The need to store ever-increasing amounts of data has become more pronounced in current days. Concepts such as the Internet of Things (IoT) and Big Data, currently in vogue, come usually associated with large amounts of data motivating the search for new ways to store and access it. Today, thousands of applications use file system interfaces to ensure persistence and fast access to the data they manage. However, existing file systems present centralized solutions or targeted to a reduced number of nodes, limiting scale and availability.

In order to tackle these challenges, this dissertation proposes LSFS, Large Scale Filesystem, which is a distributed file system, compatible with the POSIX interface, capable of scaling to networks of hundreds to thousands of heterogeneous devices, while ensuring high resilience to node-failure. These properties derive from its completely decentralized *peer-to-peer* architecture, and the use of epidemic protocols. The application of these protocols in the context of a file system is new and constitutes the main contribution of this thesis.

As further contributions, we propose a prototype of the system and an extensive experimental evaluation conducted with a real case study and in an environment of 500 network nodes. The results show that the LSFS system can tolerate catastrophic failures (e.g. that account for 25% of total nodes) while maintaining stable storage performance over time.

Keywords— File System, Peer-to-peer, Epidemic, Churn Resilience

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

SIGLAS

API Application Programming Interface.

FUSE Filesystem in Userspace.

IDC International Data Corporation.

IOT Internet of Things.

LSFS Large Scale FileSystem.

MIEI Mestrado Integrado em Engenharia Informática.

P2P Peer-to-peer.

POSIX Portable Operating System Interface.

PSS Peer Sampling Service.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UM Universidade do Minho.

1

INTRODUÇÃO

Nos dias correntes, com o advento da "Internet das Coisas"(IoT) e o vasto leque de aplicações enquadradas nesta vertente, tais como veículos conectados, casas e cidades inteligentes, segurança e automação de processos a nível empresarial, tem-se assistido a uma proliferação da quantidade de informação digital produzida. Um estudo da IDC prevê que 41.6 biliões de dispositivos IoT conectados produzirão 79.4 Zettabytes (ZB) de informação em 2025 [IDC,]. Toda esta informação surge com custos e novos desafios aliados ao suporte de armazenamento necessário. Sobretudo no que diz respeito a empresas, estas devem encontrar soluções que permitam acomodar todos os dados produzidos, tendo em consideração os custos envolvidos e as garantias que oferecem.

Se inicialmente os dados eram armazenados na periferia da rede, junto à sua fonte de produção, nomeadamente, junto aos utilizadores, seja em sistemas de armazenamento locais ou, mais tarde, partilhados e distribuídos em redes *peer-to-peer*, o surgimento da *computação em nuvem* e dos serviços por esta oferecidos fez inverter esta tendência. Assistiu-se a uma forte adoção da nuvem, tendo-se movimentado para a mesma quer dados quer aplicações, conceito que foi entitulado de "migração para a nuvem". Sobretudo no contexto das organizações, os serviços prestados pela nuvem permitiram reduzir custos, promover a rápida implementação de novas aplicações e evitar que tais organizações necessitem de manter a sua própria infraestrutura computacional.

No contexto da *Internet das coisas* esta tendência tem se mantido, existindo já um vasto número de aplicações IoT a utilizar a nuvem como recurso de computação universal e principal suporte de armazenamento. Esta estratégia tem sido levada a cabo quer a nível industrial (Altairs Smartworks [Altair SmartWorks,], GroveStreams [Grove Streams,], Xively [Xively,]), bem como académico [Gupta et al., 2014, Taher et al., 2019]. Contudo, tal alternativa não é ideal em todos os aspetos, apresentando alguns inconvenientes [Zhang et al., 2015]. As plataformas da *cloud* são normalmente hospedadas em grandes centros de dados afastados dos clientes traduzindo-se numa alta variabilidade da latência e largura de banda, bem como, normalmente, os provedores de *clouds* estabelecem limites para a largura de banda oferecida. No que respeita a segurança e disponibilidade dos dados, os clientes têm de confiar plenamente nos provedores de *cloud* para manter os seus dados seguros e acessíveis a todo o momento. Um outro inconveniente que poderia passar despercebido

mas que constitui grande parte da motivação para o retorno a uma gestão de dados na periferia da rede, é que a utilização da *cloud* apesar de ter permitido reduzir custos continua a possuí-los, quer no armazenamento bem como na transferência de informação entre os dispositivos e os serviços de nuvem[Sittón-Candanedo et al., 2019]. Num contexto IoT, estes custos são imensamente agravados devido à manipulação de uma quantidade de dados existentes numa ordem de grandeza significativamente maior.

1.1 PROBLEMA

Face à crescente necessidade de poder de armazenamento num mundo onde conceitos como a *Internet das Coisas* e *Big Data* se tornam uma realidade, é vital encontrar novas soluções que colmatem as desvantagens de depender dos sistemas de armazenamento disponibilizados pela nuvem.

Uma possível alternativa às soluções na nuvem encontra-se na adoção de sistemas de ficheiros distribuídos, como por exemplo o *Ceph*[Weil et al., 2006] ou o *HDFS*[Borthakur et al., 2008], implementados a nível de um cluster gerido localmente. Desta forma é possível obter igualmente um nível de disponibilidade considerável sem imposição de fortes restrições no que diz respeito à largura de banda e sem os custos de armazenamento e transferência de dados. Contudo, surgem outros custos, de manutenção do centro de dados, o investimento inicial de aquisição do equipamento, etc, que correspondem exatamente a grande parte da motivação de se ter migrado para a nuvem[Bigger,].

Contrariamente, uma outra alternativa seria anular totalmente estes custos e tirar proveito do espaço de armazenamento dos mais diversos dispositivos *edge* que constituem a infra-estrutura inerente ao funcionamento das próprias aplicações IoT, continuando, contudo, a oferecer o mesmo tipo de serviço, por exemplo, uma interface de sistema de ficheiros com as devidas garantias de alta disponibilidade.

Naturalmente, a implementação de um serviço como o sugerido está longe de ser trivial. Sistemas como o *HDFS* e *Ceph* garantem consistência forte ao utilizar um único servidor de metadados, no primeiro caso, e um algoritmo de consenso, no segundo caso. Este tipo de sistemas não escalam para redes em larga escala, nem se encontram adaptados para serem implementados em dispositivos heterogéneos onde a falha não é exceção mas sim a regra.

De forma a lidar com a fragilidade destes dispositivos é vital a utilização de algoritmos *peer-to-peer* que lidem exatamente com a propensão para a existência de elevados graus de agitação do sistema (termo conhecido na literatura como "churn" que corresponde a momentos de entrada e saída de nodos da rede, quer por ação voluntária ou por motivos de falha), fomentando, se necessário, uma reorganização dos nodos e uma redistribuição dos dados do sistema, permitindo, por sua vez, agilizar a sua rápida recuperação[Maia et al., 2014].

Na literatura atual existem já várias soluções de sistemas de ficheiros distribuídos distintas, contudo, nenhuma se encontra adaptada ao cenário pretendido. Alguns sistemas de ficheiros privilegiam a consistência em detrimento da disponibilidade adotando estratégias, essencialmente, baseadas no armazenamento diferenciado entre dados e metadados. Além de penalizarem a disponibilidade, a gestão independente dos metadados introduz, geralmente, gargalos no sistema que limitam a sua escalabilidade. Já outros sistemas procuram atingir maior disponibilidade e escalabilidade, não distinguindo dados de metadados a nível de armazenamento e adotando configurações completamente descentralizadas. Apesar de conseguirem escalar para redes de larga escala, o modo como organizam os seus nodos constituintes, torna-os vulneráveis perante situações de falha. Um problema não endereçado até à data na literatura consiste em combinar garantias de elevada resiliência, juntamente com a disponibilidade e escalabilidade já oferecidas. Tudo isto, oferecendo uma interface POSIX para não se ter de mudar a forma como a maioria das aplicações guardam e acedem a dados.

1.2 OBJETIVOS E CONTRIBUIÇÕES

O principal objetivo desta dissertação passa então por desenhar um novo sistema de ficheiros distribuído *peer-to-peer* capaz de tolerar níveis de falha elevados, mantendo o desempenho de armazenamento constante ou praticamente inalterado na presença das mesmas. Este sistema deve, ainda, se encontrar adaptado para instalação em infraestruturas IoT com centenas a milhares de nodos e fornecer uma interface POSIX de forma a ser directamente utilizável por diversas aplicações, isto é, sem precisar de recorrer a modificações de implementação das mesmas.

De forma a atingir este objetivo, esta dissertação apresenta três contribuições principais:

A primeira contribuição trata-se do desenho e conceção do Large Scale File System (LSFS), um sistema de ficheiros distribuído que utiliza algoritmos epidémicos em toda a comunicação entre os nodos, quer para gestão da organização da rede como para a gestão da distribuição, replicação e acesso aos dados. Nomeadamente, a utilização destes algoritmos é realizada pela integração do *DataFlasks*[Maia et al., 2014], uma *Key-Value Store* orientada para o tipo de cenário considerado, como substrato do nosso sistema de ficheiros. A natureza completamente descentralizada deste sistema dota-o ainda de alta escalabilidade, permitindo-o escalar para redes em larga escala.

Ainda, o LSFS apresenta uma interface compatível com POSIX[IEEE,]. Naturalmente, neste aspeto estabeleceu-se um compromisso entre consistência e disponibilidade dos dados, tendo-se relaxado algumas das garantias POSIX pelo simples facto de serem inatingíveis num cenário altamente distribuído. Estas considerações são discutidas em mais detalhe ao longo do documento.

Como segunda contribuição, é implementado um protótipo do sistema LSFS que, para além de implementar toda a lógica necessária para suportar uma interface de sistema de ficheiros, resulta também da reimplementação do *DataFlasks*. O *DataFlasks* foi originalmente implementado em Java orientado para ser executado com recurso à *Framework* de simulação Minha[Carvalho et al., 2011]. Através da sua reimplementação numa linguagem de mais baixo nível, C++, aliado à utilização de bibliotecas estáveis e maduras procurou-se aumentar o desempenho deste sistema bem com testar a sua aplicabilidade num contexto real, extrínseco à *Framework* de simulação Minha. Destacam-se, ainda, 4 grandes otimizações em relação ao *DataFlasks* original, nomeadamente, introdução de persistência, introdução de um balanceador de carga inteligente que adota uma estratégia eficiente de disseminação de pedidos, melhoria do algoritmo de construção de grupos com um protocolo de transferência de estado inicial e modificação do algoritmo de anti-entropia para uma recuperação mais rápida de dados e adaptada a um cenário de muitos dados.

Como terceira contribuição, esta dissertação pretende ainda mostrar o desempenho, resiliência e escalabilidade atingida por parte do protótipo desenvolvido. Para tal submeteu-se o sistema a uma extensa componente de testes, caracterizada pela utilização de diferentes configurações da rede de nodos e cargas de trabalho. A utilização de micro-cargas de stress quer de leitura como de escrita foram utilizadas para construir uma base de comparação, em termos de desempenho, com alternativas existentes de armazenamento local. Ainda, a avaliação recorreu a um caso de uso real, onde se procedeu ao treino de uma rede neuronal com recurso ao Tensorflow[Abadi et al., 2016], no qual se verificou que o nosso protótipo consegue escalar para instalações reais com 500 nodos mantendo um desempenho de aproximadamente um terço comparativamente a uma aprendizagem local. Por fim, os resultados evidenciam ainda a extrema resiliência do nosso sistema ao tolerar níveis de falha que chegam a 25% do número total de nodos da rede em intervalos constantes de 20 minutos, com penalizações mínimas ou nulas a nível de desempenho.

Em suma, esta dissertação contribui para o avanço científico na área de sistemas de gestão de dados, ao explorar uma nova estratégia, que consiste na utilização de algoritmos epidémicos na construção de sistemas de ficheiros distribuídos. Desta forma, esperamos com este projeto fornecer a motivação necessária para que novas otimizações e outros sistemas de ficheiros sejam desenvolvidos enredando no mesmo caminho.

1.3 Estrutura da dissertação

Esta dissertação encontra-se estruturada da seguinte forma. No Capítulo 2 são introduzidos conceitos e abordagens anteriores que consideramos necessários ao perfeito entendimento desta dissertação bem como se procura fornecer o contexto em que a mesma se desenvolve. Desta forma, começa-se por introduzir os sistemas *peer-to-peer* nomeadamente

no que diz respeito à sua evolução até às suas concretizações mais recentes. Depois, são alvo de estudo os sistemas de gestão de dados na sua generalidade sendo, de seguida, dado um especial foque aos sistemas de ficheiros distribuídos através da exposição e comparação de alguns sistemas de estado da arte. Por fim, é apresentado o sistema DataFlasks que constitui uma peça fundamental no desenho do LSFS, onde foi alvo de várias alterações. No capítulo em questão é providenciado o detalhe necessário à compreensão do funcionamento do mesmo, bem como, de cada uma das alterações introduzidas nos capítulos seguintes.

Os Capítulos 3 e 4 são dedicados à apresentação do LSFS. O primeiro introduz a arquitetura do sistema de ficheiros bem como a sua organização em blocos. O segundo a sua lógica de funcionamento e ainda as principais decisões de implementação aplicadas ao protótipo desenvolvido.

De seguida, no Capítulo 5 é descrita a componente de avaliação realizada sobre o LSFS. O capítulo começa por apresentar a metodologia de testes utilizada, incluindo uma descrição dos objetivos definidos, métricas colecionadas, bem como cargas de trabalho e *setups* experimentais utilizados. Posteriormente, apresentamos e analisamos os resultados obtidos para cada um dos conjuntos de experiências realizados.

Por último, no Capítulo 6 são expostas as conclusões obtidas decorrentes do trabalho realizado e apresentado o trabalho futuro.

2

ESTADO DA ARTE E CONTEXTUALIZAÇÃO

Este capítulo pretende contextualizar o leitor acerca do âmbito em que esta dissertação se insere, dotando-o do conhecimento necessário ao perfeito entendimento dos capítulos subsequentes. Com este objetivo, tendo em conta que o sistema a desenvolver é de natureza completamente descentralizada, *peer-to-peer*, será, primeiramente, fornecida um visão global do que é esta classe de sistemas, o seu modo de funcionamento e a sua evolução até serem integrados nos sistemas que atualmente utilizamos, sistemas estes como a grande parte dos que se incluem no trabalho relacionado. Depois, serão abordados os sistemas de armazenamento de dados no geral e as suas diferentes concretizações bem como as suas vantagens e desvantagens aliadas às garantias de disponibilidade e consistência que oferecem. De seguida, dar-se-á especial foque aos sistemas de ficheiros distribuídos existentes, realçando as diferenças relativamente à abordagem que será adotada pelo LSFS.

Por último, será apresentado o sistema DataFlasks. Este sistema desempenha uma papel fundamental na arquitetura do LSFS, constituindo o motivo da resiliência e escalabilidade que oferece.

2.1 SISTEMAS PEER-TO-PEER

Os sistemas *peer-to-peer* constituíram um grande avanço na área científica ligada ao armazenamento e partilha de dados. Estes são caracterizados pela sua natureza descentralizada onde cada nodo participa de igual forma na rede, evitando a necessidade de coordenação central realizada por servidores ou máquinas estáveis. No âmbito dos sistemas de armazenamento a sua aplicação permite atingir uma elevada escalabilidade e disponibilidade, que se encontra inacessível às soluções centralizadas.

Apesar de se poderem encontrar desenhos *P2P* em sistemas primordiais tais como *DNS*[Mockapetris and Dunlap, 1988], *Ficus*[Guy et al., 1990], *Bayou*[Terry et al., 1995], a atual expressão de conceito surge com o advento do *Napster*[Napster,] e *Seti@Home*[Anderson et al., 2002]. Contudo, nem um nem outro constituem um sistema completamente *P2P*. No *Seti@Home* a existência de um servidor central, permitia que blocos de dados fossem distribuídos pelos nodos para processamento, que comunicavam, posteriormente, os resulta-

dos de volta para o servidor. Nenhuma comunicação era realizada entre os nodos. Já no *Napster*, sistema criado com o objetivo de partilhar ficheiros mp3, apesar de também utilizar um servidor centralizado de gestão de metadados, a transferência efetiva dos ficheiros era realizada diretamente entre os nodos. Naturalmente, a existência de um órgão centralizado constituía não só um ponto único de falha como também um gargalo de desempenho.

Presumivelmente, a primeira solução *P2P* completamente descentralizada para partilha de ficheiros terá sido o *Gnutella*[Ripeanu, 2001]. A natureza descentralizada de soluções deste tipo induz um nível de complexidade acrescido no que respeita a procura de dados na rede. Tal, faz com que os dados tenham de ser distribuídos e a rede organizada de forma a permitir a rápida procura pelo conteúdo pretendido. Também a descoberta de nodos na rede é um problema fundamental a resolver por parte destes sistemas. O *Gnutella* procurava enfrentar tais problemas recorrendo à inundação de mensagens na rede. Cada nodo conhecia um conjunto de outros nodos aos quais realizavam os pedidos e, recursivamente, os faziam chegar aos restantes nodos da rede. Tal solução foi provada não escalar dado que as mensagens de controlo eram de tráfego quadrático no número de nodos da rede, consumindo grande parte da largura de banda. Algumas melhorias foram aplicadas ao *Gnutella* convergindo para uma arquitetura de *super-peers*[Singla and Rohrs, 2002], onde determinados nodos eram considerados especiais e preferenciais para o estabelecimento de conexões. Estes nodos preferenciais, também conhecidos como "*hubs*", mantinham ainda informação acerca do conteúdo que cada um dos nodos aos quais estabeleciam ligação partilhavam através da utilização de *bloom filters*[Tarkoma et al., 2011]. Tal permitia uma melhor pesquisa por conteúdos na rede.

Contudo, mesmo com as melhorias realizadas este sistema continuava a possuir algumas fragilidades. Os *super-peers* constituíam pontos de sobrecarga no sistema, o sistema encontrava-se propício a ataques uma vez que alguns nodos afirmavam possuir todo o conteúdo, e, apesar da arquitetura permitir restringir significativamente o tráfego de mensagens, esta continuava a ser essencialmente baseada na inundação de mensagens na rede, traduzindo-se numa sobreutilização da largura de banda da rede.

Relativamente à organização implícita da rede de nodos inerente às primeiras versões do *Gnutella*, esta é caracterizada como sendo não estruturada, cada nodo conhece um conjunto aleatório de outros nodos, os quais, por sua vez, têm conhecimento de um outro conjunto aleatório e finito de nodos. Esta configuração de rede partilha muita semelhanças com o que são chamados de grafos aleatórios ou *Érdos-Renyi*[Erdős and Rényi, 1960], herdando grande parte das suas propriedades. Nomeadamente, a rede torna-se muito robusta e extremamente resiliente a falhas, no sentido em que as falhas, mesmo as que envolvem mais do que metade dos nodos, não colocam a conectividade da rede em risco[Riviere and Voulgaris, 2011]. Por outro lado, este tipo de redes, não é, contudo, apropriado para motivos de encaminhamento de mensagens e pesquisas na rede dado não possuir uma

estrutura orientada para tal. De forma a colmatar as ineficiências neste campo, um novo grupo de sistemas distribuídos, nomeadamente as tabelas de *hash*, surgiram como soluções alternativas. Nestas, cada nodo possui os seus nodos vizinhos devidamente estruturados. A seleção de tais nodos é realizada segundo determinadas restrições que permitem que tal estrutura seja verificada. As redes formadas por estes sistemas, por oposição às primeiras, são denominadas de redes estruturadas.

As primeiras quatro tabelas de *hash* distribuídas foram introduzidas quase simultaneamente em 2001, sendo estas *CAN*[Ratnasamy et al., 2001], *Chord*[Stoica et al., 2001], *Pastry*[Rowstron and Druschel, 2001] e *Tapestry*[Zhao et al., 2004]. O *Chord* foi possivelmente destas a que teve um maior impacto pelo que servirá de exemplo. No que diz respeito à organização dos nodos na rede e ao mapeamento das chaves pelos nodos, todos estes protocolos baseiam a sua atuação em *consistent hashing*, isto é, tanto aos identificadores dos nodos como às chaves, através de um processo de *hashing*, é lhes atribuído um identificador pertencente a um determinado espaço de identificadores. No *Chord*, de acordo com o seu identificador, cada nodo ocupa uma posição num anel, ficando responsável por todas as chaves que mapeiam no espaço entre o próprio e o seu nodo antecessor. Cada nodo mantém um conhecimento O(log n) do sistema, sendo o encaminhamento também realizado em O(log n) passos, por comunicações sucessivas com nodos cujo identificador se aproxima cada vez mais da chave pretendida. Atualmente, uma das tabelas de *hash* distribuidas mais maduras e largamente utilizadas é o Kademlia[Maymounkov and Mazieres, 2002]. Face às restantes previamente enumeradas, utiliza o *XOR* como métrica de distância entre dois pontos no espaço de identificadores. Sendo esta uma métrica simétrica permite ultrapassar a rigidez das tabelas de encaminhamento encontradas por exemplo no *Chord*, bem como tornar o encaminhamento mais eficiente [Maymounkov and Mazieres, 2002]. O *Kademlia* diferencia-se ainda por permitir encaminhamento paralelo, contactando, por cada passo do protocolo, não um mas um conjunto de nodos que aproximam à chave pretendida, tornando-o mais resiliente à falha de nodos no sistema. Contudo, apesar de mais resiliente, essa resiliência encontra-se limitada pela necessidade das tabelas de *hash* distribuídas em geral serem obrigadas a manter um estrutura específica. A entrada e saída de nodos faz com que a preservação da estrutura em questão exija grandes custos de manutenção, já que, tomando como exemplo um nodo que falhe, este não pode ser simplesmente substituído por um qualquer nodo aleatório, havendo a necessidade de recorrer à infraestrutura de *routing* para recuperar da falha. Estudos anteriores [Rhea et al., 2004] mostram as dificuldades que sistemas como este, suportados por redes sobrepostas estruturadas, possuem em lidar com níveis elevados de agitação do sistema.

Um novo conjunto de protocolos, aos quais se denominaram de *gossip* ou epidémicos[Riviere and Voulgaris, 2011], ficaram conhecidos pela sua capacidade de permitirem atingir resiliência em redes de muita larga escala, onde a probabilidade de um nodo entrar e sair do

sistema, ou meramente falhar, é muito elevada. Esta resiliência, característica, essencialmente, do subconjunto destes algoritmos baseados em redes sobrepostas não estruturadas, advém da própria organização da rede bem como na proatividade em manter o conhecimento da rede em cada nodo devidamente atualizado. Estas características dotaram estes algoritmos de excelentes propriedades de auto-organização, auto-estabilização e auto-recuperação, tendo já sido aplicados com sucesso na resolução diferentes problemas de sistemas distribuídos[Demers et al., 1988, Van Renesse et al., 1998, Van Renesse et al., 2003, Leitao et al., 2010, Maia et al., 2014].

A forma como estes algoritmos atuam é semelhante ao modo como os rumores se espalham, isto é, cada nodo periodicamente seleciona um outro nodo aleatório com o qual troca informação. Havendo a necessidade de selecionar um nodo aleatório da rede e sendo impraticável a manutenção de um conhecimento global da constituição da mesma, surge uma classe de protocolos, denominada de *Peer Sampling Service*, que endereçam este problema. Estes protocolos, de que *Cyclon*[Voulgaris et al., 2005] e o *Newscast*[Voulgaris et al., 2003] são exemplo, baseiam a sua atuação na manutenção, em cada nodo, de uma vista parcial, dinâmica e aleatória de outros nodos da rede. Desta forma, selecionar um nodo aleatório da rede é equivalente a selecionar um nodo aleatório da vista local[Riviere and Voulgaris, 2011]. Estes protocolos são por si só protocolos de gossip, recorrendo a interações entre nodos para trocar informações acerca da constituição da rede e, por sua vez, manterem a sua vista local atualizada.

Para além de permitirem a seleção de nodos aleatórios da rede, os protocolos de *Peer Sampling Service* são conhecidos por destes emergirem as já referidas redes sobrepostas não estruturadas, como a encontrada no desenho do *Gnutella*. Tal como tínhamos visto, estas redes, apesar de oferecerem alta resiliência, são caracterizadas pela sua organização inadequada para o encaminhamento de mensagens. Diferentes algoritmos de gossip tornam, contudo, possível tirar proveito destas redes para diferentes utilidades, agregações[Jelasity et al., 2005], disseminação de mensagens[Eugster et al., 2003], etc. Um destes casos de uso dá-se pelo nome de *overlay slicing*[Fernández et al., 2007, Gramoli et al., 2008] e refere-se à divisão da rede sobreposta em diferentes grupos de tamanho relativo, sem a necessidade de se conhecer o tamanho real da rede. Esta divisão dos nodos em grupos possui várias aplicações podendo ser utilizada para configurar um encaminhamento mais eficiente entre nodos de diferentes grupos. Tal conceito foi já aplicado com sucesso numa base de dados chave-valor descentralizada, denominada DataFlasks[Maia et al., 2014].

2.2 Sistemas de gestão de dados

Os sistemas de gestão de dados encontram-se em constante evolução ao longo do tempo. Os sistemas de ficheiros em particular, representam uma classe de sistemas de gestão de

dados que desde cedo teve uma larga adesão, constituindo um dos principais meios de armazenamento e pesquisa de informação digital. A evolução neste campo trouxe uma grande diversidade de novas alternativas, igualmente viáveis, distinguindo-se desde a forma como armazenam os dados e providenciam o seu acesso e pesquisa, à organização que apresentam.

Posto isto, à parte de sistemas de ficheiros, é possível identificar dois outros tipos de abstração de armazenamento de dados largamente utilizados, sendo estes as bases de dados chave-valor e as bases de dados relacionais. Estes três grandes grupos de sistemas de gestão de dados caracterizam-se por oferecer interfaces distintas às aplicações que os utilizam, servindo diferentes propósitos e adequando-se a contextos distintos.

Dentro de cada tipo de abstração, uma dada solução pode ser caracterizada pelo seu carácter organizacional como sendo centralizada, distribuída estruturada e distribuída não estruturada. Um sistema é considerado centralizado caso se encontre adaptado para ser instalado numa única máquina e distribuído no caso de instalações multi-máquina. Como veremos, existem ainda soluções que se fazem dispôr das duas vertentes, permitindo quer a instalação numa máquina como em várias. Dentro das soluções distribuídas, de acordo com a sua arquitetura, as máquinas podem organizar-se em redes de acordo com uma estrutura específica, resultando num sistema distribuído estruturado, ou não existir qualquer estrutura predeterminada, sendo classificado de sistema distribuído não estruturado.

	Sistema de Ficheiros	**Key-Value Store**	**Base de dados relacional**
Centralizado	✓	✓	✓
Distribuído estruturado	✓	✓	✓
Distribuído não estruturado	✗	✓	✗

Tabela 1: Visão global de sistemas de armazenamento de dados

De acordo com a classificação proposta, realizamos uma análise do estado da arte nestes sistemas, relativamente à existência ou não de sistemas que mapeiam em cada uma das categorias. Na Tabela 1 apresentam-se os resultados obtidos, correspondendo o marcador ✓ à existência de sistemas exemplares da categoria e o ✗ à sua ausência.

Como é possível observar, cada tipo de abstração dispõe de soluções centralizadas. Exemplos como o *RocksDb*[RocksDB,] no âmbito das bases de dados chave-valor, *SQL Server*[Delaney, 2000], nas bases de dados relacionais e *Ext4*[Mathur et al., 2007] nos sistemas de ficheiros são, naturalmente, soluções que não são resilientes, onde a existência de apenas uma máquina introduz um ponto único de falha no sistema. Para além disso, a natureza centralizada destes sistemas constitui um forte entrave à escalabilidade. Estes conseguem apenas escalar de forma vertical com os recursos do servidor onde se encontram instalados.

Diversos motivos, desde o próprio aumento da resiliência, melhores garantias de disponibilidade dos dados, escalabilidade ou até a mera necessidade de partilha dos dados, como observado para os sistemas de ficheiros, fez com que fossem igualmente desenvolvidas soluções distribuídas. Dentro das bases de dados relacionais soluções como o *MySQL*[MySQL,] ou o *PostgreSQL*[PostgreSQL,], inicialmente concebidos para instalação numa única máquina, implementaram estratégias de replicação de dados, cuja finalidade principal é assegurar a existência de múltiplas cópias consistentes dos dados. Este tipo de sistemas privilegiam pois a consistência dos dados em detrimento da disponibilidade do sistema, fazendo-se auxiliar para tal de protocolos de coordenação. A coordenação é, contudo, custosa e impede estes sistemas, uma vez mais, de atingir a escala pretendida. De facto, de acordo com o teorema CAP[Brewer, 2012], nenhum sistema distribuído consegue garantir ambos consistência e disponibilidade enquanto tolerante a partições da rede. Uma vez que a tolerância a partições na rede é estritamente necessária, já que a existência das mesmas é um dado adquirido, cabe a cada um dos sistemas estabelecer um compromisso relativamente às restantes duas garantias. A distribuição oferecida através destes mecanismos de replicação faz-se segundo topologias bem conhecidas, constituindo, portanto, exemplos de distribuição estruturada. No que respeita a distribuição não estruturada do que sabemos ainda não foi aplicada a nenhuma base de dados relacional.

A distribuição encontrada nas bases de dados chave-valor concretizou-se percorrendo um caminho distinto ao dos tradicionais sistemas de gestão de base de dados relacionais. Estas procuraram atingir alta escalabilidade e disponibilidade, comprometendo a forte consistência oferecida pelos sistemas anteriormente abordados. Esta escolha é facilmente justificável pela necessidade, nos tempos correntes, dos mais variados serviços lidarem com um elevado número de utilizadores pelo que devem garantir elevado desempenho. Também a disponibilidade constitui um fator privilegiado pelos utilizadores no acesso aos serviços.

A abordagem adotada pela grande maioria das bases de dados chave-valor, tais como *PAST*[Druschel and Rowstron, 2001], *Dynamo*[DeCandia et al., 2007], *Cassandra*[Lakshman and Malik, 2010] e *Riak*[Riak,], assenta na utilização de tabelas de hash distribuídas para balancear os dados pelos nodos e permitir a sua localização eficiente. Distinguem-se, essencialmente, pelo modelo de dados considerado e pela tabela de hash distribuída que utilizam. Os protocolos de natureza *peer-to-peer* associados à utilização destas estruturas dotam estes sistemas de grande escalabilidade. As tabelas de hash distribuídas são por si só organizações estruturadas, pelo que todos estes sistemas, ao fazerem uso das mesmas, incluem-se, naturalmente, na categoria de distribuídos estruturados.

Uma base de dados chave-valor que se distingue de todas as outras é o *DataFlasks*[Maia et al., 2014]. Por oposição às demais, serve-se unicamente de protocolos *peer-to-peer* não estruturados, baseados em algoritmos epidémicos, revogando o uso de uma tabela de hash distribuída. Cada nodo no *DataFlasks* tem conhecimento, a cada momento, de um

sub-conjunto aleatório de outros nodos da rede, pelo que a rede formada é de natureza não estruturada. Trata-se, desta forma, de um exemplo de um sistema distribuído não estruturado. Os protocolos envolvidos, tal como mencionado na secção anterior, conseguem tolerar níveis mais elevados de agitação do sistema ao atuar de modo proativo na sua recuperação, ao custo de um ligeiro aumento de mensagens na rede. A utilização destes protocolos constitui pois o fator diferenciador para a resiliência que o *DataFlasks* oferece. Relativamente à forma como os dados são localizados, recorre a um algoritmo de divisão dinâmica dos nodos em grupos, e a técnicas de *hashing* com o intuito de mapear chaves nesses grupos. Este sistema apresenta-se como um sistema de armazenamento de dados adequado para larga escala, coincidindo com os cenários alvo deste projeto.

Contudo, sendo um dos requisitos do projeto a desenvolver nesta dissertação oferecer uma interface de sistema de ficheiros por oposição à API *Put-Get* partilhada pelas bases de dados chave-valor, faz com que estas se encontrem igualmente fora do domínio considerado. Ainda assim, perceber a vantagem que o *DataFlasks* apresenta face às bases de dados chave-valor concorrentes, bem como, entender como é que o modo de funcionamento do mesmo é justificativo de tal, é de suma importância para perceber como é que o nosso próprio sistema funciona e a motivação envolvida na sua concretização.

É, de facto, nos sistemas de ficheiros que se encontra a motivação inerente ao desenvolvimento deste projeto. A interface de sistema de ficheiros é, possivelmente, a abstração de sistema de gestão de dados mais utilizada, servindo de suporte a milhões de aplicações. Também esta classe de sistemas de gestão de dados se faz contemplar de distribuição, constituindo um tópico de estudo que data já algum tempo na comunidade científica. A Secção 2.3 é dedicada a apresentar como é que a distribuição se processou nesta classe de sistemas.

2.3 Sistemas de ficheiros distribuídos

Tal como enunciado na Secção 2.2, desde cedo se procurou introduzir distribuição aos sistemas de ficheiros. Inicialmente a motivação partia de permitir a partilha de ficheiros e diretorias entre máquinas conectadas em rede. Partindo de um modelo cliente-servidor, a Sun Microsystems, Inc. desenvolveu em 1984, o *NFS*[Sandberg et al., 1985], no formato de um protocolo, que possibilitou o acesso a um sistema de ficheiros remoto. A partir daí estes sistemas evoluíram no sentido de facultar o armazenamento de grandes quantidades de dados bem como aumentar a resiliência dos mesmos.

O *Andrew File System*[Howard et al., 1988] surge então, como um dos primeiros sistemas de ficheiros a efetuar a distribuição da sua árvore de diretorias por vários servidores. Este delegava a posse e responsabilidade de determinadas sub-árvores do sistema de ficheiros a diferentes servidores, efetuava caching e utilizava primitivas de locking nos nodos

responsáveis pelos ficheiros de forma a assegurar a consistência dos dados. Já, o *Ficus*[Guy et al., 1990], um outro sistema de ficheiros, foi desenvolvido com a assunção de que "network partition is a fact of life"[Popek et al., 1990], bem como apoiava a necessidade de garantir disponibilidade, pelo que adotou o conceito de operações desconectadas, permitindo que os clientes realizassem operações sobre os dados mesmo que se encontrem temporariamente desconectados da rede, implicando, posteriormente, uma resolução de conflitos.

Três outros sistemas que merecem algum destaque são o *HDFS*[Borthakur et al., 2008], *GFS*[Ghemawat et al., 2003] e o *Ceph*[Weil et al., 2006]. Estes procuraram atingir forte consistência e escalabilidade através de uma gestão separada de dados e metadados. Tanto no *HDFS* como no *GFS*, os dados são distribuídos e replicados por diferentes servidores de dados. Estes diferem entre si na forma como gerem os metadados. No caso do *HDFS* existe um nodo que é responsável por armazenar e gerir os metadados, enquanto que no *GFS* tal nodo é também responsável por replicar tais metadados num conjunto de réplicas utilizando replicação de máquina de estados e ordenação total de comandos com recurso ao *Paxos*. Já o *Ceph* vai mais além e supera o gargalo de utilizar um servidor de metadados centralizado ao adotar uma arquitetura de metadados distribuída.

Estes sistemas de ficheiros foram, contudo, desenhados para serem utilizados num contexto de um centro de dados, onde a falha é a exceção e não a regra. Por esta razão não se adequam a redes de larga escala, formadas por dispositivos heterogéneos e com capacidades de armazenamento reduzidas, como as que foram objetivadas pelo sistema retratado nesta dissertação. Outros sistemas foram especialmente desenhados tendo em vista cenários como o descrito, tendo estes por base protocolos e substratos de natureza *peer-to-peer*.

Como exemplos de sistemas de ficheiros que recaem nesta categoria, destacam-se o *CFS*[Dabek et al., 2001], *Ivy*[Muthitacharoen et al., 2002] e *Pastis*[Picconi et al., 2005]. Todos estes sistemas não possuem nodos específicos para lidar com metadados, mas armazenam quer os dados quer os metadados num mesmo substrato. Quanto ao *CFS* e ao *Ivy* utilizam ambos o mesmo substrato, a *DHash*[Dabek et al., 2004], como camada de gestão de persistência e disponibilidade, efetuando replicação. Por sua vez, a *DHash* recorre ao *Chord* para fazer o mapeamento de chaves para nodos e localizar os dados. Estes dois sistemas são, contudo, bastante distintos. O *CFS* apresenta-se como um sistema de ficheiros *read-only*, isto é, permite apenas que as escritas sejam realizadas por um único utilizador (o "dono"). O acesso ao sistema de ficheiros é realizado através de um bloco raiz indexado pela chave pública do dono. Perante a realização de alterações ao sistema de ficheiros, este bloco, que possui um *timestamp*, é novamente inserido com o mesmo identificador na *DHash*. Trata-se, desta forma, de um caso de consistência eventual porque um utilizador pode ter uma referência para uma versão desatualizada do sistema de ficheiros. Tal versão garante-se, contudo, ser internamente consistente.

Já o *Ivy* permite escritas de vários utilizadores. Este, adota um estratégia completamente distinta do *CFS* e do *Pastis* armazenando na sua tabela de hash distribuída, em vez de blocos de dados e metadados, apenas logs, um por utilizador, das alterações realizadas ao sistema de ficheiros. De forma a satisfazer operações de leitura, um utilizador tem de aceder a todos os logs de forma a reconstruir o estado corrente do sistema de ficheiros, o que o torna pouco escalável.

O *Pastis* utiliza também como substrato uma tabela de hash distribuída, a *PAST*, apesar de estruturalmente semelhante ao *CFS*, diferencia-se do mesmo por permitir escritas de vários utilizadores. Este sistema tal como *Ivy* oferece consistência eventual com e sem semânticas *close-to-open*. As semânticas *close-to-open* foram inicialmente introduzidas, juntamente com mecanismos de caching do lado do cliente, pelo *Andrew Filesystem* e consistem em apenas dar ordem para persistir as alterações realizadas num ficheiro, após o descritor do mesmo ter sido fechado[Osadzinski, 1988].

Atualmente, é possível ainda encontrar alguns projetos *open-source* de abstrações de sistema de ficheiros sobre bases de dados chave-valor como o *Cassandra* e *Dynamo* [CassFS, , DynamoFS,], com recurso ao *Fuse*. Naturalmente, a consistência obtida está inerente ao substrato considerado. Quer o *Dynamo*, quer o *Cassandra* por defeito oferecem consistência eventual, apesar de, mais recentemente, incluírem formas de especificar forte consistência para determinadas operações, trocando consistência por disponibilidade.

Todas as soluções apontadas, aplicam distribuição segundo organizações bem estruturadas. Por exemplo, no caso dos sistemas de armazenamento baseados em protocolos *peer-to-peer* apresentados, todos eles, direta ou indiretamente, fazem uso de uma tabela de hash distribuída como substrato. Face à classificação proposta na Secção 2.2, estes pertencem à categoria de sistemas distribuídos estruturados. Ainda, relativamente à Tabela 1, encontrada na mesma secção, é observável a presumível inexistência de soluções distribuídas não estruturadas para sistemas de ficheiros. Tendo-se identificado a vantagem que tais alternativas, como o *DataFlasks*, possuem comparativamente às demais estruturadas, abre-se aqui uma oportunidade para aplicar o mesmo conceito aos sistemas de ficheiros. É exatamente este ponto o foco do LSFS, isto é, ultrapassar as limitações de escalabilidade características dos sistemas centralizados e base de dados relacionais, e garantir a disponibilidade e tolerância a falhas obtida por sistemas como o *DataFlasks*, disponibilizando uma API de sistema de ficheiros.

2.4 DATAFLASKS

O DataFlasks constitui uma peça fundamental no desenho do LSFS. Este, como veremos no Capítulo 3, representa o seu substrato de armazenamento. De forma a servir de suporte ao sistema de ficheiros, determinadas modificações foram realizadas à API do DataFlasks

bem como aos seus componentes, pelo que se considera necessário possuir um perfeito entendimento do mesmo em ordem a justificar as modificações realizadas, perceber o próprio funcionamento do LSFS, como também o motivo da resiliência e escalabilidade que oferece. Este capítulo será, pois, focado em caracterizar o arquitetura do DataFlasks, expondo os componentes que o formam, bem como o seu respetivo funcionamento.

2.4.1 *Arquitetura do DataFlasks*

O DataFlasks trata-se de uma base de dados chave-valor, caracterizada como sendo completamente descentralizada, sem coordenação, escalável e robusta, orientada para cenários de vários milhares de nodos e elevados níveis de agitação do sistema. Na figura 1 apresenta-se a arquitetura do mesmo.

Este é composto por duas entidades distintas sendo estas, o cliente e os nodos de armazenamento. A lógica principal do DataFlasks encontra-se nos nodos de armazenamento que, individualmente, participam em protocolos que permitem a constituição e definição de uma rede, a disseminação de mensagens pela mesma, bem como asseguram o acesso e persistência de dados respeitando determinado nível de replicação.

O cliente age como mero ponto de entrada no acesso aos dados, oferecendo uma interface chave-valor baseada em métodos put e get de blocos, a qual é detalhada neste capítulo. O seu funcionamento passa pela manutenção de conhecimento acerca da existência de um conjunto de nodos da rede pelos quais distribui os pedidos que são realizados à sua API. Ambas estas tarefas encontram-se ao encargo de um componente denominado de "Load Balancer".

Já, os nodos de armazenamento possuem o seu funcionamento baseado na atuação conjunta das duas camadas que o formam, a de gestão da rede e a de gestão de dados, sendo a segunda diretamente dependente da primeira.

A Camada de gestão da rede, tal como o nome indica, é responsável pela constituição e manutenção da rede de nodos. Esta é formada por dois componentes que são o "Peer Sampling Service" e o "Group Construction". O Peer Sampling Service, PSS, tem como principal função a de, estabelecendo o paralelismo entre a rede e um grafo, determinar que nodos se encontram conectados a que nodos, isto é, a de criar na rede física uma rede sobreposta virtual. À medida que os nodos entram e saem da sistema deverá incorporá-los ou removê-los da rede. O componente Group Construction, por outro lado, encontra-se encarregue de acrescentar uma nova camada de abstração à rede, subdividindo-a em grupos de tamanho semelhante. Esta divisão da rede de nodos em grupos constitui um das tarefas mais importantes no DataFlasks, sendo toda a gestão da replicação baseada nestes mesmos grupos. Estes grupos são, pois, grupos de replicação. Os nodos pertencentes a um mesmo grupo encontram-se responsáveis por armazenar o mesmo conjunto de chaves. Para

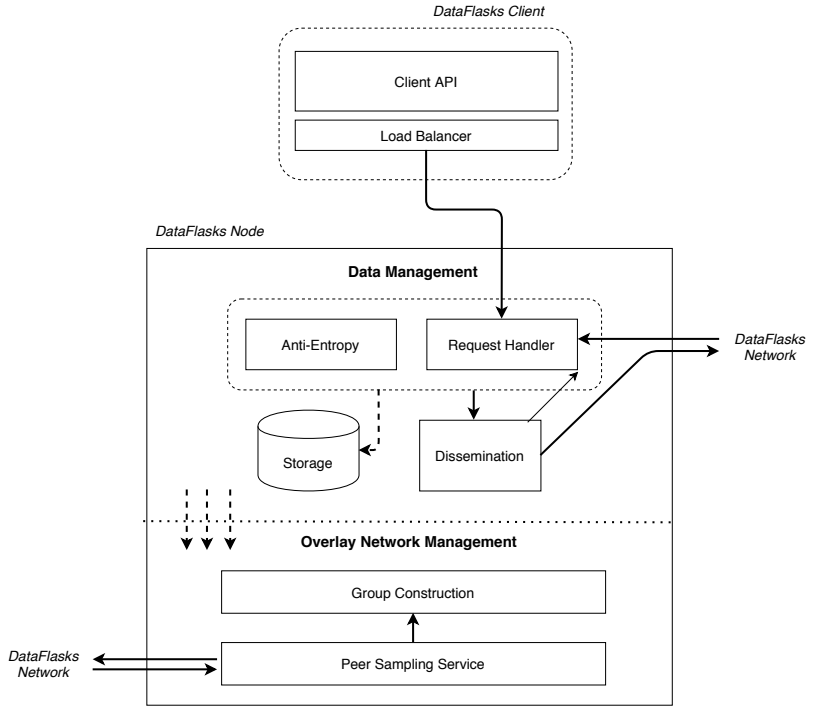


Figura 1: Arquitetura do DataFlasks

além disso, os grupos são ainda utilizados com o intuito de tornar o encaminhamento de mensagens mais eficiente.

À camada de Gestão de dados, por outro lado, são lhe atribuídas as tarefas relacionadas com o processamento dos diferentes pedidos provenientes do cliente DataFlasks. Tal processamento implica a persistência e acesso local de/a determinados blocos de dados, bem como a disseminação de pedidos na rede para que estes possam ser processados por outros nodos. Desmembrando esta camada, verifica-se que a mesma tem por base os componentes "Storage" e "Dissemination".

O componente "Storage" encarrega-se de armazenar os blocos de dados referentes a chaves alocadas ao nodo em questão, bem como de garantir o seu acesso. Possui uma API bem definida, possibilitando a especificação de diferentes implementações da mesma e, por sua vez, que os dados possam ser armazenados em diferentes formatos e meios de armazenamento.

O componente "Dissemination" serve o propósito de propagar mensagens na rede. Para tal serve-se da camada de gestão de rede para realizar o respetivo encaminhamento de mensagens que tanto pode ser aleatório como seletivo, isto é, orientado a um grupo ou nodo específico. As mensagens enviadas podem provir de dois outros componentes que fazem parte desta camada, sendo estes "Anti-Entropy" e "Request Handler".

O componente "Anti-Entropy" tem duas funções bem definidas. A primeira consiste em garantir que um nodo que entre pela primeira vez no sistema recupera todas as chaves

de que é responsável em função do grupo de replicação em que se inclui. A segunda, em preservar a consistência das chaves armazenadas pelos diferentes nodos de um mesmo grupo de replicação, realizando a recuperação, em cada nodo, de possíveis blocos de dados referentes a chaves que não foram processadas por parte do mesmo.

Já, o componente "Request Handler" é quem possui a lógica necessária ao processamento dos diferentes pedidos de dados, pedidos esses que lhe chegam quer do cliente DataFlasks, quer de outros nodos de armazenamento. O processamento de cada pedido é realizado através de um contacto directo e coordenado com os componentes "Storage" e "Dissemination".

2.4.2 *Gestão da rede*

Num mundo ideal, cada nodo possuiria um conhecimento completo da rede, isto é, a cada momento saberia exatamente que nodos fariam parte da rede e, por sua vez, conseguiria comunicar diretamente com os nodos que possuem a informação necessária à satisfação de cada um dos diferentes pedidos do cliente. Tal assunção está longe de ser aceitável quando nos referimos a uma rede de larga escala dada a dimensão da mesma bem como o elevado dinamismo que a caracteriza.

Os protocolos de *Peer Sampling Service* surgem aqui como uma forma de ultrapassar este problema. Estes constituem uma subclasse de protocolos epidémicos que implementam a descoberta de nodos e providenciam a cada nodo uma *stream* contínua de nodos aleatórios da rede, tal significa, que cada nodo tem acesso, a cada momento, a um subconjunto limitado de outros nodos da rede, ao qual se intitula de vista do sistema ou vista local. De um modo geral, estes protocolos funcionam da seguinte forma: cada nodo mantém a sua vista local de outros nodos que conhece do sistema; periodicamente, contacta um destes nodos e ambos trocam informação acerca das suas vistas; por fim atualizam a sua vista com a informação recebida.

O DataFlasks utiliza como algoritmo de *Peer Sampling Service* uma versão do protocolo Cyclon[Voulgaris et al., 2005]. Este protocolo acrescenta ao algoritmo especificado um fator idade por elemento da vista. Tal fator é muito importante para que se possa lidar com o dinamismo da rede, prevenindo que registos correspondentes a nodos que falharam ou deixaram a rede circulem indefinidamente na mesma.

2.4.3 *Construção de grupos e replicação*

A divisão dos nodos em grupos é no DataFlasks modelada como um problema denominado na literatura de "distributed slicing", que corresponde à operação de particionar um dado sistema distribuído num conjunto de k grupos de acordo com um determinado critério

específico a cada nodo[Fernández et al., 2007, Gramoli et al., 2009]. O desafio encontra-se em fazê-lo de forma autónoma e sem conhecimento do tamanho do sistema. No caso particular do DataFlasks, a divisão em grupos encontra-se dependente do tamanho pretendido para cada grupo, o que em termos práticos corresponde ao fator de replicação desejado para o sistema. Este fator de replicação, por motivos de estabilidade, corresponde na verdade a um intervalo, isto é, é limitado por um tamanho de grupo mínimo e máximo.

Posto isto, resumidamente, o algoritmo de construção de grupos desenvolve-se da seguinte forma. Cada nodo possui uma determinada posição numérica no espaço de endereçamento $]0,1]$. Tem ainda acesso a um conjunto de referências para os outros elementos do grupo em que se inclui, a que vamos chamar de vista de grupo, bem como, realiza um estimativa do número de grupos existente. Aquando da sua inicialização este não conhece nenhum outro nodo do sistema pelo que a sua vista de grupo encontra-se vazia e a estimativa que faz para o número de grupos é de um único grupo, o grupo formado por ele próprio, que ocupa todo o espaço de endereçamento $]0,1]$. À medida que vai conhecendo novos nodos, por mecanismos de PSS, vai adicionando-os ao seu grupo. Quando o número de nodos que inclui na vista de grupo supera o tamanho de grupo máximo definido, este dobra a sua estimativa para o número de grupos existente, passando a 2 grupos (ocorre um particionamento do grupo). Neste momento, o espaço de endereçamento é dividido em duas fatias iguais, sendo que, consoante a fatia em que a sua posição o coloca, este remove da vista de grupo todos os nodos que não partilham essa mesma fatia, isto é, não partilham o seu grupo. Por outro lado, cada referência para um determinado nodo do grupo possui uma idade associada que vai sucessivamente aumentando enquanto não for refrescada por outra mensagem que a inclua. Caso um determinado nodo do grupo deixe o sistema, a sua referência deixará de ser refrescada e será eliminada da vista de grupo assim que ultrapasse uma determinada idade máxima. Se o tamanho da vista de grupo passar a ser menor que o tamanho de grupo mínimo definido ocorre o processo inverso ao anteriormente descrito, isto é, a estimativa do número de grupos passa a metade ocorrendo uma junção de grupos. O algoritmo de construção de grupos converge, pois, por sucessivos particionamentos e junções de grupos, de forma a manter um tamanho de grupo e, por sua vez, um fator de replicação compreendido nos limites impostos. Face ao descrito, é necessário reter alguns aspetos importantes: o número de grupos é sempre uma potência de dois (1, 2, 4, 8, ...) correspondendo à divisão do espaço de endereçamento dos nodos em fatias de igual tamanho; o algoritmo lida com o dinamismo da rede através do particionamento e junção de grupos; o particionamento de grupos subdivide cada grupo em dois novos grupos e a junção de grupos agrupa grupos contíguos 2 a 2.

Tendo tal em conta, perceber como é que se realiza a distribuição de chaves é trivial. Recorre-se à definição de uma de função de *hash* que permite mapear de forma uniforme cada chave a armazenar no espaço de endereçamento dos nodos ($]0,1]$). Este espaço de

endereçamento, tal como descrito, é dividido em grupos de replicação de igual tamanho, pelo que fica ao encargo dos nodos inseridos em cada grupo de replicação armazenar as chaves que se posicionem no intervalo que gerem.

No DataFlasks[Maia et al., 2014], é possível encontrar todo o estudo realizado em torno deste algoritmo bem como uma prova de correção do mesmo.

2.4.4 *API e Processamento de Pedidos*

O DataFlasks tal como anteriormente mencionado é caracterizado por não possuir qualquer tipo de coordenação. Este introduziu o conceito de versionamento de chaves de forma a evitar lidar com conflitos na modificação de dados. Cada pedido é, pois, realizado sobre um determinado par (chave, versão) e qualquer modificação num objeto de dados pressupõe a utilização de uma nova versão da mesma chave. Assume-se ainda que é dever da aplicação garantir a consistência das escritas, prevenindo a ocorrência de duas escritas simultâneas sobre o mesmo par (chave, versão). Posto isto, a API do DataFlasks exporta duas operações:

```
put(chave, objeto, versao)
objeto get(chave, versao)
```

A operação de **put** recebe como *input* a chave, o objeto de dados e a versão do objeto. Já a operação de **get** exige que seja especificada tanto a chave como a versão desejada. Assume-se a unicidade de cada triplo (chave, objeto, versão).

Perante a receção de um pedido put/get, cabe ao "Load Balancer" a propagação do pedido na rede. Este, que corre a sua própria instância do algoritmo de PSS, seleciona aleatoriamente um nodo da sua vista para iniciar a disseminação do mesmo. A partir deste, o pedido é propagado na rede num formato *infect and die*[Eugster et al., 2004]. Nos diferentes nodos de armazenamento, o componente "Request Handler"encarrega-se de processar cada pedido, bem como dar continuidade à disseminação dos mesmos. Este processamento é realizado da seguinte forma. Para um pedido get, o nodo verifica se a chave referenciada se encontra armazenada localmente. Em caso afirmativo, responde ao cliente com o bloco requerido, caso contrário, dissemina os pedidos pelos nodos da sua vista. Para um pedido put, no caso da chave mapear no seu grupo de replicação, armazena-a e envia uma mensagem de confirmação ao cliente. Dissemina, ainda, o pedido no seu grupo de replicação para que os restantes nodos a armazenem. No caso da chave não mapear no seu grupo, dissemina a pedido na sua vista à semelhança do que acontece para os pedidos get.

2.4.5 *Recuperação de dados*

O DataFlasks trata-se de um sistema inevitavelmente consistente. Cada pedido realizado ao mesmo pode ser respondido por uma ou mais réplicas dentro de um mesmo grupo de replicação. Estas réplicas podem, contudo, se encontrar num estado temporariamente inconsistente na medida em que determinadas atualizações podem apenas ter chegado a um subconjunto das mesmas, por exemplo devido a falhas ocorridas na rede (o tráfego de mensagens no DataFlasks é realizado por UDP), ou na eventualidade de ocorrência de uma falha numa escrita de um nodo.

Os nodos no DataFlasks têm, portanto, de possuir um mecanismos para recuperar de situações como estas de forma a permitir que as réplicas num mesmo grupo convirjam para um mesmo estado, isto é, em que armazenem as mesmas chaves. Esta tarefa encontra-se no DataFlasks ao encargo do componente "Anti-Entropy". Este componente encarrega-se ainda da recuperação das novas réplicas que são incorporadas no sistema.

O modo como a recuperação de dados se processa no DataFlasks tem por base uma atuação proativa deste componente, que periodicamente anuncia aos elementos do seu grupo de replicação, isto é, aos nodos que se incluem na sua vista de grupo, as chaves que se encontram armazenadas localmente. Cada nodo, por sua vez, verifica se possui exatamente a versão específica de cada uma das chaves recebida. Para cada par chave-versão que um determinado nodo não possua, é emitido um pedido GET desse nodo para o originador da mensagem.

2.5 DISCUSSÃO

Cada vez mais as necessidades de armazenamento de dados são maiores e a seleção do sistema de gestão de dados a utilizar deve ser uma tarefa devidamente ponderada. Diferentes sistemas oferecem diferentes garantias e interfaces, e tal escolha deve ser contextualizada com a finalidade pretendida. O cenário em estudo insere-se num contexto de larga escala, por exemplo um ambiente IoT, pelo que garantias de um sistema escalável devem ser asseguradas. Por esta razão, bases de dados relacionais e todos os restantes sistemas que têm por base arquiteturas essencialmente centralizadas, não se adequam ao propósito considerado devido ao facto de não escalarem. Sistemas que fazem uso de protocolos de coordenação ou primitivas de locking, por forma a oferecer garantias de consistência forte, tais como HDFS ou Ceph, possuem também, no cenário considerado, escala e disponibilidade limitada.

Um passo fundamental para atingir alta escalabilidade e disponibilidade passa por adotar uma arquitetura descentralizada, recorrendo a protocolos *peer-to-peer*, bem como pela relaxação da consistência do sistema a troco de disponibilidade. Dentro dos sistemas que

fazem uso destes protocolos incluem-se bases de dados chave valor, como *Cassandra* ou *Dynamo*, e sistemas de ficheiros, como *CFS* ou *Ivy*. Contudo, ao optarem por organizações estruturadas, nomeadamente tabelas de hash distribuídas, têm dificuldades em lidar com níveis elevados de agitação do sistema, como os observados em redes de larga escala. Um subconjunto de protocolos *peer-to-peer* conhecidos pela sua capacidade regenerativa são os protocolos epidémicos. Estes já foram aplicados com sucesso no *DataFlasks*, uma base de dados chave-valor distribuída, motivo pelo qual o mesmo foi definido como substrato do nosso sistema. Todavia, implementar um sistema de ficheiros utilizável e coerente sobre um substrato de armazenamento como o DataFlasks, que exibe garantias de consistência eventual, sem coordenação, onde não é garantida a consistência de escritas, e que apenas expõe uma interface put/get de blocos, impõe um desafio significativo que esta dissertação pretende abordar. Relembrando a tabela 1 , inserida na secção 2.2, que resume os tipos de sistemas de gestão de dados atualmente existentes na literatura, é possível enquadrar o LSFS na posição de sistemas de ficheiros distribuídos não estruturados. Tal como se pode observar na nova versão da tabela (Tabela 2), o LSFS vem avançar o estado da arte introduzindo um conceito aos sistemas de ficheiros que ainda não foi explorado.

	Sistema de Ficheiros	**Key-Value Store**	**Base de dados relacional**
Centralizado	✓	✓	✓
Distribuído estruturado	✓	✓	✓
Distribuído não estruturado	✗→ **LSFS**	✓	✗

Tabela 2: Enquadramento do LSFS nos sistemas de armazenamento de dados

3

ARQUITETURA DO SISTEMA

O LSFS foi desenvolvido com o intuito de constituir uma solução escalável, tolerante a falhas e dotada de alta disponibilidade. Tal como relatado no Capítulo 2, um passo fundamental para atingir a devida escalabilidade passa pela adoção de uma arquitetura distribuída e, mais do que isso, dotada de uma organização descentralizada. Este requisito foi tido em conta no desenho do LSFS, não se tendo recorrido a qualquer componente central que pudesse constituir um gargalo do sistema.

3.1 VISÃO GERAL DO SISTEMA

Numa visão global, o LSFS tem por base um conjunto de nodos de armazenamento, possivelmente dispersos por uma rede de larga escala, encontrando-se em execução em um qualquer dispositivo com poder de processamento e armazenamento variável. Esta rede de nodos que constitui o substrato do sistema trata-se de uma rede *peer-to-peer*, onde cada nodo aplica o mesmo conjunto de algoritmos de forma autónoma, isto é, o seu progresso faz-se com base nas decisões que toma localmente, não pactuando com nenhum outro nodo segundo um qualquer algoritmo de coordenação. As interações entre os vários componentes ocorrem pois uniformente por toda a rede.

Já no que respeita a tolerância a falhas, apenas um dos sistemas de estado da arte, o DataFlasks, se mostrou apto para lidar e recuperar com sucesso de elevados níveis de instabilidade do sistema, característicos de redes de larga escala. O DataFlasks ultrapassa o desafio em questão optando por uma organização de rede não estruturada, tirando vantagem da utilização de um subconjunto de algoritmos *peer-to-peer*, que são os algoritmos epidémicos. No desenho do nosso sistema decidimos contemplar a robustez dos algoritmos utilizados no DataFlasks, tendo-se definido o DataFlasks como substrato do mesmo. Naturalmente, o DataFlasks tendo sido concebido para ser utilizado como uma base de dados chave-valor não possui quer a sua API quer os seus componentes devidamente adequados e otimizados para poderem ser utilizados por parte de um sistema de ficheiros. Posto isto, para além de uma reimplementação total do mesmo, determinadas modificações tiveram de ser realizadas para que pudesse ser utilizável para o nosso caso de uso.

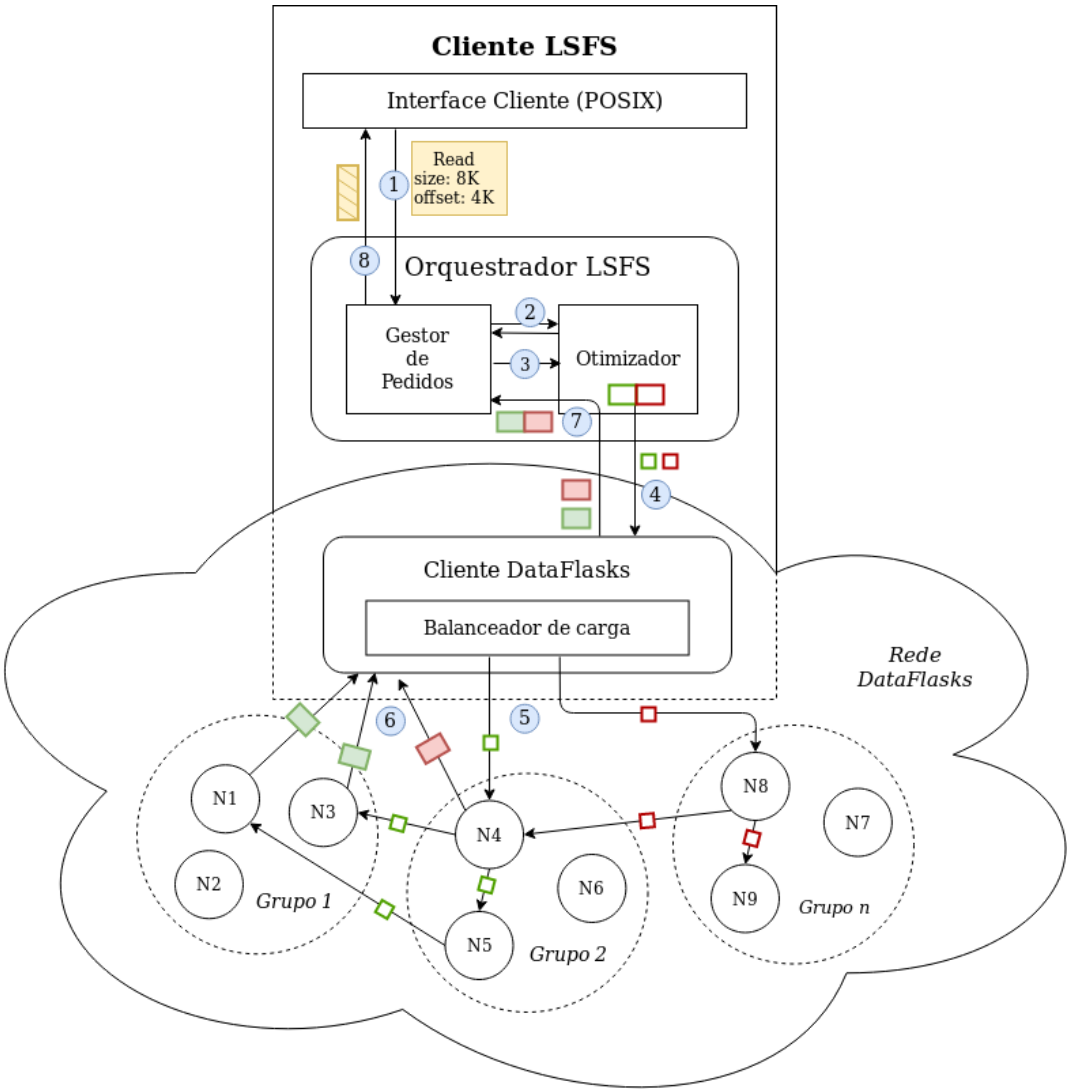


Figura 2: Arquitetura geral do sistema

Tendo em conta que o substrato do nosso sistema é o DataFlasks, a rede *peer-to-peer* que anteriormente se falava é a rede do DataFlasks, bem como os nodos de armazenamento são, por sua vez, nodos DataFlasks. Na Figura 2 tais nodos são identificados pelas abreviaturas Nx. Ainda na figura, é possível observar que os nodos são no DataFlasks agrupados em grupos de tamanho idêntico, constituindo grupos de replicação de dados. Os nodos inseridos num mesmo grupo encontram-se, pois, encarregues de armazenar o mesmo conjunto de chaves, sendo que cada chave é mapeada num único grupo.

De forma a tirar partido do espaço de armazenamento do DataFlasks e utilizá-lo sob a forma de um sistema de ficheiros, recorre-se ao cliente do sistema de ficheiros, Cliente LSFS. Este, incorpora-se na rede do substrato e estabelece comunicação com os demais nodos de

armazenamento. Relativamente à sua constituição, encontra-se dividido em 3 componentes fundamentais: "Interface Cliente", "Orquestrador LSFS" e "Cliente DataFlasks".

O primeiro componente diz respeito à interface disponibilizada. Um dos principais requisitos estabelecidos para o sistema é que exportasse uma interface POSIX, pela facilidade de compatibilidade com a grande maioria das aplicações. Este componente trata de exportar a interface especificada traduzindo cada pedido em invocações à lógica do sistema de ficheiros. O componente que encapsula toda esta lógica necessária à satisfação dos pedidos é o *Orquestrador LSFS*. Ao encargo deste último, encontram-se todas as questões que respeitam a gestão de dados e metadados, desde a divisão de ficheiros em blocos, mapeamento de blocos a chaves, resolução de *pathnames*, ao contacto com o substrato para acesso e persistência de informação. Os pedidos são satisfeitos através de uma ação conjunta entre os dois componentes em que o mesmo se divide, sendo estes o *Gestor de pedidos* e o *Otimizador*. O *Gestor de pedidos* constitui o ponto de entrada do *Orquestrador LSFS*, onde em função do tipo de pedido, seja de escrita, leitura, relacionado com metadados, sabe o procedimento que deverá levar a cabo para retornar uma resposta à camada superior. Tal implica, normalmente, contactar com o *Otimizador*. Este último, mantém algum estado acerca do sistema de ficheiros, nomeadamente efetua caching de metadados de ficheiros, bem como implementa funções que permitem a paralelização de determinadas operações, otimizando o desempenho do sistema de ficheiros.

Por último, como componente base do cliente do sistema de ficheiros encontra-se o *Cliente DataFlasks*. Este componente é vital para o funcionamento de todo o sistema já que age como mediador do acesso ao substrato e, por sua vez, aos dados. A interface oferecida pelo mesmo corresponde a uma extensão da API Put-Get originalmente concebida para o DataFlasks, abordada em mais detalhe na Secção 3.5. A principal função deste componente é pois a de, encontrando-se incorporado na rede de nodos, disseminar os pedidos provenientes do *Orquestrador LSFS* na rede, para que cheguem aos nodos de armazenamento alvo, coletar as respostas provenientes dos mesmos e, por sua vez, retornar a resposta devida. Para a disseminação dos pedidos este componente recorre a um subcomponente que é o *Balanceador de carga* que constitui uma das principais fontes de melhoria de desempenho por parte do sistema de ficheiros, isto é, quanto mais acertadas forem as decisões para que nodos enviar os pedidos, mais rápida é a obtenção das resposta, bem como menor a carga introduzida na rede. Como veremos mais à frente este componente será alvo de um estudo mais detalhado.

3.2 Fluxo de pedidos no LSFS

Tendo-se apresentado cada um dos componentes que formam a arquitetura do sistema será agora elucidado, com um exemplo prático de uma leitura realizada sobre o mesmo, a forma como decorre o fluxo de pedidos pela arquitetura. Na Figura 2 encontram-se

assinaladas as diferentes etapas envolvidas no processamento do pedido em questão. Ora, o primeiro passo consiste, mediante receção de tal pedido na interface do cliente, em fazê-lo chegar ao *Orquestrador LSFS*, nomeadamente, ao *Gestor de pedidos*. O pedido referido trata-se pois de uma operação de leitura (*read*), de tamanho (*size*) 8 KiB e *offset* 4 KiB. Considere-se, ainda, que o DataFlasks armazena blocos de 4 KiB de tamanho.

Com base nesta informação, a primeira interação que o *Gestor de pedidos* realiza é com o *Otimizador* (Etapa 2) com o intuito de conhecer o tamanho efetivo do ficheiro e verificar se a leitura se encontra em conformidade com tal tamanho. Uma vez que uma operação de leitura é sempre seguida de uma operação de *open* do ficheiro, onde os seus metadados foram requisitados ao substrato, o *Otimizador* possui esta informação armazenada em cache pelo que responde imediatamente. Caso seja possível satisfazer o tamanho requisitado para a leitura, a partir de tal tamanho e do *offset* especificado, a lógica do *Gestor de pedidos* permite-o conhecer exatamente que chaves mapeiam nos blocos pretendidos (para o caso em questão, considerando que cada bloco tem 4 KiB de tamanho, tratam-se de dois blocos). Desta forma, o *Gestor de pedidos* instrui o *Otimizador* para paralelizar o processo de obtenção dos blocos (Etapa 3).

De seguida, o *Otimizador* constrói duas mensagens e comunica com o *Cliente DataFlasks* para proceder ao seu envio (Etapa 4), ficando a aguardar pela receção das respostas. Já o *Cliente DataFlasks*, com recurso ao *Balanceador de carga*, propaga os pedidos na rede (Etapa 5). Por sucessivas disseminações, os pedidos atingem o grupo de replicação responsável por armazenar cada uma das chaves. A etapa seguinte (Etapa 6) consiste no envio das respostas com os referidos blocos para o cliente. Observe-se que a chave referente ao pedido identificado pela cor verde mapeia no Grupo 1 e o pedido atinge dois nodos do grupo pelo que duas respostas são retornadas. Para o caso do pedido a vermelho, apenas se atinge um nodo do grupo de replicação da chave sendo apenas uma única resposta retornada.

Assim que chega uma das respostas ao *Cliente DataFlasks* este dá o pedido por terminado e retorna o bloco ao Otimizador que, uma vez na posse de ambos os blocos, os agrega e passa ao *Gestor de Pedidos* (Etapa 7). Por fim, na Etapa 8, a devida resposta é retornada ao utilizador do sistema de ficheiros.

Apesar de se ter recorrido a uma leitura para o exemplo considerado, o processamento de uma escrita seguiria uma percurso muito semelhante. Uma tarefa extra que teria de ser efetuada consiste na disseminação da escrita por todos os nodos do grupo do grupo de replicação da chave, de forma a garantir que todos armazenam os mesmos blocos.

3.3 Organização do sistema de ficheiros

Tal como a maioria dos sistemas de ficheiros distribuídos contemporâneos, o LSFS distingue operações de dados de operações sobre metadados. Cada ficheiro é constituído por

vários blocos de dados onde se armazenam o conteúdo do ficheiro propriamente dito, e é ainda constituído por um bloco especial, bloco *inode*, que compreende os metadados do ficheiro. As estruturas de dados utilizadas pelo LSFS são semelhantes às utilizadas por um sistema de ficheiros Unix comum, contendo os metadados encontrados num bloco *inode* a mesma informação que se pode encontrar num *inode* tradicional (tamanho, permissões, etc). Por flexibilidade de escrita passaremos a referir os blocos de dados por *dblocks*, e os blocos *inode* por *iblocks*.

Uma diretoria é neste formato considerada também um *iblock*, contudo, trata-se de um *iblock* especial. Para além dos metadados inerentes à diretoria, é ainda formado por um conjunto de referências para os seus *iblocks* filho, podendo os últimos constituir, por sua vez, ficheiros ou outras diretorias. Este formato estabelece uma organização hierárquica da àrvore de diretorias tal como a encontrada nos sistemas de arquivos Unix.

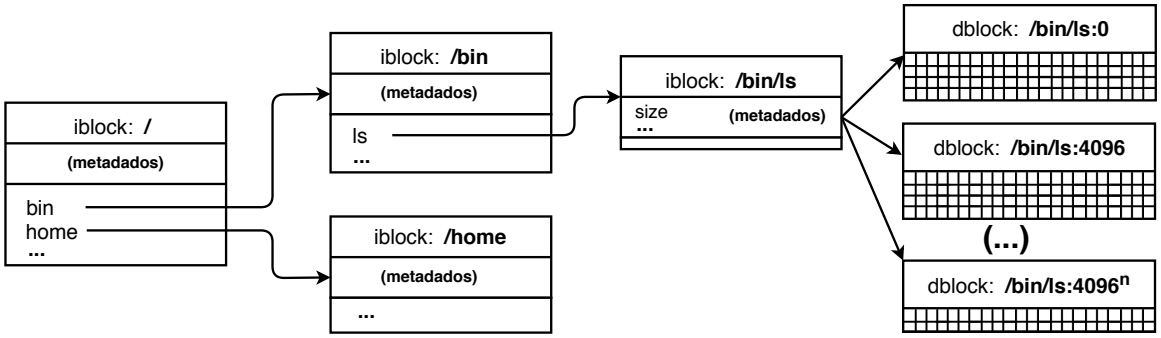


Figura 3: Arquitetura de blocos

Por motivos de escalabilidade e no sentido de evitar pontos únicos de falha, o LSFS não diferencia os dois tipos de blocos a nível de armazenamento. Quer os dados como os metadados são armazenados no *DataFlasks*, que constitui o único substrato de armazenamento. Outros sistemas de ficheiros distribuídos de estado da arte, tais como *CFS*, *Pastis* e *Ivy*, seguem o mesmo princípio, utilizando um único sistema de armazenamento, baseado em blocos, para a persistência. Estes, exibem uma organização em blocos semelhante à descrita, diferenciando-se, contudo, na definição de mutabilidade/imutabilidade dos blocos bem como no mapeamento dos blocos a chaves. No *CFS*, por exemplo, todos os blocos à exceção do bloco raiz são imutáveis, sendo referenciados pelo valor de *hash* do seu conteúdo. O nodo raiz, por outro lado, é mapeado através da chave pública do "dono" do sistema de ficheiros. A aplicação de *hashing* ao conteúdo permite validar a integridade dos dados, contudo, uma vez que os *inodes* são também endereçados através do seu valor de *hash*, as atualizações realizadas aos ficheiros resultam na recomputação dos diferentes blocos afetados até à raiz. Já, o *Pastis*, com o intuito de evitar o efeito cascata provocado pela atualização de ficheiros, considera todos os blocos *inode* mutáveis, cria um par de chaves pública-privada por cada

inode, e referencia-o através da chave pública criada. A utilização destes pares de chaves servia ainda propósitos de autenticação e segurança.

No LSFS pretendíamos, essencialmente, verificar qual era o desempenho e resiliência que conseguíamos obter ao construir um sistema de ficheiros sobre uma rede não estruturada, pelo que a segurança e integridade dos dados constituíam tópicos fora do domínio de estudo. Assume-se, pois, que não existe corrupção de dados, nem utilizadores maliciosos. Definiu-se, portanto, uma organização onde cada *iblock* é referenciado através do caminho completo do respetivo *inode*, e cada *dblock* pela agregação do caminho completo do ficheiro a que se refere e o offset correspondente ao bloco.

Esta organização, esboçada na Figura 3, permitiu evitar que o acesso aos ficheiros fosse realizado percorrendo recursivamente as sub-árvores onde o ficheiro se inclui, como acontecia nos sistemas mencionados, mas, por acesso direto ao *iblock* pretendido. Outra decisão aplicada ao LSFS passou pela definição de um tamanho de bloco de dados fixo. Desta forma, evitou-se armazenar no *iblock* específico de um ficheiro, a listagem das chaves referentes a todos os blocos que o formam, sendo apenas necessário conhecer o tamanho do ficheiro de forma a calcular os diferentes *offsets* e chaves associadas a cada bloco.

No LSFS, o tamanho do bloco constitui um valor parametrizável. Tal tamanho deverá, contudo, ser inferior a 64KiB, dado representar o tamanho máximo total de um pacote IP. Atente-se ainda que o tamanho de dados máximo transmíssivel na rede encontra-se limitado pelo *mtu* das ligações utilizadas que é, normalmente, bastante mais reduzido que o valor considerado, implicando fragmentação. Dado que o LSFS utiliza, essencialmente, udp para a comunicação entre os nodos a perda de um dos pacotes resultantes do processo de fragmentação resulta na perda de todo o pacote. Desta forma, com o intuito reduzir a probabilidade de ocorrência de tais situações, definiu-se 4KiB como valor por defeito para o tamanho de cada bloco de dados no LSFS, valor este que se utilizará também no protótipo desenvolvido.

Tendo em conta o valor descrito, também os *offsets* utilizados serão sempre múltiplos de 4096 Bytes, tais como os que se observam na figura. Considere-se, portanto, o processamento de um pedido de leitura de tamanho $16KiB$ e *offset* 4096 sobre o ficheiro \bin\ls. Dado que o *offset* é um múltiplo de $4KiB$ é apenas necessária a requisição de 4 *dblocks* ao substrato ($16KiB/4KiB = 4$) para satisfazer o pedido. Tendo em conta o formato utilizado para a tradução de blocos em chaves e o *offset* especificado, as chaves requeridas seriam as seguintes: \bin\ls:4096, \bin\ls:8192, \bin\ls:12288 e \bin\ls:16384.

Todos os blocos são também no LSFS imutáveis, pelo que cada chave inserida no substrato é versionada e, todas as atualizações ocorridas aos blocos, são realizadas pela inserção de um novo bloco com uma versão superior.

3.4 MODELOS DE COERÊNCIA

A escolha do DataFlasks como substrato do nosso sistema prende-se com o facto de nenhum outro sistema dos representados no estado da arte se encontrar apto para atingir a escala e resiliência que procuramos neste trabalho. O DataFlasks foi desenhado especificamente para redes de larga escala, e é baseado nas seguintes assunções:

"O sistema é assíncrono, pelo que nenhuma assunção pode ser feita relativamente ao tempo que um determinado nodo demora a executar o seu algoritmo, ou o tempo que dada mensagem demora a chegar ao seu destinatário. ... Assume-se que os nodos podem entrar e sair do sistema por iniciativa própria. Os nodos podem falhar e recuperar, mas não se desviam dos seus algoritmos. Os nodos que recuperam podem perder o seu estado, mas não pode ocorrer corrupção de dados. As mensagens podem se perder ou podem ser transmitidas em duplicado, havendo, contudo, uma probabilidade maior que 0 *da mensagem chegar ao destinatário."*[Maia et al., 2014]

Ao incorporarmos o DataFlasks no nosso sistema herdámos as assunções realizadas. É de notar ainda que o DataFlasks não possui qualquer mecanismo de controlo de concorrência. Pelo contrário, introduziu um sistema de versões, delegando ao cliente a obrigação de versionar cada operação. Desta forma, cada chave encontra-se sempre associada a uma determinada versão, par (chave, versão), devendo a versão ser atualizada na ocorrência de uma atualização do objeto. O DataFlasks não garante, pois, consistência nas escritas, pelo que escritas concorrentes sobre o mesmo par (chave, versão) podem facilmente provocar divergências nos estados das réplicas, das quais não consegue recuperar.

Ainda, a não adoção de protocolos de coordenação faz com que as réplicas no DataFlasks se possam encontrar, em determinados momentos, inconsistentes, fruto de atualizações que, por motivos como a perda de mensagens, não chegaram a todas as réplicas de um mesmo grupo. Neste sentido, recorre a mecanismos de anti-entropia de forma a uniformizar o estado das réplicas e, por sua vez, restabelecer a consistência. O DataFlasks trata-se, portanto, de um sistema de natureza inevitavelmente consistente. Tendo-se definido o DataFlasks como sistema de armazenamento do LSFS, também este último conseguirá oferecer no máximo consistência eventual.

3.4.1 *Requisitos de coerência no LSFS*

A motivação para o desenvolvimento do LSFS enquadrou-se desde início num contexto de larga escala, por exemplo IoT, em que existem milhares de dispositivos a produzir informação e a armazena-la num sistema de armazenamento que providencie o seu acesso futuro, para processamento e transformação em informação útil. Este tipo de cenário é cada vez mais comum no quotidiano das empresas. Tome-se o exemplo de uma linha de produção com diversos sensores e unidades com processamento e armazenamento limitado

que monitorizam fatores como temperatura, humidade, etc. uma função habitual destes dispositivos é a escrita das medições realizadas num conjunto de logs independentes. Estes logs são, posteriormente, combinados para efeitos de análise ou até para alimentarem um determinado algoritmo de aprendizagem máquina. Este exemplo pode ser generalizado a diferentes outros casos de uso semelhantes, que são bastante frequentes em contextos de aplicações de larga escala.

Ora, é exatamente este tipo de casos de uso a que o LSFS se destina, procurando armazenar os dados próximos da sua fonte de produção, como alternativa a um armazenamento em nuvem, e oferecendo uma interface POSIX para acesso e persistência dos mesmos. Posto isto, num sistema de ficheiros, à semelhança de um qualquer sistema de gestão de dados, são duas as entidades que realizam acessos ao mesmo, os escritores e os leitores. O padrão de acesso aos dados considerado para este caso de estudo e nesta dissertação é caracterizado pela atuação de múltiplos escritores e múltiplos leitores num formato *Write Once Read Many*.

O LSFS foi desenhado exatamente para atender a este padrão tendo sido definidos os seguintes requisitos: Vários escritores podem utilizar o sistema em simultâneo, contudo, operam sobre ficheiros independentes; O padrão de escritas por parte de um escritor consiste em escritas sequenciais de ficheiros completos; Leituras podem se intercalar com escritas, mas cada leitor deve aceder sempre a um vista coerente do sistema de ficheiros; As escritas realizadas com sucesso encontrar-se-ão inevitavelmente visíveis aos diferentes leitores.

3.4.2 *Modelo de coerência no LSFS*

De acordo com os requisitos definidos para o sistema fica claro que o LSFS tem de oferecer no mínimo consistência eventual. Atente-se que, apesar do DataFlasks oferecer este nível de consistência, o faz para blocos individuais. Um dos objetivos impostos consiste, pois, em providenciar o mesmo nível de consistência para conjuntos de blocos logicamente relacionados como os que podemos encontrar na organização do LSFS.

De seguida, vamos mostrar como é que o LSFS consegue satisfazer os requisitos de coerência impostos e, desta forma, oferecer garantias de consistência eventual. Neste sentido, vamos começar por explicar como é que era possível lidar com a presença de um único escritor no sistema. Posteriormente, passaremos para um contexto de múltiplos escritores (M escritores) onde será descrito o modelo adotado pelo LSFS. Atente-se que tal modelo depende diretamente da organização de blocos definida para o sistema de ficheiros.

Antes de passar à explicação resta caracterizar o padrão de acessos ao sistema de ficheiros por parte dos escritores e leitores. Para o caso de um escritor, este abre um ficheiro novo, escreve sequencialmente vários blocos de dados e fecha o ficheiro. Já, no caso de um leitor o padrão de acessos consiste em abrir o ficheiro, realizar um conjunto de leituras, aleatórias ou sequenciais, sobre o mesmo e fecha-lo. Admita-se ainda que nenhuma restrição é realizada

relativamente às diretorias sobre as quais atuam cada um dos escritores, num contexto de múltiplos escritores, pelo que a possibilidade de ocorrência de atualizações simultâneas sobre as mesmas é um fator a considerar na análise.

3.4.2.1 1 *Escritor*

Considere-se, portanto, numa primeira instância o caso mais simples onde existe apenas 1 escritor e 1 leitor. Dado que não é admissível a modificação de ficheiros tem-se que o escritor pode-se limitar a escrever os diferentes blocos de dados do ficheiro associados a uma qualquer versão, por exemplo um *timestamp*, que permanece única para cada chave. Perante o fecho do ficheiro, são escritos os metadados do mesmo e, por último, atualizado o *iblock* referente à diretoria onde o mesmo se inclui, sendo lhe atribuída uma nova versão. Relativamente às versões utilizadas para as diretorias correspondem a identificadores numéricos incrementados para cada atualização. Uma vez que não existem escritas concorrentes não ocorrem conflitos ao nível das mesmas.

As réplicas no DataFlasks convergem no sentido de incorporar todas as versões para cada chave de que são responsáveis, pelo que, para as leituras bastaria retornar a última versão encontrada para a chave, o que no caso dos *dblocks* seria única. Para o caso dos *iblocks* de diretoria, pode acontecer que o bloco retornado não corresponda à última atualização da diretoria, dado que a resposta pode ter provindo de uma réplica que ainda não se encontra completamente consistente. Contudo, tal é admissível num cenário de consistência eventual.

3.4.2.2 *M Escritores*

Passando para um cenário onde existem múltiplos escritores tem-se que, comparativamente ao cenário de um só escritor, existe a possibilidade de ocorrência de conflitos na atualização simultânea, por parte de dois escritores, de um mesmo *iblock* de diretoria. Tal acontece, quando dois escritores tentam, por exemplo, inserir, cada um, um novo ficheiro numa dada diretoria. Estes, requisitam a última versão do *iblock* respetivo, incluem o ficheiro escrito e realizam a inserção do *iblock* atualizado, incrementando a sua versão. Uma vez que ambos leram a mesma versão também a versão incrementada será a mesma, pelo que serão propagados na rede dois blocos distintos associados ao mesmo par chave-versão, sendo gerada uma inconsistência.

Um modo de contornar este problema seria associar a cada versão o identificador do cliente que a produziu e, por sua vez, aplicar uma política de seleção da versão que prevalece, por exemplo, a versão associada ao maior identificador. Apesar de ser possível, desta forma, atingir consistência eventual, isto é, inevitavelmente a leitura da diretoria retornará sempre o mesmo *iblock*, torna possível, devido à política imposta, a ocorrência de um problema conhecido nas bases de dados por *lost update*[Weikum and Vossen, 2001]. Este problema ocorre quando em duas atualizações simultâneas, uma se perde. Neste caso, ao manter

apenas uma das versões do *iblock*, um dos ficheiros não é inserido na diretoria e perde-se para sempre, havendo, por sua vez, perda de coerência do sistema de ficheiros. O *Pastis*, por exemplo, resolve este problema, procedendo ao *rollback* de uma das atualizações, pelo que tem de esperar pela resposta de todas as réplicas. Uma estratégia como esta não é, contudo, aplicável ao DataFlasks dado ter sido desenhado para redes de larga escala, onde cada nodo possui um conhecimento reduzido da mesma.

A estratégia adotada no LSFS de forma a garantir a coerência do sistema de ficheiros consistiu em igualmente associar a cada versão o identificador do cliente que a produziu , mas, em vez de descartar uma das atualizações, introduziu o conceito de *Conflict-free Replicated Data Types (CRDTs)*[Shapiro et al., 2011] aos metadados de diretoria. As diretorias foram configuradas para armazenar um conjunto sempre crescente de referências para outros *inodes*, não sendo permitidas remoções dentro das mesmas. A aplicação desta estratégia foi realizada pela definição de uma função de junção que permite agregar diferentes versões dos metadados de uma mesma diretoria sem perder dados. Um sistema de ficheiros com as características descritas, apesar de distinto de um sistema convencional, não suportando remoções, permite satisfazer todos os requisitos necessários para o nosso caso de uso.

3.5 Extensão da API do DataFlasks

A interface put/get disponibilizada pelo DataFlasks encontrava-se inadequada para suportar o modelo de coerência adotado. De forma a contemplar tal modelo algumas modificações tiveram de ser realizadas à sua API.

A primeira modificação advém da necessidade de realização de junção de metadados de diretoria. Dado que é no DataFlasks que se encontram armazenadas cada uma das versões para uma determinada chave é também nos nodos de armazenamento deste o local propício à execução das junções. No entanto, cada nodo DataFlasks trata os dados de forma opaca, como conjuntos de bytes, pelo que era necessário arranjar uma forma de garantir que a realização de um put de um iblock de diretoria, desencadeasse localmente a operação de junção. Para tal, introduziu-se uma nova operação, **put_with_merge**, que veio substituir a operação de *put* na persistência deste tipo de blocos.

Por outro lado, a adoção de versionamento associado a uma gestão descentralizada das versões impôs a necessidade de uma operação de consulta da última versão conhecida para uma determinada chave, **get_latest_version**. A própria operação de **get** foi também adaptada para aceitar a possibilidade de não se especificar a versão pretendida. Neste caso, procurou-se acelerar o processo de obtenção da última versão para uma determinada chave, que doutra forma teria ser realizado recorrendo à realização de dois pedidos: um, *get_latest_version*, para obter a última versão da chave pretendida; outro, *get*, para efetivamente requerer o bloco de dados associado.

Para além destas, duas outras operações, **put_batch** e **get_batch**, foram também incorporadas na interface servindo unicamente propósitos de otimização e desempenho. Resumidamente, estas procuram tirar proveito de pedidos de escrita e leitura de maior tamanho recorrendo a estratégias de paralelização dos mesmos. Estas operações serão alvo de um estudo mais detalhado na Secção 4.4. De seguida, é possível encontrar a especificação completa da API atualizada do DataFlasks.

```
void put(const string& key, long version, const char* blk, size_t size);
string get(const string& key, long* version = nullptr);
void put_with_merge(const string& key, long version, const char* data, size_t
    size);
long get_latest_version(const string& key);
void put_batch(vector<string>& keys, vector<long>& versions, vector<const char*>&
     blks, const vector<size_t>& sizes)
void get_batch(const vector<std::string>& keys, vector<string>& blk_strs);
```

4

IMPLEMENTAÇÃO

Uma vez apresentadas as principais decisões arquiteturais e organizacionais que caracterizam globalmente o funcionamento do LSFS, serve o presente capítulo o propósito de relatar a forma como o mesmo foi concretizado na prática e a sua, respetiva, evolução no protótipo disponibilizado. São aqui explicados todos os detalhes de implementação e otimizações realizadas quer diretamente sobre o LSFS, quer recorrendo a modificações da lógica original presente nos diferentes componentes do DataFlasks.

4.1 INTERFACE DE SISTEMA DE FICHEIROS

O LSFS foi desenhado para exportar uma interface POSIX. Para a sua implementação optou-se por uma solução realizada totalmente em espaço de utilizador. Para tal, recorreu-se ao Fuse[Szeredi,], versão 3.9.1, como camada de indireção que, de forma transparente, redireciona chamadas ao sistema de ficheiros, por exemplo por parte de uma aplicação, para a nossa implementação da API da biblioteca Fuse contida no "Orquestrador LSFS". Ao contrário do nosso sistema, existem outros sistemas de ficheiros distribuídos que são implementados como uma extensão do *Kernel* do sistema operativo, contudo, no nosso cenário onde a rede será o gargalo e não o acesso a disco ou a mudanças de contextos entre *Kernel* e espaço de utilizador, o Fuse surge como uma alternativa igualmente viável, tal como relatado em [Tarasov et al., 2015]. Para além disso, ao utilizar o Fuse foi possível tirar proveito de vantagens como depuração de código mais fácil, portabilidade, resiliência, entre outras.

4.2 JUNÇÃO DE DIRETORIAS

A definição do LSFS como um sistema de múltiplos escritores tornou possível a ocorrência de atualizações simultâneas ocorridas sobre a mesma diretoria. O DataFlasks rejeitando a utilização de protocolos de coordenação, já que tal constituiria uma entrave à escalabilidade, passou à aplicação o encargo de garantir a consistência das escritas. No LSFS víamos essa

escalabilidade como um requisito que não tencionávamos abdicar, pelo que, uma vez mais, a utilização de protocolos de coordenação não era admissível. A resolução de conflitos tinha pois de ocorrer de forma descentralizada e individualmente em cada um dos nodos de armazenamento.

Um passo fundamental para que a resolução de conflitos pudesse ser realizada em cada nodo consistia em garantir que todas as versões de uma mesma chave mapeavam nas mesmas réplicas. Este problema encontrava-se trivialmente resolvido pela dependência exclusiva da chave no mapeamento de blocos a grupos de réplicas. A cada nodo chegam, pois, todas as versões alguma vez criadas para uma determinada chave, quer diretamente por disseminação a partir do cliente, quer por recuperação através de mecanismos de anti-entropia. Outro fator imprescindível à resolução de conflitos parte da capacidade de diferenciar versões criadas por diferentes clientes. A associação do identificador do cliente a uma determinada versão permitiu ultrapassar situações de ocorrência de escritas de diferentes clientes sobre o mesmo par (chave, versão). A probabilidade de situações como estas ocorrerem num sistema distribuído sem coordenação é muito elevada. A Figura 4 apresenta a ocorrência de um desses casos sobre uma única réplica.

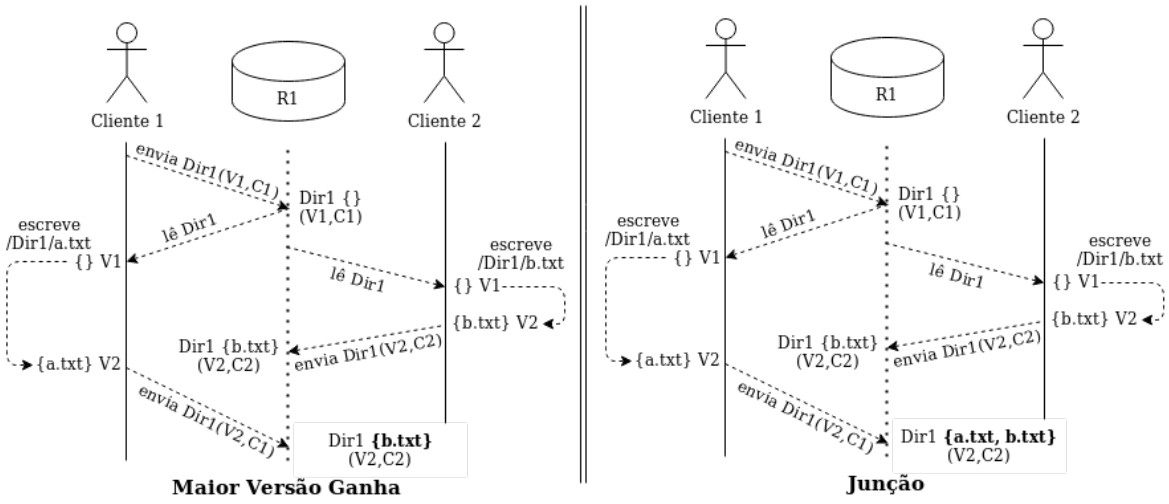


Figura 4: Políticas de resolução de conflitos

Nesta, dois clientes, C1 e C2, adicionam, concorrentemente, os ficheiros a.txt e b.txt à diretoria Dir1. Como ambos leram a mesma versão da diretoria, V1, o incremento da versão ocorrido em cada cliente associa a mesma versão V2, às duas atualizações. Desta forma, à réplica R1 chegariam duas versões V2 distintas. O componente *Request Handler* do DataFlasks, consultando o seu log de escrita, certificar-se-ia que apenas a primeira atualização a chegar era processada. Estas atualizações poderiam, contudo, chegar por ordem distintas a diferentes réplicas, introduzindo inconsistências. Ao associar a cada versão o identificador do cliente que a criou, duas versões distintas (V2,C1) e (V2,C2) atingem as diferentes réplicas, pelo que ambas são processadas.

A presença de diferentes versões para um determinada chave, exige que uma prevaleça e que seja a mesma para todas as réplicas, de forma a garantir consistência. Tal como mencionado na Secção 3.4 a utilização de uma política de resolução de conflitos, como a seleção da maior versão, apesar de permitir atingir consistência, pode tornar o sistema de ficheiros incoerente. Considere-se que as versões são ordenadas segundo o maior identificador de versão, e o desempate realizado pelo identificador de cliente maior. Observando a Figura 4 facilmente se verifica que este tipo de políticas permite a perda de dados. A seleção da versão (V2,C2) em detrimento da versão (V2,C1) originou a perda do ficheiro a.txt.

A solução adotada para o LSFS, por outro lado, estabelece como versão prevalecente o resultado da aplicação de um processo de junção sobre todas as versões conhecidas. Desta forma, procura preservar quer a coerência do sistema de ficheiros quer a sua consistência eventual. O algoritmo de junção utilizado é baseado na união de conjuntos ($\cup$), cada conjunto consistindo nos inodes filho presentes em cada versão do iblock da diretoria em questão. De forma a evitar que este algoritmo tenha de ser executado sempre que um *get* é realizado sobre um iblock de diretoria, esta lógica foi migrada para o momento da escrita do bloco. Posto isto, aquando da receção de um pedido *put_with_merge*, o processo de junção é imediatamente executado, resultando na criação de um novo *iblock*. Este *iblock* é armazenado sobre a maior versão presente localmente. Desta forma, para satisfazer um pedido de *get* basta, independentemente do tipo de bloco requerido, retornar o valor associado à maior versão existente para a chave especificada. Na Figura 4 observa-se que, contrariamente à solução da esquerda, a réplica R1 converge para um estado em que mantém ambos os ficheiros inseridos na diretoria Dir1. Verifica-se ainda que, apesar da versão (V2,C1) ter atingido a réplica depois da versão (V2,C2), é a última que é utilizada para identificar o *iblock* alterado.

A persistência local do *iblock* resultante da operação de junção possibilitou ainda que outra otimização fosse realizada. Tirando proveito das propriedades associativas e comutativas características da operação de união, é suficiente que a junção seja realizado entre o *iblock* referente à maior versão corrente (resultante de sucessivas operações de junção), e o da nova versão recebida. O Algoritmo 1 descreve este processo de incorporação de um determinado *iblock* de diretoria na base de dados de uma das réplicas responsáveis pelo seu armazenamento.

Como variáveis de entrada do algoritmo encontra-se a chave, o par identificador da versão, (*versão*,*id_cliente*), e o bloco de dados associado, *bloco_recb*. O primeiro passo no armazenamento de um *iblock* deste tipo corresponde à procura na base de dados do que é considerada a última versão da chave especificada. Esta tarefa é abstraída pela função *retorna_último_bloco* que, na existência de tal objeto, devolve quer o bloco de dados, *bloco_atual*, quer o par formado pela sua versão e identificador do cliente, (*versão_atual*,*id_cliente_atual*).

Algoritmo 1: Pseudo-código de armazenamento de um *iblock* de diretoria

Entrada: *chave*, (*versão*, *id_cliente*), *bloco_recb*

```
1  bloco_atual, (versão_atual, id_cliente_atual) ← retorna_último_bloco(chave)
2  Se bloco_atual é diferente de vazio então
3      bloco_agregado ← junção(bloco_atual,bloco_recb)
4      Se (versão, id_cliente) < (versão_atual, id_cliente_atual) então
5          armazenado ← inserir(chave, (versão_atual,id_cliente_atual),bloco_agregado)
6          Se armazenado então
7              inserir(chave, (versão, id_cliente), bloco_recb)
8      caso contrário
9          inserir(chave, (versão, id_cliente), bloco_agregado)
10 caso contrário
11     inserir(chave, (versão, id_cliente), bloco_recb)
```

Caso não exista nenhuma versão armazenada para a mesma chave, não é necessária a execução da operação de *junção*, sendo que a versão recebida é armazenada inalterada na base de dados (operação *inserir*) (linha 11). Caso tal versão exista, é invocada a operação de *junção* entre o *bloco_atual* e o bloco recebido (linha 3). Duas situações podem, então, acontecer: a versão referente ao bloco recebido ser ou não lógicamente inferior à maior versão existente. Para o caso da versão recebida ser inferior, tem-se que o bloco resultante da operação de *junção*, *bloco_agregado*, é armazenado na base de dados associado à mesma versão que anteriormente prevalecia na base de dados, isto é, associado ao par (*versão_atual*,*id_cliente_atual*) (linha 5). É de notar ainda que caso tal escrita seja realizada com sucesso, é criada uma entrada na base de dados para a versão recebida (linha 6 e 7). A justificação para tal relaciona-se com motivos de anti-entropia. É necessário registar todas as versões processadas de cada chave para evitar que o nodo inicie protocolos para a recuperação dessas chaves. A ordem destas duas escritas não poderia, contudo, ser alterada já que se existisse um erro na escrita da versão resultante da junção tendo-se já registado a versão recebida, tal versão não voltaria a ser processada, resultando na perda das suas alterações. Por outro lado, quando a versão recebida supera a versão atual, uma única escrita é necessário que seja efetuada, associando a versão recebida, atualmente a maior, ao *bloco_agregado* (linha 9).

4.2.1 *Tolerância a partições temporárias da rede*

A motivação para a adoção de uma estratégia baseada na junção de diretorias tinha por base a constituição de um sistema de ficheiros que, de forma coerente, não introduzisse a perda de ficheiros, garantindo ainda a convergência inevitável das réplicas para um estado

comum. A Figura 5 permite mostrar que tais premissas se verificam mesmo perante a ocorrência de particionamentos temporários da rede.

No exemplo apresentado, o cliente 1, C1, possui em todo momento connectividade com a réplica R1, e o cliente 2, C2, com a réplica R2. Aquando do particionamento da rede, a réplica R1 e R2 não se conseguem contactar. Posto isto, tem-se que o exemplo pode ser dividido em três momentos distintos.

No primeiro momento existe conetividade total entre clientes e réplicas. O cliente 1 começa por criar a diretoria Dir1. A atualização referida atinge inicialmente a réplica R1 que armazena o respetivo *iblock* associado à versão (V1,C1), isto é, versão 1 proveniente do cliente 1. Esta atualização, por disseminação, chega ainda à réplica R2, que igualmente armazena o bloco com a mesma versão. As réplicas encontram-se, neste estado, consistentes.

No segundo momento, a ocorrência de um particionamento da rede faz com que as atualizações do cliente 1 apenas cheguem à réplica R1, e as do cliente 2 à réplica R2. Durante este período o cliente 1 insere dois ficheiros na diretoria, sendo a versão da mesma elevada na réplica R1 para (V3,C1). Já o cliente 2 insere um outro ficheiro, b.txt, incrementando o valor da versão na réplica R2 para (V2, C2). Neste ponto, as réplicas divergem. A réplica R1 possui uma versão do *iblock* que é superior ao da réplica R2 ((V3,C1) > (V2,C2)). Os ficheiros incluídos em cada *iblock* formam conjuntos disjuntos. O armazenado pela réplica R1 inclui os ficheiros a.txt e c.txt, enquanto que o armazenado pela réplica R2 o ficheiro b.txt.

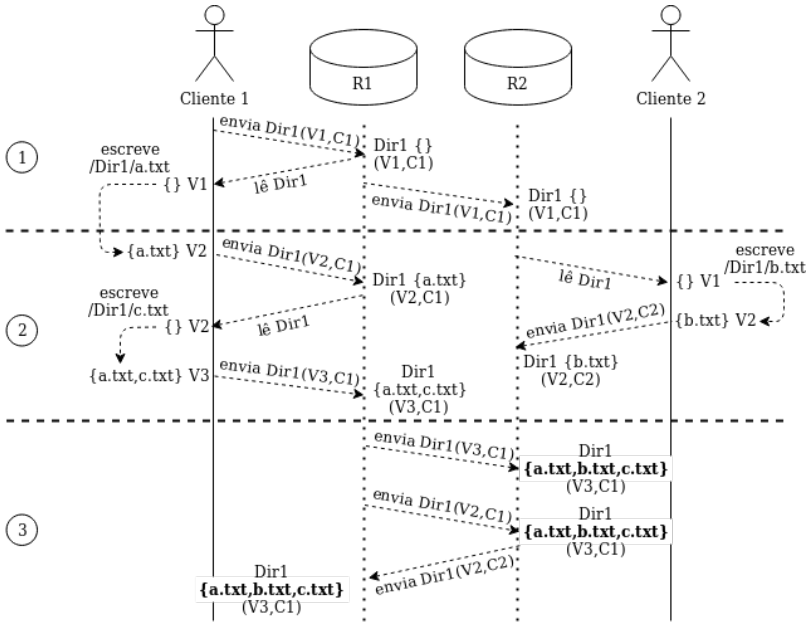


Figura 5: Junção de diretorias - Convergência de réplicas

Por último, no terceiro momento, as réplicas conseguem comunicar novamente, sendo que, por mecanismos de anti-entropia, recuperam as atualizações perdidas. A primeira atualização recuperada pela réplica R2 é a (V3, C1). Em concordância com o Algoritmo 1, a

operação de junção é realizada sobre esta e a última versão armazenada pela réplica, (V2,C2). Da união dos conjuntos de ficheiros resulta um novo *iblock* formado pelos ficheiros a.txt, b.txt e c.txt ($\{a.txt, c.txt\} \cup \{b.txt\} = \{a.txt, b.txt, c.txt\}$). Este *iblock* é associado à maior versão, (V3,C1). Um processo simétrico ocorre na incorporação da atualização (V2,C2) por parte da réplica R1. Ambas as versões convergem, pois, para a mesma versão, onde o *iblock* da diretoria Dir1 inclui a totalidade dos ficheiros escritos. É de notar que a atualização da versão (V2,C1), apesar de atingir por último a réplica R2, em nada altera o seu estado final. Tal se verifica porque a atualização (V3,C1), anteriormente processada, foi resultado de uma junção sobre essa versão. Atente-se ainda que, qualquer que fosse a ordem de recuperação das mensagens, a convergência ocorria para o mesmo estado.

4.3 GESTÃO DE FICHEIROS ABERTOS

Uma das operações mais frequentemente invocadas sobre qualquer sistema de ficheiros consiste na requisição de metadados de ficheiros (representada no FUSE pela operação *getattr*). Desde que um ficheiro é aberto até à libertação do respetivo descritor, esta operação é invocada potenciamente uma grande quantidade de vezes. Cada uma destas invocações traduz-se num pedido *get* realizado ao substrato DataFlasks, penalizando fortemente o desempenho do mesmo.

No LSFS cada ficheiro é criado uma vez, escrito por um único cliente e permanece inalterado durante o resto do seu período de vida. Tal facto, propiciou a realização de *caching* dos seus metadados por parte do cliente que opera sobre o mesmo.

A concretização de *caching* foi realizada pelo componente Otimizador, seguindo a seguinte estratégia. Este componente mantém uma estrutura com os metadados dos ficheiros que se encontram abertos no sistema de ficheiros. A cada um dos metadados é associada uma flag que permite identificar se o ficheiro foi criado (flag *Created*), ou simplesmente acedido (flag *Accessed*). A introdução dos metadados nesta estrutura pode, pois, ocorrer em apenas duas situações: invocação de um pedido de *create* (1) ou *open* (2) à API do Fuse;

Na primeira situação, o ficheiro não existe pelo que é criado e inicializado um novo objeto de metadados referente ao novo ficheiro. Neste objeto é atribuído ao ficheiro um tamanho inicial 0 e povoados os restantes atributos com as permissões e parâmetros especificados no pedido. Estes metadados são, posteriormente, armazenados, na estrutura de ficheiros abertos, associados à flag *Created*. Na situação número 2, uma vez que o ficheiro já existe, os metadados são obtidos a partir do substrato e associados à flag *Accessed*. Uma vez incluídos nesta estrutura, futuras invocações da operação *getattr* são satisfeitas em memória.

Durante o período em que o ficheiro se encontra aberto, o mesmo pode ser submetido a operações de leitura, *read*, ou escrita, *write*. Apenas as operações de escrita modificam, contudo, os metadados do ficheiro, nomeadamente o tamanho e tempos de modificação. A

remoção dos metadados da estrutura é realizada aquando de uma operação de *release* ou *close* sobre o ficheiro. Se os metadados se encontrarem associados à flag *created*, são ainda submetidos para armazenamento no substrato, bem como são alterados os metadados da diretoria em que o ficheiro foi criado de forma a inclui-lo na respetiva diretoria.

4.4 otimização de leituras e paralelismo

A definição do FUSE como a camada de abstração utilizada pelo LSFS para a implementação do sistema de ficheiros, foi acompanhada por inúmeras vantagens. Uma das vantagens proporcionadas pelo FUSE consiste numa implementação interna de *readahead*. Isto é, o FUSE oferece a capacidade de pedir mais informação ao sistema de ficheiros, numa operação de leitura, que a efetivamente requerida pela aplicação com o intuito de acelerar a resposta a pedidos futuros.

Esta otimização é realizada, contudo, de forma inteligente e ponderada. O FUSE estabelece a quantidade de dados que requisita antecipadamente ao sistema de ficheiros com base na sua expectativa do padrão de leituras. Se o padrão de leituras se mostrar aleatório (leitura de blocos aleatórios sobre um ou mais ficheiros), este não consegue antecipar que blocos vão ser lidos a seguir, pelo que requisita apenas o tamanho do bloco de dados requerido. Contrariamente, em leituras sequenciais de ficheiros antevê que os próximos blocos que serão requisitados pela aplicação são os que sucedem imediatamente os blocos anteriormente lidos. Posto isto, em vez de submeter leituras com o tamanho especificado pela aplicação, submete leituras de tamanho, potencialmente, muito superior. O tamanho máximo que pode ser especificado numa leitura encontra-se limitado pelo valor de *max_readahead*, que por defeito é 128KiB.

A capacidade do *Fuse* realizar *readahead*, impulsionou a introdução e configuração de paralelismo no LSFS. Cada ficheiro é no LSFS constituído por blocos de tamanho fixo de 4KiB. Dependendo do tamanho definido para cada leitura/escrita, a mesma será traduzida num diferente número de pedidos ao substrato. Tais pedidos podem ser submetidos paralelamente já que a satisfação de cada pedido é processada de forma independente por diferentes réplicas da rede. A concretização do paralelismo no LSFS fez-se com recurso à extensão da API do cliente DataFlasks com as operações *put_batch* e *get_batch*.

Quanto maior a quantidade de dados requerida em cada leitura/escrita, maior o potencial de paralelização da operação. Contudo, o aumento da paralelização, faz aumentar também o número de mensagens a circular na rede, por sua vez, contribuindo para uma sobrecarga da mesma. Uma sobrecarga excessiva da rede origina a perda de um maior número de mensagens, consequentemente, faz aumentar a ocorrência de *timeouts* e o tempo de resposta de cada pedido. A introdução de paralelismo constitui, pois, um compromisso entre saturação da rede e desempenho.

Com o intuito de limitar o fator de paralelismo quer nas escritas quer nas leituras foram definidos dois parâmetros no LSFS, *limit_write_paralelization_to* e *limit_read_paralelization_to*. Estes parâmetros são utilizados para subdividir leituras e escritas grandes em *batches* de tamanho limitado.

4.5 MODIFICAÇÕES DATAFLASKS

No decorrer da implementação do LSFS praticamente todos os componentes do DataFlasks foram alvo de pequenas modificações, que advieram, essencialmente, da extensão realizada à sua API. O foco principal deste capítulo é dar a conhecer, por outro lado, as modificações cuja principal motivação consiste no aumento do desempenho do sistema de ficheiros, bem como na sua adaptabilidade a um cenário real.

Todas as modificações realizadas ao DataFlasks foram pensadas para não afetar a sua usabilidade por parte de outras aplicações, tendo sido implementadas como uma extensão ou melhoramento do mesmo.

4.5.1 *Storage - Persistência*

Uma das principais funções de um nodo no DataFlasks é garantir o acesso e persistência dos dados que o mesmo é responsável por armazenar, de acordo com o grupo em que se inclui. No DataFlasks a persistência dos dados encontrava-se fora do domínio de estudo, tendo o armazenamento sido implementado em memória. Contrariamente, no LSFS procurámos implementar um protótipo adaptado para uma utilização real, pelo que decidimos oferecer persistência.

O LSFS foi projetado para executar em dispositivos de reduzidas dimensões, com poder de processamento e armazenamento limitado. Tal facto levou-nos a optar por recorrer a uma base de dados embebida, mais especificamente uma base de dados chave-valor embebida, dado que a natureza dos dados a armazenar são também pares chave-valor. A base de dados utilizada foi o LevelDB[LevelDB,] na versão 1.22.0. Criada pela Google, faz o mapeamento entre *strings* chave para *strings* valor. A seleção do LevelDB deveu-se essencialmente a duas razões. Em primeiro lugar, devido ao facto da sua API ter sido implementada nativamente em C++, tal como o nosso sistema, bem como é caracterizada por ser leve e eficiente. Em segundo lugar, pelo facto de disponibilizar uma operação que permite percorrer as chaves de acordo com um determinado prefixo. De forma a perceber o porquê da utilidade de tal operação, é necessário conhecer a configuração definida para as chaves no LevelDB, apresentada na Figura 6.

Tal como sugere a figura, a configuração das chave na base de dados resulta da agregação, com recurso a cardinais, dos diferentes componentes que identificam um bloco no LSFS, a

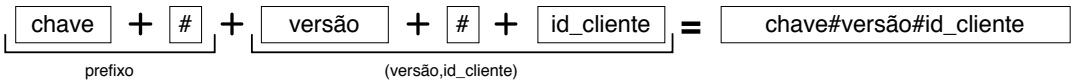


Figura 6: Configuração das chaves no LevelDB

sua chave, versão e identificador do cliente. A partir desta configuração, estabelecendo como prefixo a *string* formada pela chave e o cardinal que a sucede, é possível utilizar a operação referida para percorrer as diferentes versões armazenadas para uma determinada chave. A partir destas, aceder ao bloco correspondente à maior versão é trivial. No LSFS dependemos do acesso a tal versão em muitas operações do sistema de ficheiros, desde a mera satisfação de pedidos *get* ao processo de junção de diretorias. Desta forma, evita-se, pois, a necessidade de percorrer toda a base de dados para a satisfação deste tipo de pedidos, beneficiando o desempenho do sistema.

Diferentes sistemas de armazenamento são, contudo, apropriados para casos de uso distintos pelo que, sendo a nossa implementação do DataFlasks individualmente também uma contribuição desta dissertação, mantivemos uma configuração modular para o componente de "Storage". Uma aplicação que pretenda utilizar a nossa implementação do DataFlasks com um sistema de armazenamento distinto deverá apenas realizar a sua própria implementação da interface definida para este componente, sendo esta semelhante à exposta na secção 3.5 para o substrato de armazenamento.

Ainda de forma a tornar o DataFlasks o mais genérico possível, também a lógica de execução de junções, dado ser específica da aplicação, foi removida deste componente e do DataFlasks. A implementação da função de junção propriamente dita é realizada no contexto do LSFS, e passada ao nodo DataFlasks como uma dependência.

4.5.2 *Load Balancer - Disseminação eficiente*

O Load Balancer é um dos componentes principais do cliente DataFlasks. Este, permite que o cliente se integre na rede de nodos, facultando a disseminação dos pedidos realizados à sua API pelos diferentes nodos de armazenamento. A seleção dos *peers* a utilizar para a disseminação de cada pedido em específico possui um grande impacto no desempenho do sistema. Quanto maior for o número de *hops* necessário para atingir um nodo capaz de processar o pedido, maior o tempo de resposta necessário à sua satisfação, bem como maior a carga na rede induzida pelas sucessivas disseminações do mesmo.

4.5.2.1 *Random Load Balancer*

No DataFlasks foi implementada uma versão básica deste componente, assente num conhecimento reduzido da rede. Denominado de Random Load Balancer, este componente

tem acesso à mesma informação que qualquer outro nodo presente na rede, isto é, a uma vista reduzida e aleatória do sistema obtida por mecanismos de PSS.

Com base nesta vista, o balanceamento de carga é realizado pela seleção aleatória e sucessiva de elementos presentes na mesma para os quais os diferentes pedidos são propagados. Na ocorrência de um *timeout* é selecionado outro nodo para propagar o pedido. A estratégia utilizada nesta versão do componente, assentando numa disseminação baseada na seleção aleatória de nodos da rede encontra-se muito pouco otimizada, sendo potencialmente necessário a ocorrência de muitos *hops* antes do pedido chegar ao nodo que realmente armazena ou irá armazenar os dados e ser satisfeito.

4.5.2.2 *Smart Load Balancer*

Com o intuito de aumentar o desempenho do sistema, foi no LSFS implementada uma nova versão deste componente que intitulámos de Smart Load Balancer. O objetivo passava pela realização de uma aprendizagem contínua acerca do sistema de forma a auxiliar o processo de seleção dos nodos a utilizar na propagação de cada pedido.

Desta forma, para além de uma instância do algoritmo de PSS, este componente passou a executar a sua própria instância do algoritmo de construção de grupos. Tal facto, permite ao cliente ter conhecimento do número de grupos existente em cada momento. Permite-lhe, ainda, classificar cada nodo de acordo com o grupo em que se inclui.

Com base nesta informação e ao longo das sucessivas iterações do algoritmo de PSS, este componente vai colecionando um conjunto limitado de nodos pertencentes a cada grupo. O conhecimento que o cliente possui do sistema deixa, pois, de ser uma vista aleatória de outros nodos da rede, mas essa vista foi substituída por um conjunto de nodos por cada grupo do sistema. Este conhecimento que o cliente mantêm do sistema permite-o, com base no mapeamento de chaves em grupos, estabelecer uma comunicação direta com os nodos responsáveis por satisfazer cada pedido em específico, desta forma, aproximando o DataFlasks a uma *zero-hop key-value store*. Na eventualidade de não se conhecer nenhum nodo do grupo em que determinada chave mapeia, é utilizado um nodo aleatório de outro grupo para a disseminação do pedido, tal como acontece no caso do balanceador de carga aleatório.

A vista do cliente foi implementada como uma lista de listas. Cada uma das listas representando o conhecimento acerca de um grupo do sistema. Uma vez que esta vista é dependente do número de grupos, também esta se deverá conseguir adaptar a situações de dinamismo da rede, nomeadamente, situações onde o número de grupos diminui ou aumenta. De forma a superar este dinamismo, a sua gestão foi colocada ao encargo do algoritmo de construção de grupos que conhece exatamente os momentos de alteração do número de grupos. Recordando a forma como as transições do número de grupos ocorrem (Subsecção 2.4.3), isto é, pela partição de cada grupo em dois ou pela junção de

grupos contíguos dois a dois, foram desenvolvidas duas funções, *split_groups_from_view* e *merge_groups_from_view*, respetivamente, invocadas pelo algoritmo de construção de grupos aquando da ocorrência da transição respetiva.

Para além da capacidade da vista se adaptar perante alterações na dimensão global da rede, tem ainda de conseguir lidar com a entrada e saída esporádica de nodos da mesma, de forma a evitar ao máximo enviar pedidos para nodos que já deixaram o sistema. A estratégia utilizada passou pela manutenção de um fator idade por registo da vista, sendo que tal registo é eliminado assim que atinge uma determinada idade máxima. Ainda no sentido de privilegiar os nodos mais recentemente descobertos no sistema, tais registos são organizados em cada lista de forma ordenada pela idade, sendo que, uma vez que é limitado o número de nodos armazenados por grupo, são esquecidos mais rapidamente os nodos com referências mais antigas. Relativamente, à entrada de nodos no sistema, o algoritmo de PSS garante que estes são automaticamente incorporados.

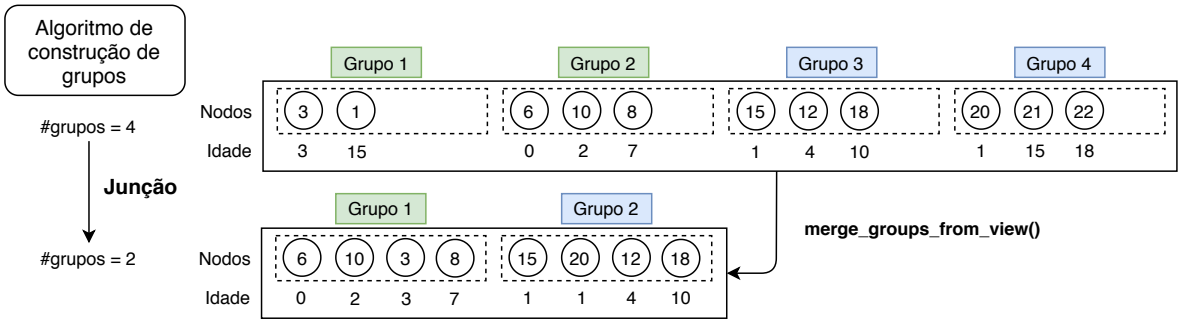


Figura 7: Smart Load Balancer - Operação de junção de grupos da vista

De forma a consolidar algumas ideias do modo como a vista é gerida, apresenta-se na Figura 7, a execução da operação de junção de grupos para um cenário de 4 grupos iniciais e um conhecimento máximo de 4 nodos por grupo. Na figura verifica-se que na vista resultante o grupo 1 é formado exclusivamente por nodos dos primeiros dois grupos da vista inicial e o grupo 2 por nodos dos restantes grupos. Verifica-se, ainda, que o nodo 1, bem como, o 21 e 22, não são incorporados na vista final dado se ter atingido o tamanho máximo de conhecimento por grupo estabelecido e pelo facto de estes registos serem os mais antigos da vista.

4.5.3 *Group Construction - Otimização de transferência de estado inicial*

O algoritmo de construção de grupos utilizado no DataFlasks, converge através do processamento das mensagens de PSS trocadas regularmente pelos nodos no decorrer do seu ciclo de vida. Isto é, aquando da inicialização dos nodos, cada nodo tem conhecimento da existência de um único grupo, o grupo em que se inclui. À medida que se vão trocando

mensagens, este passa a conhecer mais nodos do sistema e a refinar o seu conhecimento acerca do número de grupos existentes de forma a manter um fator de replicação dentro dos limites impostos. Ou seja, considere-se a inicialização de um sistema com 1000 nodos e um intervalo de replicação $[3, 5]$ tem-se que o número de grupos que representa a convergência é 256. Ora, relembrando a condição do número de grupos ser sempre uma potência de 2, a convergência para o estado final num determinado nodo atinge-se passando por cada um dos diferentes estados 1, 2, 4, 8, 16, 32, 64, 128 e 256.

Este caminho, naturalmente, terá de ser percorrido quando se inicializa a rede pela primeira vez, dado que todos os nodos partem do mesmo conhecimento da rede, isto é, nenhum nodo possui de início mais informação que qualquer outro. Este cenário, onde nos encontramos a inicializar toda a rede corresponde, contudo, a uma situação esporádica, sendo que na maioria dos casos a rede já se encontra inicializada e estável. Neste último cenário, os novos nodos que entram no sistema possuem um conhecimento potencialmente muito reduzido da rede comparativamente aos nodos que já se encontram na mesma. Desta forma, com o intuito de possibilitar uma transferência de estado mais rápida por parte destes nodos, evitando passar por todos os estados identificados, foi no LSFS extendido o componente Group Construction com uma fase inicial a que denominamos de Recuperação de Vista de Grupo.

O objetivo pretendido era o de, a partir do contacto com outros nodos da rede, ficar a conhecer rapidamente o número de grupos existente bem como povoar a vista de grupo com as referências para os nodos que se incluem no mesmo grupo em que o nodo mapeia (recorde-se que para uma determinada estimativa do número de grupos ser válida tem de ser suportada por um tamanho de vista de grupo compreendida no intervalo de replicação adotado). Os nodos que se encontram em melhores condições de providenciar esta informação são os que, atingido o estado de convergência, partilham o mesmo grupo que o nodo em questão, já que conhecem a totalidade dos nodos que formam o grupo. Apesar do nodo não conhecer à partida tais nodos, estes, por outro lado, são capazes de, a partir da identificação do nodo, reconhecê-lo como pertencente aos seus próprios grupos.

Desta forma, a estratégia passou por disseminar na rede uma mensagem especial de PSS solicitando aos nodos pertencentes ao grupo onde o novo nodo se inclui que lhe enviem as suas estimativas para o número de grupos, juntamente com a vista de grupo respetiva. Um nodo que receba uma mensagem deste tipo, conhecendo o número de grupos existente, sabe exatamente em que grupo o mesmo se inclui. Desta forma, envia-lhe uma mensagem com as informações requisitadas caso pertença ao mesmo grupo, ou propaga a mensagem para eventuais nodos que conheça do grupo em questão.

Com o intuito de validar que a extensão realizada ao algoritmo de construção de grupos permite, efetivamente, uma transferência de estado mais rápida que o algoritmo tradicional, procedeu-se à realização de uma experiência sobre uma rede de 500 nodos reais instalados

sobre instâncias Google Cloud do tipo *g1-small*(Este tipo de instâncias é semelhante às utilizadas na avaliação do sistema, pelo que no capítulo 5 é possível encontrar informações mais detalhas acerca das configurações das mesmas). Esta rede foi configurada de forma a que fossem formados 128 grupos, isto é, uma média de 4 nodos por grupo. A experiência procedeu-se da seguinte forma: A rede foi inicializada, tendo-se aguardado até que a mesma se encontrasse estável; De seguida, foram adicionados dois nodos à rede, um que executa o algoritmo tradicional e outro o algoritmo extendido, tendo-se registada a evolução acerca do conhecimento que estes nodos possuem da rede. De referir que a adição destes nodos não altera o número de grupos presentes na rede.

No gráfico da Figura 8, expõe-se a visão que cada um dos nodos introduzidos possui à cerca do número de grupos existente, a partir do momento que foram adicionados. Encontra-se, ainda, explicitado o momento em que convergiram para o estado final, isto é, que possuem um conhecimento perfeito acerca do número de grupos existente, encontrando-se a sua vista de grupo coerente com tal estimativa.

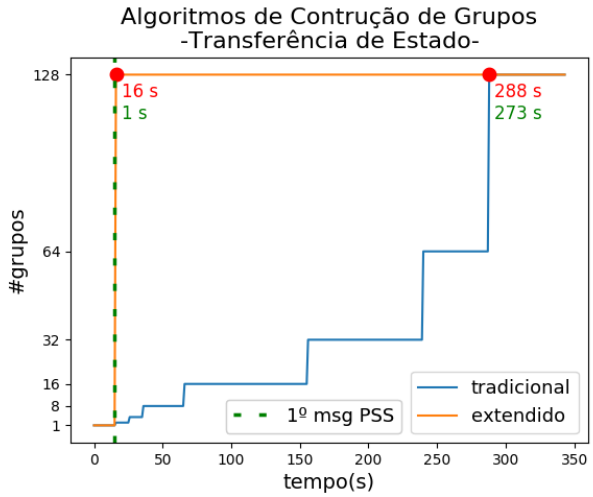


Figura 8: Transferência de estado dos algoritmos de construção de grupos tradicional/extendido

Tal como é possível observar a extensão do algoritmo de construção de grupos exibe uma transferência de estado imediata. De facto, no período inicial onde ambos os algoritmos mantêm conhecimento de um único grupo, o nodo encontra-se em fase de inicialização, onde nenhuma troca de mensagens PSS ocorreu (recorde-se que os algoritmos de construção de grupos são alimentados por mensagens de PSS). A partir do envio da primeira mensagem de PSS, o algoritmo extendido demora apenas 1 segundo para convergir para o número de grupos real, enquanto que, o algoritmo tradicional demora 273 segundos, tratando-se de uma diferença bastante significativa. Verifica-se ainda, tal como explicado anteriormente, o padrão em escada de evolução do algoritmo tradicional, passando por todos os patamares

de número de grupos admissíveis até atingir o estado final. Prova-se, desta forma, a eficácia da extensão realizada ao algoritmo.

4.5.4 *Anti-Entropy - Operação em 3 fases*

A estratégia de como no DataFlasks se resolve o problema de recuperação de chaves passa pelo anúncio periódico de todas as chaves que cada nodo armazena. Esta estratégia encontra-se inadequada para um cenário realista onde o número de chaves armazenadas em cada nodo é, potencialmente, muito elevado. Primeiramente, devido à impossibilidade de fazer caber num único pacote UDP todas as chaves armazenadas. Em segundo lugar, mesmo procedendo à divisão das chaves por vários pacotes UDP, é impraticável cada nodo da rede se encontrar constantemente, numa periodicidade minimamente reduzida, a anunciar todas as chaves que possui, dado que tal introduziria um sobrecarga considerável na rede.

De forma a colmatar este problema o desenho e algoritmo do componente *Anti-Entropy*, responsável pela recuperação das chaves, foi no LSFS alvo de uma reestruturação. O ciclo de vida deste componente foi organizado segundo 3 fases explícitas: inicialização, recuperação e operação.

4.5.4.1 *Fase de Inicialização*

O componente de Anti-Entropia possui uma dependência direta do componente de construção de grupos. A função principal do primeiro é a de recuperar chaves não armazenadas pelo nodo, mas que mapeiam no seu grupo. A função do último consiste na divisão dos nodos em grupos, cálculo do número de grupos existentes e identificação do grupo em que o nodo se inclui. Facilmente se pode concluir que um conhecimento mal formado acerca número de grupos existentes, posicionaria o nodo num grupo distinto do real, podendo este ser mais ou menos abrangente, levando-o a armazenar chaves desnecessárias ou a descartar chaves que o mesmo deveria armazenar. A mesma situação é aplicável à recuperação de chaves por parte do componente de Anti-Entropia. Considere-se, por exemplo, um novo nodo que entra numa rede estável formada por 128 grupos, o conhecimento inicial que possui do sistema é de um único grupo, enquanto que o componente construção de grupos não refina essa estimativa. Nesse intervalo de tempo, todas as potenciais chaves que lhe pudessem chegar por mecanismos de anti-entropia seriam incorporados na sua base de dados, uma vez que todas essas chaves mapeariam no suposto grupo global a que o nodo pertence.

Esta primeira fase serve o propósito de contornar esse problema, impedindo o componente de Anti-Entropia de realizar qualquer tarefa enquanto o componente de Group Construction não atinge o estado de convergência final. Em termos práticos isto reflete-se numa espera

passiva pela conclusão do protocolo de recuperação de vista de grupo descrito na Subsecção 4.5.3.

4.5.4.2 *Fase de Recuperação*

Uma vez culminada a fase de inicialização, um nodo encontra-se devidamente atualizado relativamente à posição que ocupa na rede, tendo conhecimento dos restantes nodos que partilham o mesmo grupo. Contudo, este não se encontra ainda em condições de responder a pedidos ao cliente já que a sua base de dados encontra-se inconsistente relativamente à dos outros nodos do grupo. Esta segunda fase tem como objetivo, pois, a recuperação da base de dados por parte do nodo.

Tal como mencionado em cima, torna-se impraticável num sistema real anunciar periodicamente todas as chaves incluídas na base de dados de um determinado nodo, já que o seu número é potencialmente muito grande. Anunciar iterativamente um subconjunto das mesmas é uma solução possível mas que requer um número substancialmente elevado de iterações para que um novo nodo incluído pela primeira vez no sistema consiga recuperar.

A solução adotada passa pelo novo nodo, conhecendo os restantes nodos do grupo em que se inclui, escolher aleatoriamente um para enviar uma mensagem a requisitar o estabelecimento de uma interação direta para transferência da base de dados. Perante a receção de um pedido deste tipo, o nodo alvo começa por verificar se, consoante a posição do nodo emissor, o mesmo se inclui no mesmo grupo que o próprio. Esta verificação é realizada de forma a evitar que o nodo recupere a partir de uma base de dados que não é compatível com o seu grupo. Caso seja compatível, é estabelecida uma conexão TCP entre os dois, e transferidas sequencialmente todas as chaves armazenadas pelo nodo contactado. É de referir que pode acontecer que algumas das chaves enviadas não pertençam ao grupo em questão, podendo ter origem em algum momento do tempo em que o número de grupos existente era distinto do atual, contudo, apenas as chaves que mapeiam no grupo corrente seriam incorporadas pelo novo nodo. A escolha do TCP encontra-se relacionada com a natureza desta tarefa. O que se pretende é garantir que todas as chaves pertencentes ao grupo do nodo são incorporadas pelo mesmo. Tal garantia é difícil de obter num protocolo de transporte não confiável como o UDP. Outro motivo relaciona-se com o facto do TCP efetuar controlo de congestionamento, o que é desejável neste caso, já que se espera que os momentos de recuperação dos nodos coincidam com picos na utilização da rede.

4.5.4.3 *Fase de Operação*

A última fase corresponde à fase de operação. O nodo encontra-se devidamente atualizado e pronto para responder aos pedidos dos clientes. Uma vez atingida esta fase o componente de anti-entropia permanece nesta até que o nodo falhe ou deixe o sistema. O objetivo desta fase é que os nodos consigam recuperar chaves que, numa execução normal e estável,

deveriam ter sido incorporadas na sua base de dados. Alguns dos motivos pelos quais tais chaves podem não ter sido incorporadas correspondem a possíveis erros de escrita, falhas na rede ou fragmentações temporária da mesma, etc.

Para que uma determinada escrita seja completada é necessário que um subconjunto dos nodos responsáveis por armazenar a chave envolvida, efetivamente a armazene e reporte para o cliente o sucesso na escrita. Parte-se, então, do principio que existe sempre um subconjunto de réplicas dentro de um grupo que possui uma determinada chave. De forma a detetar que uma chave está em falta é vital que as réplicas que possuem tal chave a anunciem às restantes. Decidiu-se, pois, aplicar uma estratégia semelhante à implementação já existente do DataFlasks onde cada nodo, periodicamente, anuncia as suas chaves. A única diferença aplicada foi na quantidade de chaves anunciadas em cada iteração. Em vez do conjunto total das chaves, é anunciado apenas, em cada momento, um subconjunto contíguo de chaves. A nível de implementação tal corresponde ao cálculo de um *offset* aleatório no universo total das chaves, e à seleção das n chaves que sucedem imediatamente esse *offset*, sendo que n constitui um parâmetro configurável.

Este modo de recuperação de chaves vai de encontro às garantias de consistência eventual que se procura oferecer por parte do sistema de ficheiros, na medida em que, inevitavelmente, todas as chaves são anunciadas dentro de um grupo e recuperadas pelas réplicas que não as possuem.

4.6 PROTÓTIPO LSFS

A implementação do protótipo do sistema de ficheiros foi realizada totalmente em C++. Para a distribuição das chaves pelos nodos de armazenamento foi utilizada a função hash definida por defeito para *strings* na biblioteca *C++ standard library*. Verificámos, por comparação com implementações providenciadas por outras bibliotecas, que esta conseguia manter um bom equilíbrio entre dispersão e eficiência.

Toda a comunicação entre os nodos é realizada, à exceção da recuperação inicial da base de dados, recorrendo a UDP como protocolo de transporte. Para o contacto, cada nodo tem conhecimento apenas do endereço ip dos nodos que deseja contactar, pelo que se assume a presença de todos os nodos numa mesma rede. Neste sentido, cada nodo de armazenamento bem como o cliente executam a sua própria instância de um servidor UDP assíncrono que gere um pool de threads. A implementação deste servidor fez-se com recurso à biblioteca *boost*[Boost,] v1.71.0. Foi ainda adotado *protocol buffers*[Protobuf,] v3.11.4 como tecnologia de serialização.

5

AVALIAÇÃO

Tendo sido apresentadas as principais decisões arquiteturais e de implementação postas em prática na elaboração do protótipo do LSFS, será dado agora especial destaque à componente da sua avaliação. Na secção 5.1 é descrita a metodologia de testes definida e nas secções 5.2 e 5.3 são apresentados e analisados os resultados obtidos para as diferentes experiências realizadas.

5.1 METODOLOGIA

Primeiramente, serão descritos os objetivos da avaliação, isto é, o que se pretende avaliar com cada uma das experiências. Seguidamente, serão especificados os diferentes tipos e conjuntos de experiências realizados. Por último, dar-se-á a conhecer o ambiente de execução de cada uma das experiências.

5.1.1 *Objetivos*

Desde início, a principal motivação para a construção de um sistema com a arquitetura que exibe o LSFS foi a de, endereçando redes de larga escala caracterizadas pela instabilidade que apresentam, assegurar a persistência dos dados bem como um desempenho estável, mesmo perante a ocorrência de falhas. Por outro lado, a viabilidade deste sistema encontrava-se ainda fortemente vinculada à capacidade que o mesmo tem para escalar horizontalmente a partir da inclusão de mais nodos.

Posto isto, a metodologia de testes foi desenhada com o objetivo de avaliar três aspetos fundamentais:

- **Resiliência** $\longrightarrow$ O sistema deve ser capaz de assegurar a persistência dos dados. Por outras palavras, tal significa que a ocorrência de falhas não deveria poder levar o sistema de ficheiros para um estado do qual não o mesmo não consiga recuperar. Pretende-se, desta forma, verificar qual é o comportamento do sistema perante a introdução de diferentes níveis de falha.

- **Escalabilidade** $\longrightarrow$ O LSFS pretende tirar proveito do espaço de armazenamento de centenas a milhares dispositivos de capacidade reduzida. A capacidade do sistema escalar para a quantidade de dispositivos considerada é uma condição necessária para que tal seja possível. A escalabilidade do LSFS será analisada para redes com até 500 nodos.

- **Desempenho** $\longrightarrow$ Como em qualquer sistema de gestão de dados o LSFS deve garantir um nível de usabilidade adequado. O desempenho surge como uma das principais formas de medir essa usabilidade. O desempenho do LSFS será medido para diferentes cargas de trabalho, quer com recursos a testes por exaustão, micro testes, como para um caso de uso real.

5.1.2 *Cargas de trabalho*

A fim de cumprir com cada um dos objetivos definidos, o sistema foi sujeito a duas cargas de trabalho distintas, os micro e macro testes.

Os micro testes correspondem a uma carga de trabalho sintética e exaustiva. Com este tipo de testes pretendeu-se avaliar, essencialmente, o desempenho máximo que era possível obter com a utilização do LSFS. O desempenho do LSFS foi medido tanto para operações de escrita como de leitura.

Por outro lado, o propósito dos macro testes era perceber como é que o sistema de ficheiros se comportava no contexto de um caso de estudo real. Quanto ao caso de estudo selecionado, encontra-se inserido num tema atualmente em voga que são os processos de aprendizagem máquina. No nosso caso em específico o sistema de ficheiros é avaliado de acordo com a sua prestação no treino, com recurso à framework Tensorflow, de uma rede neuronal para classificação de imagens. Uma vez mais o desempenho do sistema foi um dos aspetos avaliados, contudo, o principal foco incidiu em analisar a escalabilidade e resiliência oferecida pelo LSFS. Para tal, neste tipo de testes recorre-se a redes de maiores dimensões para a instalação do LSFS, bem como, são incluídos testes onde se aplica instabilidade à rede pela remoção e introdução de nodos.

5.1.3 *Ambiente Experimental*

Para que as experiências se pudessem aproximar o mais possível de um contexto real de uma rede de larga escala e não havendo possibilidade de montar uma rede de tal dimensão num domínio local, dada a insuficiência de recursos, houve necessidade de utilizar a nuvem como ambiente de testes. O provedor de nuvem onde as experiências foram realizados foi a

Google Cloud. A nuvem permitiu-nos emular uma rede de nodos de larga escala através da criação de múltiplas máquinas independentes com acesso a recursos limitados.

Diferentes tipos de máquinas foram utilizadas para servir propósitos distintos. Para os nodos de armazenamento, de forma a garantir uma maior proximidade a dispositivos IoT de pequenas dimensões, utilizaram-se máquinas com poucos recursos, quer a nível de armazenamento como processamento. Estas, correspondem, na nomenclatura utilizada pela *Google*, a máquinas do tipo *g1-small*, dotadas de 1 vCPU, 1.7 GB de RAM. O disco, por defeito, presente nestas instâncias são, tal como em qualquer outra, discos rígidos. As experiências preliminares realizadas revelaram uma alta variabilidade no desempenho deste tipo de discos. Por esta razão, todas as instâncias foram modificadas de forma a incluir um disco ssd (*pd-ssd*) como disco de inicialização. Nestas instâncias em particular trata-se de um disco de 15GB de armazenamento.

Com o intuito de possuir um maior controlo sobre a rede, estas máquinas foram inseridas no contexto de um cluster de Kubernetes[Kubernetes,], encontrado-se a ser geridas pelo mesmo. Cada nodo da rede é na verdade um POD, que corresponde, no modelo de objetos de Kubernetes, à unidade de processamento mais básica que pode ser instalada. Internamente executam sob a forma de containers Docker. Estes Pods são alocados e distribuídos, no momento da sua criação, pelas diferentes máquinas que fazem parte do cluster.

Para que esta gestão dos Pods fosse possível, foi necessário incluir na arquitetura uma outra máquina que corresponde ao nodo mestre de kubernetes. Esta máquina é do tipo n1-standard-16 equipada com 16 vCPUs, 60 GB de RAM, e 12 GB de disco (tal constitui o tipo de máquina de referência para clusters de até 500 máquinas). Ainda para facilitar a comunicação entre os Pods, uma vez que cada Pod executa numa máquina distinta, foi instalada uma rede sobreposta Flannel que se encarrega de realizar o encaminhamento do tráfego entre as máquinas. Desta forma, problemas como a realização de "port forwarding" entre Pods deixam de ser um problema.

A Arquitetura de testes não se encontraria completa sem um nodo onde fosse instalado o cliente do sistemas de ficheiros. Para este, foi reservada uma máquina com um poder de processamento superior comparativamente aos nodos LSFS. Tal máquina é do tipo n1-standard-4, dotada de 4 vCPUs e 15GB de RAM. Também a nível de armazenamento foi alocado um disco maior. A especificação do tamanho do disco é fornecida junto à análise dos resultados, dado variar em função do tipo de teste realizado. Ainda para os testes do Tensorflow foi adicionada a esta máquina uma unidade de processamento gráfico Nvidia Tesla T4.

De forma a facilitar a execução dos testes bem como providenciar acesso a cada uma das máquinas referidas foi adicionada à arquitetura mais uma máquina, do tipo n1-standard-4, dotada de um endereço externo para comunicação com o exterior. Todos os testes foram

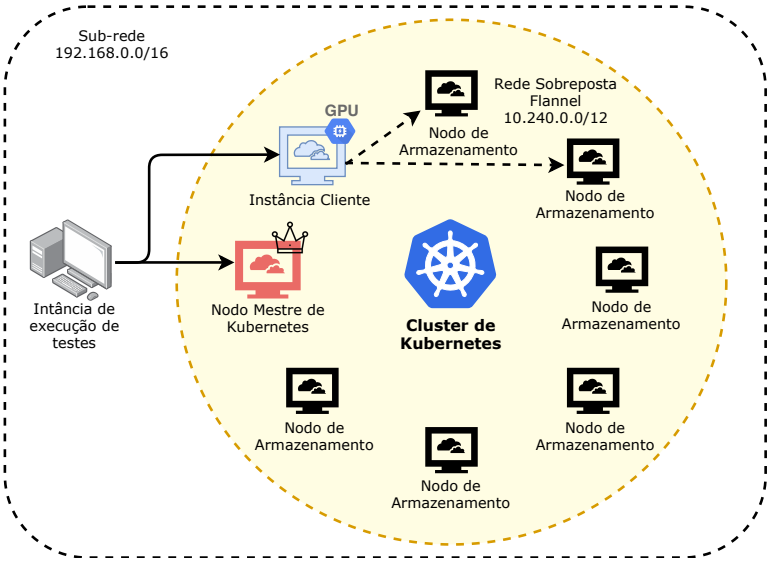


Figura 9: Ambiente Experimental

realizados a partir desta máquina recorrendo a scripts Ansible e Python. Na Figura 9 encontra-se representada uma visão completa de toda esta arquitetura de testes descrita.

Todas as máquinas referidas foram monitorizadas durante o período de execução de cada uma das experiências. Como ferramentas de monitorização foram utilizadas o dstat e iostat para controlar os níveis de disco, cpu, memória e rede de cada uma das máquinas. Para os testes com o Tensorflow recorreu-se ainda à ferramenta nvidia-smi com o intuito de monitorizar a utilização dos recursos do gpu. Os registos referentes a cada uma das métricas colecionadas foram largamente utilizados para motivos de depuração, bem como, auxiliarão o processo de análise dos resultados.

5.1.4 *Gestão de Pods*

A adoção do modelo de Kubernetes providenciou-nos grande flexibilidade na gestão da rede de nodos. Sobretudo para os testes de resiliência, onde era necessário introduzir a falha de nodos e a criação de novos, era impraticável a sua implementação pela criação e remoção efetiva de máquinas. Desta forma, torna-se apenas necessário instruir o nodo mestre do cluster, para remover determinados pods e criar outros com as configurações pretendidas.

A forma como um cluster de Kubernetes é gerido, por defeito, consiste na instanciação de pods de acordo com os recursos disponíveis em cada máquina. Para o cenário que pretendíamos construir foi necessário o estabelecimento de determinadas regras na alocação destes pods. Em primeiro lugar, cada nodo de armazenamento só se podia encontrar responsável por um e um só nodo da rede. Em segundo lugar, a máquina cliente não podia ser utilizada para executar nenhum pod para além do cliente LSFS. A forma como

tal foi garantido passou pela utilização de uma funcionalidade de Kubernetes que consiste em aplicar um rótulo a cada uma das diferentes máquinas. A instanciação dos Pods é, posteriormente, realizada com base nesses rótulos. Por exemplo, no caso da inicialização de uma rede de 500 nodos, existem 500 máquinas criadas com o intuito de servir de suporte a esses nodos. Cada uma é rótulada de acordo com um identificador crescente. No momento da instanciação dos pods são passados identificadores correspondentes a máquinas distintas aos diferentes pods, onde os mesmos devem ser alocados.

No que respeita a introdução de instabilidade no sistema, é mantido conhecimento acerca do identificador da máquina onde cada um dos nodos ativos se encontra em execução. Em cada período de instabilidade, para cada um dos nodos selecionados para remoção, são guardados os identificadores das máquinas a que foram alocados. Tais identificadores são utilizados de forma a que a introdução de novos nodos ocorra apenas sobre máquinas livres, isto é, onde nenhum outro nodo se encontra em execução.

5.2 MICRO TESTES

Para além de se pretender, com este tipo de carga de trabalho, avaliar o desempenho máximo que o LSFS é capaz de oferecer, os micro testes foram ainda realizados com um segundo propósito, nomeadamente, um propósito experimental. No decorrer da implementação do LSFS algumas otimizações foram introduzidas com o intuito de melhorar o desempenho do sistema de ficheiros. Duas destas considerámos possuir um impacto bastante significativo a nível do desempenho. A primeira consiste na utilização do *Smart Load Balancer* em detrimento do *Random Load Balancer*. A segunda corresponde à introdução de paralelismo. Pretende-se, pois, com estes testes verificar o impacto que cada uma destas otimizações possui no desempenho do sistema de ficheiros, bem como, identificar a configuração que permite oferecer o desempenho máximo.

Neste sentido, diferentes combinações das referidas otimizações foram utilizadas. Cada uma destas é identificada de acordo com o tipo de balanceador de carga que utiliza (random/smart), bem como de acordo com o nível de paralelismo em função da operação em estudo (xk, onde x corresponde à quantidade máxima de dados que pode ser requisitada/-submetida de forma paralela). A utilização do identificador "lsfs-random-32k", por exemplo, para uma dada carga de leitura, indica que foi utilizado uma instalação do LSFS que recorre ao *Random Load Balancer* e onde o parâmetro *limit_read_paralelization_to* foi definido para 32KiB (O significado de tal parâmetro encontra-se descrito na Secção 4.4). Todas as diferentes configurações definidas para o LSFS foram testadas sobre uma rede de 100 nodos.

Para o substrato foram definidas as seguintes configurações. O intervalo de troca de mensagens de Peer Sampling é de 10 segundos para os nodos de armazenamento, e de 5 segundos para o cliente. Cada nodo armazena uma vista de tamanho 8, sendo 7 o número

de registos substituídos em cada troca de mensagens. O algoritmo de anti-entropia foi configurado para executar em intervalos de 20 segundos. Quanto ao grau de replicação, foi definido um número mínimo de 3 nodos por grupo e um máximo de 5. Tal corresponde a uma rede formada por 32 grupos e a uma média de 3 nodos por grupo. Para as configurações que utilizam o *Smart Load Balancer* estabeleceu-se um conhecimento para o mesmo de 3 nodos por grupo, consistindo num conhecimento perfeito da rede.

A mera apresentação dos resultados de desempenho obtidos pelas diferentes configurações do LSFS por si só não nos oferece uma perceção clara do desempenho efetivo do LSFS. Diversos fatores, sobretudo num contexto distribuído, influenciam a velocidade de resposta a que os pedidos são satisfeitos. Por exemplo, será indubitavelmente distinto a nível do desempenho oferecido se, para uma mesma configuração do LSFS, forem utilizadas máquinas com mais ou menos recursos para os nodos de armazenamento. Neste sentido, de forma a melhor compreender os resultados, alguns testes preliminares foram realizados. O objetivo aqui foi o de, a partir de um conjunto reduzido de testes, tentar providenciar uma ideia do que seria o desempenho máximo expectável do LSFS tendo em conta a sua arquitetura e ambiente utilizado para os testes. Os testes preliminares consistiram na execução dos mesmos testes realizados sobre o LSFS mas, para 3 diferentes configurações locais. As primeiras duas correspondem a um setup puramente local, tendo-se utilizado duas máquinas distintas: uma máquina igual à utilizada para o cliente nas configurações do LSFS e outra correspondente à de um nodo de armazenamento. Os testes realizados com o primeiro tipo de máquinas serão identificados como "local-nodo-cliente", e os com a máquina do segundo tipo por "local-nodo-armazenamento". A terceira configuração é semelhante à primeira, mas utiliza o Fuse como camada de indireção. Esta configuração é identificada pelo nome "fuse-local-cliente". É de referir ainda que para este conjunto de testes a capacidade de armazenamento escolhida para o nodo cliente foi de 64 GiB, de forma a permitir um período de experiência mais extenso para o caso das configurações locais.

A totalidade dos testes de micro carga realizados formam o 1º conjunto de experiências. De seguida, proceder-se-á à sua especificação e discussão de resultados.

5.2.1 1º *conjunto de experiências*

Para o conjunto de experiências em questão recorreu-se à ferramenta de análise Filebench[Tarasov et al., 2016], quer para a realização das experiências quer para a medição do desempenho. A seleção do Filebench deveu-se à popularidade do mesmo como ferramenta de *benchmarking* de sistema de ficheiros, bem como à facilidade que o mesmo oferece de, através da sua flexível *Workload Model Language*, especificar diferentes cargas de trabalho.

As cargas de trabalho utilizadas dividem-se em cargas de leitura e de escrita. Cada uma recorre a uma thread que iterativamente e durante um período de tempo fixo realiza o maior

número de operações de escrita ou leitura que consegue, em blocos de 4 ou 32KiB sobre um ficheiro. Para cada carga, operações de escrita não alternam com operações de leitura pelo que cargas de leitura são precedidas de um povoamento integral dos dados a serem lidos. As cargas de leitura distinguem-se ainda das de escrita no que respeita à distribuição das operações na medida em que cargas de escritas realizam-se exclusivamente de modo sequencial contrariamente às leituras que podem ser de natureza sequencial ou aleatória.

De forma a identificar cada uma das diferentes cargas de trabalho são utilizadas as abreviaturas: seq e aleat, para identificar, respetivamente, as distribuições sequenciais e aleatórias das operações; leitura/escrita diferencia o tipo de carga em leitura ou escrita; Yk, representa o tamanho do bloco em KiB que cada operação lê/escreve. Seguindo esta terminologia e a título de exemplo, a carga de trabalho identificada por leitura-seq-4k refere-se a uma carga formada por leituras sequencias de blocos de 4KiB de tamanho.

Cada uma das cargas de trabalho foi configurada para correr durante 20 minutos, ou, no caso das configurações locais, até um limite estipulado face ao poder de armazenamento da máquina em questão, nomeadamente 50GiB para a configuração "local-nodo-cliente" e 8GiB para a configuração "local-nodo-armazenamento". Para cada uma das cargas de trabalho, o Filebench calcula a latência, em milissegundos por operação, e o débito, quer em operações por segundo como em MiB por segundo. Para a análise dos testes será dado especial relevo ao débito em MiB/s. Cada experiência foi executada 3 vezes pelo que o débito analisado constitui a média das 3 experiências. Para além disso, é ainda apresentado o desvio padrão obtido.

5.2.1.1 *Configuração local*

Tal como descrito, o principal objetivo inerente à realização destes testes numa configuração local não é, pois, o de servir como termo de comparação do LSFS, mas perceber de que forma o desempenho do LSFS se encontra limitado por fatores externos ao mesmo.

Neste sentido, selecionou-se a configuração *local_nodo_cliente* como configuração local base. Esta configuração permite estabelecer um teto máximo de desempenho idealmente atingível pelo LSFS caso não contemplasse distribuição. As Tabelas 3 e 4 mostram os débitos de escrita e leitura, respetivamente, obtidos para as diferentes experiências por parte de cada um das configurações utilizadas. O desempenho obtido pela configuração base descrita é superior a 800 MiB/s nos testes de escrita, cerca de 30 MiB/s para leituras sequenciais, 8,37 MiB/s para leituras aleatórias de blocos de 4KiB e 36,1 para leituras aleatórias de blocos de 32KiB.

É possível realizar algumas observações face aos valores apresentados. Primeiramente, é notável a discrepância na ordem de grandeza do débito entre leituras e escritas. A razão para tal acontecer deve-se a um conceito denominado de *writeback*, utilizado por parte de sistemas Linux. Tal conceito consiste na utilização de uma zona mantida em memória, a que se dá o nome de *page cache*, para o armazenamento de escritas sem a necessidade de as propagar

diretamente para disco. Posteriormente, as alterações atingem o disco em *background* e em *batches*. O acesso a memória sendo bastante mais rápido que o acesso a disco aumenta em muito o desempenho de escrita por parte destes sistemas. Verifica-se ainda a ocorrência de um débito semelhante nas diferentes escritas sequenciais, apesar do tamanho do bloco lido ser distinto. A razão para isto acontecer encontra-se na realização de *readahead*, isto é, em trazer para memória mais dados que os efetivamente requisitados. Desta forma, em ambos os casos, é semelhante a quantidade de dados, utilizada na satisfação de pedidos, que é obtida a partir de memória. Por último, não é observável o mesmo padrão para as leituras aleatórias, já que só é possível tirar proveito do *readahead* para blocos maiores que 4KiB, o que justifica as divergências ocorridas.

			Média (MiB/s)	$\sigma(MiB/s)$
Escrita	seq-escrita-4k	local_nodo_cliente	824,2 MiB/s	43,76 MiB/s
		local_nodo_armazenamento	28,5 MiB/s	0 MiB/s
		fuse_local_cliente	92,3 MiB/s	1,51 MiB/s
	seq-escrita-32k	local_nodo_cliente	866,53 MiB/s	8,69 MiB/s
		local_nodo_armazenamento	28,47 MiB/s	0,06 MiB/s
		fuse_local_cliente	381,5 MiB/s	5,95 MiB/s

Tabela 3: Resultados Filebench - Operações de escrita (*Setup* Local)

Apesar de se ter utilizado esta mesma máquina como cliente de sistema de ficheiros no LSFS, a verdade é que, neste último, é nos nodos de armazenamento que os dados se encontram efetivamente armazenados. Posto isto, sendo as máquinas utilizadas para efeitos de armazenamento bastante limitadas a nível de recursos, também o desempenho que estas oferecem será um fator limitante do desempenho do próprio LSFS. A configuração *local_nodo_armazenamento* pretende a dar a conhecer o impacto no desempenho da utilização deste tipo de máquinas.

Para o caso das escritas, o desempenho é reduzido a 28,5 MiB/s. Este decréscimo abrupto advém muito provavelmente da reduzida quantidade de RAM disponível nestas máquinas (1,7 GB em vez de 15 GB), que inibe ou dificulta a utilização do modelo *writeback*. Também nas leituras o decréscimo do débito se fez sentir, variando entre 15 a 25% para as diferentes cargas de trabalho. Para além da RAM, também o disco contribuiu para esta redução, já que no modelo da *Google Cloud* quanto menor for o tamanho do disco menor o débito que o mesmo oferece, como declarado em [Google,].

Por último, também a utilização do *Fuse*, ao introduzir uma camada extra de indireção ao LSFS, constituía um potencial fator penalizador do desempenho do mesmo. Na configuração *fuse_local_cliente* procurou-se verificar de que forma a sua utilização introduzia ou não perdas no desempenho comparativamente à configuração de base. Em contraste com a última, o *Fuse* é nesta utilizado num modelo *pass-through*.

			Média (MiB/s)	$\sigma(MiB/s)$
Leitura	leitura-seq-4k	local_nodo_cliente	30,53 MiB/s	0,4 MiB/s
		local_nodo_armazenamento	26,1 MiB/s	0 MiB/s
		fuse_local_cliente	30,53 MiB/s	0,41 MiB/s
	leitura-seq-32k	local_nodo_cliente	30,3 MiB/s	0,06 MiB/s
		local_nodo_armazenamento	26,07 MiB/s	0,06 MiB/s
		fuse_local_cliente	30,3 MiB/s	0 MiB/s
	leitura-aleat-4k	local_nodo_cliente	8,37 MiB/s	0,12 MiB/s
		local_nodo_armazenamento	6,33 MiB/s	0,47 MiB/s
		fuse_local_cliente	8,37 MiB/s	0,12 MiB/s
	leitura-aleat-32k	local_nodo_cliente	36,1 MiB/s	0,44 MiB/s
		local_nodo_armazenamento	29,57 MiB/s	0,06 MiB/s
		fuse_local_cliente	36,1 MiB/s	0,46 MiB/s

Tabela 4: Resultados Filebench - Operações de leitura (*Setup* Local)

No que respeita a leituras não se verificaram grandes diferenças de desempenho entre ambas as configurações (Tabela 4). De facto, também o *Fuse* põe em prática algumas otimizações, tais como *readahead*, que o permitem obter um desempenho considerável. Contrariamente, para as escritas o débito diminuiu de 824,2 MiB/s para 92,3 MiB/s, para blocos de 4KiB, e de 866,53 MiB/s para 381,5 MiB/s, para blocos de 32KiB. As diferenças ocorridas dizem respeito, essencialmente, ao facto de no *Fuse* o modelo *writeback* se encontrar desativado por defeito[Vangoor et al., 2017]. Tal facto, permite ainda explicar o porquê de se ter atingido um débito maior em blocos de 32KiB. Dado que as escritas são realizadas diretamente sobre o disco e não agrupadas em *batches* em memória, o Fuse consegue tirar proveito de submeter um menor número de escritas para disco quanto maior for o tamanho do bloco escrito.

5.2.1.2 *Escritas - Desempenho do LSFS*

Na Tabela 5 apresentam-se os débitos obtidos para as cargas de trabalho de escrita referentes às diferentes combinações de otimização aplicadas ao LSFS. Tal como é observável, o LSFS atinge um débito máximo de 5,07 MiB/s para escritas de blocos de 4KiB e 20,43 MiB/s para blocos de 32KiB.

Relativamente às escritas de 4KiB, apenas o tipo de balanceador de carga utilizado têm um impacto visível no débito oferecido. Tal facto vai de encontro ao expectável na medida em que 4KiB constitui o tamanho pré-definido para os blocos armazenados no substrato. Neste sentido, fazer variar o grau de paralelismo admissível para as escritas em nada altera o comportamento do sistema, na medida em que não é possível paralelizar a escrita de um bloco de tal tamanho. Já as diferenças observadas para os diferentes balanceadores advém do próprio funcionamento inerente aos mesmos. A utilização do *random load balancer* introduz uma maior disseminação de mensagens na rede que é responsável por um aumento da carga

quer na rede como nos diferentes nodos. O aumento da carga na rede é ainda intensificado pelo tipo de pedidos em trânsito, que, correspondendo a pedidos *put*, carregam um *payload* de tamanho considerável. Esta sobrecarga na rede é ainda visível pelos resultados referentes à escrita de blocos de 32KiB sobre o mesmo balanceador de carga. Para este tamanho de bloco é possível paralelizar cada pedido até 8 vezes, contudo, uma paralelização de 2 vezes, correspondente à configuração "random_8k" foi suficiente para diminuir ligeiramente o débito obtido, evidenciando uma saturação da rede.

			Média (MiB/s)	$\sigma(MiB/s)$
Escrita	escrita-seq-4k	random_4k	2,2 MiB/s	0 MiB/s
		random_8k	2,2 MiB/s	0 MiB/s
		smart_4k	5,0 MiB/s	0 MiB/s
		smart_8k	4,96 MiB/s	0,06 MiB/s
		smart_16k	5,07 MiB/s	0,12 MiB/s
	escrita-seq-32k	random_4k	2,23 MiB/s	0,06 MiB/s
		random_8k	2,08 MiB/s	0,06 MiB/s
		smart_8k	8,27 MiB/s	0,06 MiB/s
		smart_16k	13,43 MiB/s	0,06 MiB/s
		smart_32k	20,43 MiB/s	0,42 MiB/s

Tabela 5: Resultados Filebench - Operações de escrita (LSFS)

Contrariamente, a utilização do *smart load balancer*, sendo caracterizada por uma redução significativa do número de mensagens em circulação na rede bem como do número de hops necessários à satisfação de cada pedido, permitiu beneficiar do aumento do paralelismo, tendo o débito aumentado gradualmente em função do estabelecimento de um maior grau de paralelismo admissível. Pelas mesmas razões às mencionadas para os blocos de 4KiB, o estabelecimento de um nível de paralelismo superior a 32KiB não traria qualquer vantagem face ao débito máximo apresentado.

5.2.1.3 *Escritas - Recursos Utilizados pelo LSFS*

Na Tabela 6 encontram-se apresentados a média de utilização de recursos por parte dos diferentes nodos constituintes do LSFS para as diferentes experiências de escrita de ficheiros realizadas. No que respeita o consumo de memória RAM, tanto para o cliente como para os nodos de armazenamento, os resultados evidenciam a manutenção de um mesmo nível de utilização em ambas as experiências, para as diferentes configurações de otimização do LSFS. Os níveis referidos rondam os 400 MiB para o nodo cliente e os 300 MiB para o caso de um nodo de armazenamento. Na implementação do LSFS houve um esforço acrescido no sentido limitar o consumo de memória utilizada, na sua maioria, estabelecendo-se valores máximos para o tamanho que as estruturas de dados utilizadas podiam atingir. Tal como veremos, estes valores vão se manter pouco variáveis para os diferentes conjuntos de experiências.

			Cliente		Nodos de Armaz.	
			CPU(%)	RAM(MiB)	CPU(%)	RAM(MiB)
Consumo de escritas	escrita-seq-4k	random-4k	6,27%	416,65 MiB	49,67%	317,93 MiB
		random-8k	6,12%	420,66 MiB	49,41%	320,43 MiB
		smart-4k	11,89%	408,80 MiB	5,96%	318,63 MiB
		smart-8k	7,70%	405,13 MiB	5,84%	318,19 MiB
		smart-16k	8,09%	420,06 MiB	5,83%	312,84 MiB
	escrita-seq-32k	random-4k	5,36%	418 MiB	49,94%	319,21 MiB
		random-8k	4,56%	418,90 MiB	43,39%	319,96 MiB
		smart-8k	15,82%	412,51 MiB	11,80%	316,12 MiB
		smart-16k	11,96%	402,72 MiB	13,35%	315,82 MiB
		smart-32k	15,56%	422,34 MiB	18,29%	315,86 MiB

Tabela 6: Recursos utilizados (em média) pelo LSFS para operações de escrita

Contrariamente à RAM, os resultados de CPU apresentam-se variáveis face às diferentes configurações do LSFS e tipos de escrita. Mapeando estes valores com os obtidos para o débito na Tabela 5, verifica-se a presença de um padrão de crescimento semelhante. De facto, de forma a providenciar um débito maior é necessário processar um maior número de mensagens, o que implica um aumento dos recursos de CPU. Verifica-se ainda uma diminuição considerável na utilização deste recurso para as configurações que recorrem ao *smart load balancer*. A razão para tal acontecer prende-se com a menor disseminação de mensagens na rede, que se traduz numa redução considerável no número de mensagens que cada nodo tem de processar.

5.2.1.4 *Leituras - Desempenho do LSFS*

Uma tabela semelhante à abordada para o desempenho nas escritas, apresenta-se na Tabela 7 para as experiências com operações de leitura. Contrariamente às escritas, estas são realizadas quer de modo sequencial como aleatório sobre um ficheiro pré-populado no LSFS. O modo adotado em cada experiência possui um grande impacto sobre o desempenho oferecido pelo sistema de ficheiros, tal como mostram os valores presentes na tabela. De facto, para leituras sequenciais é possível oferecer um débito que ronda os 27,5 MiB/s, enquanto que para leituras aleatórias o débito máximo que se conseguiu atingir foi de 4,5 MiB/s para blocos de 32KiB e 2,1 MiB/s para blocos de 4KiB.

A razão pela qual se observam tais diferenças relaciona-se com a maior ou menor facilidade de realização de *readahead* por parte do Fuse. Em escritas sequenciais, independentemente do tamanho de bloco utilizado, é favorável a execução de *readahead*. Comparando os débitos obtidos nas diferentes configurações do LSFS para leituras sequenciais de 4KiB e 32KiB, é visível uma variação mínima dos resultados. Efetivamente, seja 4KiB ou 32KiB o tamanho do bloco que o Filebench requisita ao Fuse, cada pedido é mapeado numa operação de leitura de tamanho igual a 128KiB, que constitui o valor definido, por defeito, como máximo

			Média (MiB/s)	$\sigma(MiB/s)$
Leitura	leitura-seq-4k	random_4k	2,4 MiB/s	0 MiB/s
		random_8k	2,53 MiB/s	0,06 MiB/s
		random_16k	2,17 MiB/s	0,06 MiB/s
		smart_64k	22,8 MiB/s	0,36 MiB/s
		smart_96k	27,27 MiB/s	0,29 MiB/s
		smart_128k	17,43 MiB/s	1,34 MiB/s
	leitura-seq-32k	random_4k	2,43 MiB/s	0,06 MiB/s
		random_8k	2,53 MiB/s	0,06 MiB/s
		random_16k	1,9 MiB/s	0,1 MiB/s
		smart_64k	23,03 MiB/s	0,38 MiB/s
		smart_96k	27,53 MiB/s	0,51 MiB/s
		smart_128k	22,13 MiB/s	1,86 MiB/s
	leitura-aleat-4k	random_4k	2,1 MiB/s	0,1 MiB/s
		smart_4k	1,9 MiB/s	0 MiB/s
	leitura-aleat-32k	random_4k	2,2 MiB/s	0 MiB/s
		random_8k	2,03 MiB/s	0,06 MiB/s
		random_16k	1,4 MiB/s	0 MiB/s
		smart_16k	2,33 MiB/s	0,06 MiB/s
		smart_32k	4,5 MiB/s	0,1 MiB/s

Tabela 7: Resultados Filebench - Operações de leitura (LSFS)

readahead. Ao utilizarmos um nível paralelismo inferior a 128KiB gere-se a percentagem desses dados que é efetivamente requisitada em cada momento. Os resultados apresentados mostram que o aumento do paralelismo tem um impacto maioritariamente benéfico no desempenho de ambos os balanceadores de carga. Naturalmente, tal aumento só é vantajoso até ao nível de saturação da rede, a partir do qual a perda de mensagens e a ocorrência de *timeouts* se tornam incomportáveis. O nível de saturação da rede é atingido aos 128KiB para o *smart load balancer* e aos 16KiB para o *random load balancer*. Uma vez mais se verifica a superioridade do *smart load balancer* relativamente ao *random load balancer* que ao adotar um estratégia de roteamento eficiente previne um sobreutilização dos recursos de rede.

Para as leituras aleatórias o caso muda de figura, já que a realização de *readahead* é dificultada pelo padrão de acessos ao ficheiro. De facto, para blocos de 4KiB o Fuse não tira qualquer vantagem da sua realização. As experiências realizadas vieram provar isso mesmo, tendo o débito se mantido constante para diferentes níveis de paralelismo. Na Tabela 7 apresentam-se, pois, unicamente, os resultados obtidos para uma configuração do LSFS com paralelismo limitado a 4KiB. Verifica-se que, apesar de um encaminhamento mais eficiente por parte do *smart load balancer*, para este tipo de leituras, o balanceador de carga aleatório consegue reduzir ligeiramente o débito, devido à menor complexidade de gestão da sua vista de roteamento bem como do processo de seleção dos nodos. Para blocos de 32KiB, o Fuse já é capaz de realizar algum *readahead*, razão pela qual o aumento de

paralelismo para 32KiB na configuração com o *smart load balancer* se traduziu num aumento do desempenho. É de notar ainda que, face às experiências com leituras sequenciais, os débitos são globalmente mais reduzidos para as leituras aleatórias. Atente-se, por exemplo, nas linhas leitura-seq-4k e leitura-aleat-4*k* na configuração random-4*k*, onde não se efetua paralelismo. Nesta situação, o débito decresce de 2, 4 MiB/s para 2, 1 MiB/s. A razão para a ocorrência deste decréscimo encontra-se uma vez mais no *readahead* que, apesar de não se concretizar já que é inibida a ocorrência de paralelismo, são na mesma, para o caso das leituras sequenciais, submetidos ao *Kernel* pedidos de 128KiB, traduzindo-se em menos pedidos realizados e num menor número de mudanças de contexto entre *Kernel* e espaço de utilizador.

5.2.1.5 *Leituras - Recursos Utilizados pelo LSFS*

O resultados de monitorização obtidos para as experiências de leitura revelaram seguir os mesmos padrões identificados para as escritas na Subsecção 5.2.1.3. Isto é, a RAM manteve-se, uma vez mais, constante em cerca de 400 MiB e 300 MiB para o cliente e nodos de armazenamento, respetivamente. As configurações do LSFS que recorrem ao balanceador de carga aleatório superaram as configurações do *smart load balancer* relativamente ao consumo de CPU, tendo-se verificado, em ambas as situações, uma variabilidade coerente com o débito oferecido. É de realçar ainda que para as configurações do LSFS que providenciaram um melhor desempenho, o nível de CPU manteve-se sempre, para todos os nodos, abaixo dos 25%. No Apêndice A.1 encontram-se expostos a totalidade dos resultados de monitorização para as experiências de leitura realizadas.

5.3 MACRO TESTES

Os macro testes foram divididos em dois conjuntos de experiências distintos, sendo estes o 2º e 3º conjunto de experiências. O 2º conjunto de experiências foca-se essencialmente em provar o desempenho e escalabilidade do sistema, enquanto que o 3º conjunto de experiências em verificar a resiliência que o mesmo oferece. Em ambos os conjuntos recorreu-se a uma rede formada por 500 nodos de armazenamento para a instalação do LSFS.

Os grupos de replicação foram configurados para o seu tamanho se encontrar compreendido entre 3 e 6. No tamanho da rede considerado, a estabilidade é garantida para um número de grupos igual a 128, o que equivale a uma fator de replicação médio de 4. Os resultados mostram que tal fator de replicação é suficiente para suportar consideráveis níveis de instabilidade do sistema. Quanto ao conjunto de dados utilizado para o treino da rede neuronal recorreu-se ao *dataset* completo da versão LSVRC2012 do ImageNet que possui cerca de 150GiB de tamanho. Ao encargo de cada nodo encontra-se, pois, aproximadamente

1.2GiB de dados. Uma vez que o povoamento da rede do LSFS é realizado a partir da máquina cliente, esta foi equipada com um disco de 200 GiB.

A rede neuronal utilizada para os testes foi a LeNet por constituir uma das redes neuronais convolucionais mais conhecidas para o efeito, bem como pelo facto de ser bastante intensiva a nível de IO, o que requer uma maior utilização por parte do sistema de ficheiros[Sarkar, 2019]. Relativamente à parametrização utilizada, adotou-se um tamanho de *batch* de 64 e 10000 como *shuffle buffer*.

Contrariamente ao 1º conjunto de experiências, onde diferentes combinações de otimizações foram postas em prática, nestas experiências, apenas uma configuração foi utilizada para os testes, nomeadamente a combinação que permitia atingir o melhor débito. Tal combinação corresponde à utilização do *Smart Load Balancer* em detrimento do *Random Load Balancer* e ao estabelecimento de um nível de paralelismo de leitura de 96KiB e 32KiB para o 2º e 3º conjunto de experiências, respetivamente. A razão da redução no nível de paralelismo para o 3º conjunto de experiências, encontra-se explicada na subsecção que lhe é dedicada (Subsecção 5.3.2). É de referir que o motivo pelo qual não é explicitado o valor do paralelismo para as escritas deve-se ao facto do treino da rede neuronal corresponder a uma carga de trabalho formada exclusivamente por leituras.

Ainda comparativamente ao 1º conjunto de experiências, foi necessário alterar os parâmetros referentes ao mecanismo de PSS no DataFlasks com o intuito de adaptá-lo a uma rede de dimensão bastante superior. Os parâmetros alterados foram o tamanho da vista mantida por cada nodo e o número de registos trocados em cada mensagem, que passaram para 20 e 15, respetivamente. O aumento referido foi realizado de forma a garantir a manutenção de boas propriedades de disseminação da rede. O aumento do tamanho da rede implicou ainda o conhecimento de um maior número de nodos por parte do *Smart Load Balancer*. Dado ser necessário refrescar um maior número de registos da sua vista, diminuiu-se o intervalo de envio de mensagens de PSS no cliente para 2 segundos.

5.3.1 2º *conjunto de experiências*

Neste conjunto de experiências o desempenho do sistema de ficheiros é avaliado no decorrer da execução de 3 épocas de treino da rede neuronal. Como métricas de desempenho definiu-se o tempo total necessário para completar o conjunto das 3 épocas, bem como o débito médio em imagens por segundo obtido ao longo das experiências realizadas.

A análise do desempenho não seria, contudo, possível se não existisse um termo de comparação sobre o qual pudéssemos tirar conclusões. A configuração utilizada como base de comparação pretende simular um contexto real, onde o procedimento habitual de treino de uma rede neuronal é realizado recorrendo a dados armazenados localmente, frequentemente num ambiente em nuvem. Para tal recorreu-se a uma única instância cliente,

para onde o *dataset* é carregado e onde o treino é realizado. Em ambas as configurações foram realizadas três repetições de forma a garantir consistência nos resultados, sendo o *dataset* pré-populado antes da realização de cada experiência.

Setup	**Tempo Médio de Treino - 3 épocas (segundos)**	**Débito Médio / Época (imagens / segundo)**
Local	$4798,33$ (σ = 1,53)	$861,41$ (σ = 55,15)
LSFS	$12584,0$ (σ = 42,23)	$326,57$ (σ = 23,21)

Tabela 8: Resultados Tensorflow - Comparação Local/LSFS(500 nodos)

A Tabela 8 mostra os resultados de treino obtidos em ambas as configurações. Os resultados mostram que o LSFS é cerca de $2,6$ vezes mais lento a treinar, tendo demorado cerca de 3 horas e meia a completar 3 épocas de treino. Em contraste, o sistema de ficheiros local demorou 1 hora e 20 minutos. Este resultado é interessante porque considera uma situação real, na qual o sistema de ficheiros não está constantemente a servir uma carga máxima de operações de armazenamento, como acontece nas experiências que recorrem ao FileBench. Assim, é possível identificar qual o compromisso de desempenho esperados de utilizar o LSFS ou uma instalação local, a qual apresenta um ponto único de falha, para aplicações reais.

5.3.1.1 *Recursos Utilizados*

Face aos resultados obtidos para o 1º conjunto de experiências, observou-se um aumento significativo quer da RAM quer do CPU utilizado pelo nodo cliente, ultrapassando os 6 GiB e 60% de utilização, respetivamente, em ambas as configurações, tal como observável na Tabela 9. O motivo pelo qual o nível de utilização destes recursos aumentou substancialmente no nodo em questão, prende-se com a execução, neste nodo, de uma instância do Tensorflow. De facto, observando os valores obtidos destes recursos para os nodos de armazenamento na configuração do LSFS verifica-se a manutenção de níveis semelhantes comparativamente aos do 1º conjunto de experiências. Verifica-se ainda uma maior utilização quer destes recursos quer da unidade de processamento gráfico, no nodo cliente, para o caso da configuração local, o que vai de encontro às diferenças observada no débito para as diferentes configurações (Tabela 8).

	Cliente			Nodos de Armazenamento	
	GPU(%)	CPU(%)	RAM(MiB)	CPU(%)	RAM(MiB)
LSFS	$20,80\%$	$69,38\%$	$6179,87$ MiB	$8,61\%$	$310,46$ MiB
Local	$38,13\%$	$92,19\%$	$7134,69$ MiB		

Tabela 9: Recursos utilizados pelas configurações Local e LSFS no 2º conjunto de experiências

5.3.2 3º *conjunto de experiências*

Com este conjunto de experiências pretendia-se verificar se o LSFS conseguia assegurar a persistência dos dados, tolerando e recuperando de períodos de instabilidade caracterizados pela presença de falhas dos nodos da rede. Pretendia-se, ainda, verificar o impacto que a introdução de falhas possuía no desempenho do mesmo. A estratégia adotada passou pela realização de quatro experiências de treino da rede neuronal onde se aumentou sucessivamente a percentagem de aplicação de falhas no sistema. Durante o período de execução das experiências é monitorizado o débito a que o Tensorflow processa o *dataset*, em imagens por segundo, e analisada a variação do débito perante a introdução das falhas.

Cada experiência contempla 5 iterações de introdução de falha, em intervalos de tempo de 20 em 20 minutos, no decorrer de 1 época de treino da rede neuronal. Os primeiros 10 minutos de execução constituí o período de aquecimento onde se procura manter o débito estável. Após esse período são introduzidas as primeiras falhas no sistema.

Quanto à implementação da instabilidade do sistema foi realizada através da substituição de determinados nodos da rede por outros completamente novos, pelo que cada um tem de recuperar a base de dados completa. Para os testes realizados assumiu-se um nível de instabilidade uniforme por toda a rede. A estratégia adotada passou pela seleção de nodos de forma cíclica por todos os grupos da rede, sendo que cada nodo é substituído por outro que mapeia no mesmo grupo do nodo que veio substituir. Em termos práticos, tal significa que a injeção de 10% de instabilidade no sistema corresponde à injeção de 10% de instabilidade em cada grupo.

De forma a conseguir reagir à instabilidade do sistema foi utilizada, face ao 2º conjunto de experiências, uma nova configuração para o LSFS. Ao mesmo foi introduzida redundância, nomeadamente, no que respeita o envio de pedidos do cliente para os nodos do substrato. Isto é, o *Smart Load Balancer* foi configurado para enviar cada pedido para 3 nodos distintos ao invés de um único tal como definido para as restantes experiências. Desta forma, procurou-se reduzir a probabilidade de ocorrência de *timeouts*, decorrentes do envio de pedidos do cliente para nodos que já deixaram o sistema. Contudo, uma vez que se encontram mais mensagens em circulação na rede, a sobrecarga na mesma aumenta, o que por sua vez aumenta a propensão para a perda de mensagens e, uma vez mais, para a ocorrência de *timeouts*. Neste sentido, comparativamente ao 2º conjunto de experiências, foi ainda necessário reduzir o nível de paralelismo de leitura de 96KiB para 32KiB, tratando-se de um decréscimo proporcional ao aumento do número de mensagens. A configuração em estudo neste conjunto de experiências será identificada como configuração orientada a falhas.

O primeiro passo neste estudo passou, pois, por conhecer o desempenho base oferecido por esta configuração. Para tal, executou-se uma primeira experiência de treino da rede neuronal onde não foram introduzidas falhas. Na Figura 11 são apresentados os débitos

a que o Tensorflow consegue processar o *dataset* de treino nesta configuração. Por outro lado, a Figura 10 corresponde à mesma representação gráfica, obtida para a configuração utilizada no 2º conjunto de experiências, a que nos referimos de configuração de alto desempenho. Atente-se que ambas as configurações constituem alternativas igualmente viáveis para diferentes assunções de disponibilidade da rede de nodos. Comparando estas configurações pretende-se elucidar o leitor acerca do modo como o LSFS pode ser configurado de forma a providenciar uma maior resiliência perante cenários em que se preveja uma maior instabilidade da rede e o impacto que tal acréscimo de resiliência possui no desempenho do sistema.

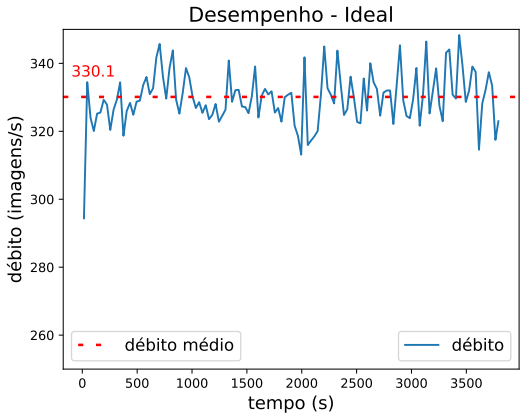


Figura 10: Configuração de alto desempenho - Débito de leitura do Tensorflow

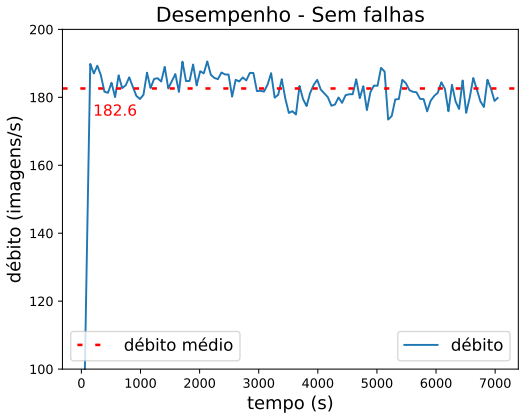


Figura 11: Configuração orientada a falhas - Débito de leitura do Tensorflow

Como é possível observar, o débito médio diminui na configuração orientada a falhas em cerca de 45%, nomeadamente, de 330,1 imagens lidas por segundo para 182,6. Este decréscimo do débito repercute-se ainda no tempo necessário à execução de 1 época de treino da rede neuronal, que demora agora cerca de 7700 segundos em vez de 4100. Estes valores vão de encontro ao que era previsto. De facto, o padrão de leituras do Tensorflow consiste na seleção aleatória de um conjunto de ficheiros do *dataset*, sobre o qual realiza leituras sequenciais, pelo que o acesso aos dados por este realizado é maioritariamente sequencial. Observando os resultados obtidos para leituras sequenciais no 1º conjunto de resultados, verifica-se também que, limitando o paralelismo de 96KiB para 32KiB, tal como acontece nesta configuração, traduz-se numa perda de aproximadamente 45% do desempenho. É de notar ainda a presença de variabilidade no débito de ambos os gráficos. Tal deve-se ao facto de nos encontrarmos num ambiente em nuvem, ou seja, que não conseguimos controlar, na medida em que é impossível prever o comportamento da rede, bem como de outros fatores. Verifica-se ainda que a variabilidade no débito se intensifica na configuração optimizada ao desempenho, diferença que pode ser facilmente justificada pela ordem de grandeza do

débito, isto é, quanto maior o débito também maior é o impacto que determinados picos de congestionamento da rede possuem no mesmo.

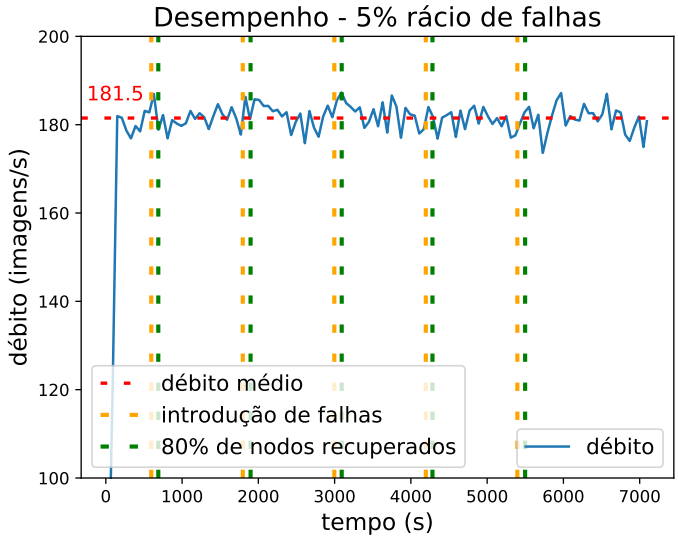


Figura 12: 5% de falhas - Débito de leitura do Tensorflow

A partir da configuração do LSFS orientada a falhas, foram executados 3 experiências que permitem avaliar a sua capacidade de superar diferentes níveis de instabilidade da rede. É de referir que cada uma das experiências parte de um mesmo estado pré-populado do *dataset*, tendo-se recorrido à realização e recuperação de instantâneos da base de dados dos nodos de forma a garantir isso mesmo. As Figuras 12, 13 e 14 mostram os gráficos de desempenho obtidos para cenários de 5%, 15% e 25% de falha, respetivamente. Nestes gráficos identificam-se, com linhas laranja a tracejado, os momentos em que se iniciam os processos de introdução de falhas, nomeadamente, em que os nodos são removidos, e, com linhas verde a tracejado, os momentos em que a maioria dos nodos (80%) que os vieram substituir se encontram devidamente incorporados no sistema, isto é, tendo recuperado as suas bases de dados face ao grupo em que se incluem.

Comparando o cenário em que se introduz 5% de falhas (Figura 12) com o que nenhuma falha é introduzida (Figura 11), verifica-se que 5% de falhas não são suficientes para afetar o funcionamento ou desempenho do LSFS. O gráfico apresenta reduzida variabilidade em torno de um débito médio de 181,5 imagens por segundo. Contrariamente, para 15% e 25% de falhas (Figuras 13 e 14), os gráficos representados refletem um padrão baseado em picos e vales de desempenho.

Contrariamente ao que seria a primeira intuição, imediatamente após a introdução das falhas ocorre um aumento do desempenho. A razão para tal acontecer relaciona-se com o grau de utilização da rede e as mensagens que nesta circulam. Durante o treino da rede neuronal o cliente submete, essencialmente, pedidos de leitura de dados, isto é, pedidos *get*,

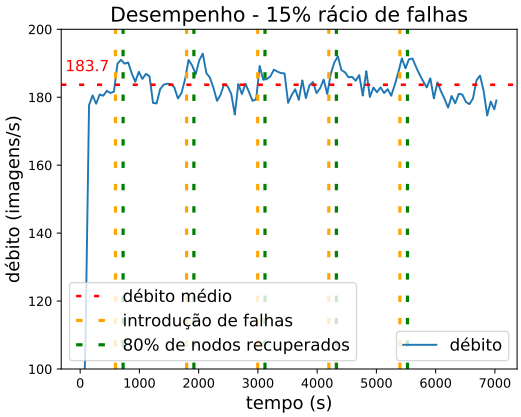


Figura 13: 15% de falhas - Débito de leitura do Tensorflow

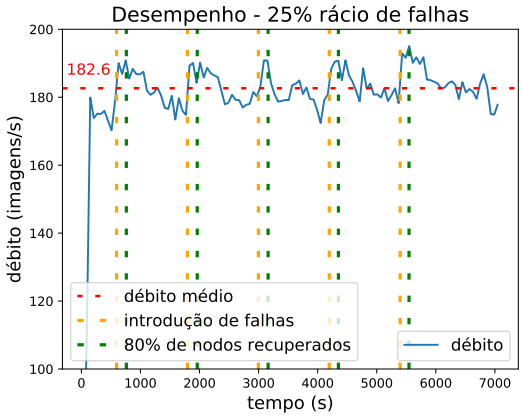


Figura 14: 25% de falhas - Débito de leitura do Tensorflow

recebendo dos vários nodos da rede os blocos associados a cada uma das chaves requisitadas. Ora, é nas mensagens de resposta que a grande maioria da informação que circula na rede se encontra, pelo que, neste cenário, é no cliente que está o gargalo de rede.

Considerando que, por motivos de redundância, o *smart load balancer* propaga cada pedido para 3 nodos distintos do grupo de replicação da chave, em condições normais também recebe de cada um uma mensagem de resposta. A injeção de falhas faz com que parte dos nodos selecionados para o envio dos pedidos, não respondam, dado que os mesmos deixaram o sistema. Ao receber menos respostas, menor a sobrecarga da rede no cliente, bem como menor o número de mensagens que o mesmo tem de processar, beneficiando, desta forma, temporariamente, o desempenho do mesmo. As Figuras 15 e 16 enquadram-se num estudo realizado ao nodo cliente, permitindo provar a veracidade desta teoria.

Na Figura 15 encontra-se representada a variação do tráfego de entrada, no nodo cliente, para cada uma das experiências em que foram introduzidas falhas. Tal como se pode observar, a introdução de falhas origina um vale no gráfico, que se torna mais profundo à medida que o nível de falha aumenta, o que vai de encontro às assunções realizadas. Quanto menor o número de mensagens de resposta recebidas, menor é o tráfego de entrada no cliente. Também a Figura 16 expressa a mesma ideia. Nesta, é comparado o número de pedidos emitidos pelo cliente relativamente ao número de respostas recebidas para o cenário de 15% de falha. Uma vez mais se verifica a formação de vales na linha referente às respostas, que, aquando da introdução das falhas, diverge da linha dos pedidos. Este gráfico, dá-nos, contudo, mais informação que permite corroborar o que se afirmou anteriormente. É visível que, para além de vales, também ocorrem picos no número de pedidos emitidos, coincidentes no tempo. Este aumento do número de pedidos traduz-se no acréscimo do desempenho observado.

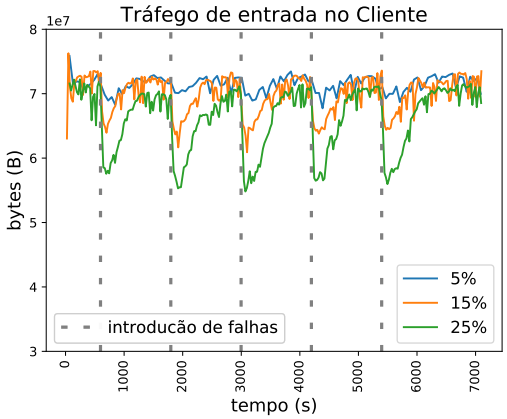


Figura 15: Tráfego de entrada no Cliente, em Bytes, para diferentes níveis de falha

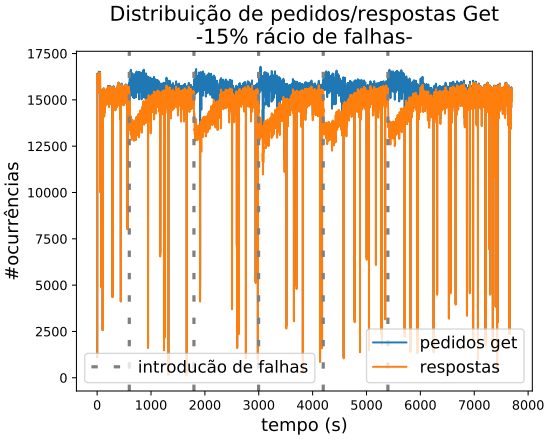


Figura 16: Distribuição de pedidos/respostas para 15% de falha dos nodos

Após cada um dos picos observados no desempenho (Figuras 13 e 14), o débito regressa de novo aos valores médios observados para as experiências onde se introduz 5% de falhas ou onde nenhuma falha é introduzida. De facto, para todas as experiências realizadas é de realçar a ocorrência de um débito médio, de cerca de 180 imagens/segundo, que se mostra muito semelhante independentemente do nível de falha aplicado, o que revela a extrema resiliência bem como disponibilidade oferecida pelo LSFS. Esta disponibilidade é facilmente justificada pelo próprio funcionamento do LSFS. Tendo em conta que existem 500 nodos, 128 grupos de replicação, e aproximadamente 4 nodos por grupo, a introdução de 25% de replicação consiste na substituição de cerca de 1 nodo por grupo. Ora, uma vez que cada novo nodo recupera a partir de apenas um outro nodo do grupo em que se inclui, restam ainda 2 nodos por grupo, em perfeitas condições de execução, para satisfazer os pedidos do cliente. Também a recuperação dos nodos ocorrendo uniformemente por toda a rede, não possui um impacto significativo na mesma.

É de referir ainda que após a execução de cada experiência, foram realizados testes de verificação da integridade dos dados armazenados no LSFS, por computação de *checksums* e comparação com os dados originais, não tendo sido detetada qualquer anomalia, isto é, não se verificou perda nem corrupção de dados.

5.3.2.1 *Recursos Utilizados*

Na Tabela 10 apresentam-se os resultados de monitorização obtidos para os testes com introdução de falhas, para a configuração do LSFS adaptada para maior resiliência. Uma vez mais se verifica uma utilização acrescida dos recursos no nodo cliente em consequência da execução do Tensorflow. Comparativamente aos valores obtidos para o 2º conjunto de experiência (Tabela 9) verifica-se, contudo, uma diminuição do nível de utilização dos recursos tanto no cliente como nos nodos de armazenamento em concordância com o

decréscimo no desempenho. No mesmo sentido, a reduzida variabilidade do débito médio observado para as experiências com os diferentes níveis de falha faz-se sentir também no consumo de recursos que se apresentam muito semelhantes(Tabela 10).

	Cliente			Nodos de Armazenamento	
	GPU(%)	CPU(%)	RAM(MiB)	CPU(%)	RAM(MiB)
LSFS (0%)	10,9%	55,7%	4014,28 MiB	6,38%	308,88 MiB
LSFS (5%)	10,31%	54,86%	3760,48 MiB	6,66%	305,25 MiB
LSFS (15%)	10,55%	55,24%	3784,37 MiB	6,86%	312,93 MiB
LSFS (25%)	10,68%	55,31%	3798,21 MiB	7,62%	314,76 MiB

Tabela 10: Recursos utilizados pelo LSFS para diferentes níveis de falha

5.3.3 *Discussão*

Os resultados apresentados mostram que o desenho do LSFS permitiu atingir os objetivos inicialmente propostos. A nível de desempenho, o LSFS oferece um débito máximo de 20,43 MiB/s para escritas, 27,53 MiB/s para leituras sequenciais e 4,5 MiB/s para leituras aleatórias, tendo estes valores sido obtidos para experiências com blocos de 32K. Verificou-se ainda que as otimizações realizadas sobre o LSFS tiveram os resultados esperados. Tanto a introdução do *smart load balancer* como a aplicação de um nível de paralelismo adequado traduziram-se em melhorias substanciais no desempenho.

Quando instalado sobre uma rede de 500 nodos, o LSFS foi apenas 2,6 vezes mais lento no treino da rede neuronal comparativamente ao observado para uma configuração local. Trata-se este de um resultado notável tendo em conta a presença da rede no sistema, e o grau de distribuição que esta configuração apresenta. Provou-se, ainda, uma grande escalabilidade exibida por parte do LSFS.

As experiências realizadas permitiram também mostrar que o LSFS pode ser facilmente configurado para cenários de elevada propensão de falhas. Este exibiu uma enorme resiliência, tendo mantido um desempenho estável e em condições plenas de funcionamento perante níveis de falha de até 25% do número total de nodos presentes na sua rede.

6

CONCLUSÕES

Esta dissertação apresenta o LSFS, um sistema de ficheiros completamente distribuído, que tira proveito do espaço de armazenamento de centenas a milhares de dispositivos presentes em redes de larga escala. Para este tipo de cenários, onde a falha é a regra e não a exceção, o seu desenho permite oferecer garantias de alta disponibilidade e resiliência forte.

Tendo-se analisado o estado da arte concluiu-se que nenhum outro sistema de ficheiros era capaz de oferecer as garantias de escalabilidade e resiliência procuradas por esta dissertação, isto é, que permitisse, ao mesmo tempo, escalar para redes de grande dimensão e tolerar elevados níveis de instabilidade da rede. O sistema que se destacou pela positiva neste domínio foi o DataFlasks, que, tratando-se de uma base de dados chave-valor descentralizada, diferencia-se das restantes ao adotar um configuração de rede aleatória bem como o uso de algoritmos de natureza epidémica. Estes algoritmos constituem o motivo da resiliência e escalabilidade que o caracteriza.

No sentido de alcançar as mesmas propriedades, o LSFS apresenta uma arquitetura que tem por base o DataFlasks como único substrato de armazenamento. Neste, são armazenados quer os dados como os metadados do sistema de ficheiros, sendo que tal constitui uma condição necessária à escalabilidade, já adotada por alguns sistemas de ficheiros do estado da arte.

De forma a providenciar um sistema coerente e utilizável em conjugação com garantias de consistência eventual, algumas decisões foram tomadas no LSFS. Foi adotado um modelo *write once read many* que, satisfazendo os requisitos de utilização objetivados, evitou problemas decorrentes da atualização simultânea de ficheiros. Já no que respeita a atualização de diretorias utilizou-se uma estratégia baseada na realização de junção de metadados, mantendo-se uma visão sucessivamente crescente das mesmas, isto é, foi abolida a possibilidade de remoção de ficheiros, e agregadas diferentes versões no tempo das mesmas diretorias. Este modelo permitiu garantir coerência do sistema de ficheiros, mostrando-se válido para o tipo de casos de uso projetados nesta dissertação, nomeadamente, aplicações de larga escala, por exemplo de IoT, que agregam e processam dados produzidos por múltiplos dispositivos em rede.

Para além de garantias de coerência, resiliência e disponibilidade, também o desempenho impunha-se como uma medida fundamental de usabilidade. No LSFS, o desempenho prendia-se essencialmente em conseguir tirar a máxima vantagem da largura de banda disponível. Neste sentido destacam-se duas contribuições que permitiram acelerar o processo de satisfação dos pedidos. Uma destas consistiu na introdução de paralelismo ao nível dos pedidos, isto é, pedidos grandes são divididos noutros de menores dimensões que são propagados na rede em simultâneo em detrimento de uma execução sequencial. A segunda foi aplicada ao DataFlasks, nomeadamente, foi desenvolvido um balanceador de pedidos inteligente que realiza uma aprendizagem constante da rede de forma a tomar decisões mais precisas para que nodos encaminhar os pedidos. Desta forma, procurando-se reduzir o número de *hops* necessários à sua satisfação.

Uma grande contribuição desta dissertação incidiu na elaboração de um protótipo do LSFS bem como uma extensa componente de testes que permitiram a sua avaliação. O protótipo foi sujeito quer a micro cargas de stress, de escrita e leitura, como também foi avaliado perante um cenário real de treino de uma rede neuronal com recurso ao Tensorflow, sendo que, neste último caso, recorreu-se a uma configuração do LSFS com 500 nodos. Em ambos os cenários o LSFS exibiu um bom desempenho face ao grau de distribuição que apresenta, o que veio provar também um boa escalabilidade por parte do mesmo. Para além destas experiências, o comportamento do LSFS foi ainda analisado perante cenários de falha dos seus nodos constituintes. Para este caso, o LSFS foi configurado para oferecer uma maior resiliência a troco de uma redução de desempenho, isto é, aumentou-se a redundância no envio de pedidos pelo cliente, contrabalançando o aumento de mensagens na rede com uma redução no nível de paralelismo adotado. O LSFS mostrou-se resiliente a níveis de falha de carácter catastrófico, tendo recuperado com sucesso, sem penalizações de desempenho, falhas de até 25% da totalidade dos seus nodos.

6.1 TRABALHO FUTURO

O trabalho realizado nesta dissertação abriu também importantes contribuições ciêntificas como trabalho futuro.

Diversos fatores bem como parâmetros descritos neste trabalho afetam a resiliência bem como o desempenho e escalabilidade oferecida pelo LSFS, tratando-se, pois, de um compromisso que é necessário estabelecer. Um maior nível de confiança na rede permite-nos estabelecer, por exemplo, um maior grau de paralelismo das operações, enquanto que a expectativa de uma rede mais instável leva-nos a privilegiar a resiliência pela introdução de redundância. A razão pela qual tal constitui um compromisso deve-se ao facto da rede se apresentar como um fator limitante, isto é, a largura de banda é, por si só, um recurso limitado e variável, sendo impossível encontrar uma configuração do LSFS que seja ideal

para todos os cenários. O que se motiva aqui é a aplicação de processos de aprendizagem máquina com intuito de dinamicamente se selecionar a melhor configuração possível. Como variável de rede poderia-mo-nos basear na periodicidade de ocorrência de *timeouts* na satisfação de pedidos.

Ainda no que respeita o desempenho do LSFS destaca-se o balanceador de carga utilizado no DataFlasks como lugar propício a novas otimizações. Nesta dissertação provou-se que é possível manipulando o algoritmo de seleção de nodos para envio de pedidos obter melhorias de desempenho significativas. Este estudo encontra-se, contudo, na sua fase inicial sendo possível pela introdução de novo conhecimento desenvolver implementações ainda mais otimizadas. Por exemplo, à medida que os pedidos são satisfeitos seria possível realizar um estudo contínuo acerca dos nodos que demoram menos tempo a enviar respostas ao cliente, desta forma, dá-nos uma ideia de que nodos se encontram mais ou menos saturados, sendo que esse conhecimento pode ser utilizado para privilegiar o envio de pedidos para nodos menos saturados.

BIBLIOGRAFIA

[Abadi et al., 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283.

[Altair SmartWorks,] Altair SmartWorks. https://www.altairsmartworks.com/.

[Anderson et al., 2002] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. (2002). Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61.

[Bigger,] Bigger, J. Reduce Your On-Site Computer Hardware Costs with Cloud Computing . https://www.marconet.com/blog/reduce-your-onsite-computing-hardware-costs-with-cloud-computing. Accessed: 2020-01-09.

[Boost,] Boost. https://www.boost.org.

[Borthakur et al., 2008] Borthakur, D. et al. (2008). Hdfs architecture guide. *Hadoop Apache Project*, 53(1-13):2.

[Brewer, 2012] Brewer, E. (2012). Pushing the cap: Strategies for consistency and availability. *Computer*, 45(2):23–29.

[Carvalho et al., 2011] Carvalho, N. A., Bordalo, J., Campos, F., and Pereira, J. (2011). Experimental evaluation of distributed middleware with a virtualized java environment. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, page 3. ACM.

[CassFS,] CassFS. https://github.com/jdarcy/CassFS.

[Dabek et al., 2001] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. (2001). Wide-area cooperative storage with cfs. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 202–215. ACM.

[Dabek et al., 2004] Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, M. F., and Morris, R. T. (2004). Designing a dht for low latency and high throughput. In *NSDI*, volume 4, pages 85–98.

[DeCandia et al., 2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM.

[Delaney, 2000] Delaney, K. (2000). *Inside Microsoft SQL Server 2000*. Microsoft Press.

[Demers et al., 1988] Demers, A., Greene, D., Houser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. (1988). Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Systems Review*, 22(1):8–32.

[Druschel and Rowstron, 2001] Druschel, P. and Rowstron, A. (2001). Past: A large-scale, persistent peer-to-peer storage utility. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 75–80. IEEE.

[DynamoFS,] DynamoFS. https://github.com/denismo/DynamoFS.

[Erdős and Rényi, 1960] Erdős, P. and Rényi, A. (1960). On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60.

[Eugster et al., 2004] Eugster, P., Guerraoui, R., Kermarrec, A.-M., and Massoulié, L. (2004). From epidemics to distributed computing. 37.

[Eugster et al., 2003] Eugster, P. T., Guerraoui, R., Handurukande, S. B., Kouznetsov, P., and Kermarrec, A.-M. (2003). Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374.

[Fernández et al., 2007] Fernández, A., Gramoli, V., Jiménez, E., Kermarrec, A.-M., and Raynal, M. (2007). Distributed slicing in dynamic systems. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pages 66–66. IEEE.

[Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system.

[Google,] Google. Block storage performance. https://cloud.google.com/compute/docs/disks/performance.

[Gramoli et al., 2008] Gramoli, V., Vigfusson, Y., Birman, K., Kermarrec, A.-M., and van Renesse, R. (2008). A fast distributed slicing algorithm. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 427–427.

[Gramoli et al., 2009] Gramoli, V., Vigfusson, Y., Birman, K., Kermarrec, A.-M., and Van Renesse, R. (2009). Slicing distributed systems. *IEEE Transactions on Computers*, 58(11):1444–1455.

[Grove Streams,] Grove Streams. https://www.grovestreams.com/.

[Gupta et al., 2014] Gupta, T., Singh, R. P., Phanishayee, A., Jung, J., and Mahajan, R. (2014). Bolt: Data management for connected homes. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 243–256, Seattle, WA. USENIX Association.

[Guy et al., 1990] Guy, R. G., Heidemann, J. S., Mak, W.-K., Page Jr, T. W., Popek, G. J., Rothmeier, D., et al. (1990). Implementation of the ficus replicated file system. In *USENIX Summer*, pages 63–72.

[Howard et al., 1988] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. (1988). Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81.

[IDC,] IDC, I. D. C. The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast . https://www.idc.com/getdoc.jsp?containerId=prUS45213219. Accessed: 2019-11-19.

[IEEE,] IEEE. 1003.1-2017 - ieee standard for information technology–portable operating system interface (posix(r)) base specifications, issue 7. https://www.ieee.org/.

[Jelasity et al., 2005] Jelasity, M., Montresor, A., and Babaoglu, O. (2005). Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):219–252.

[Kubernetes,] Kubernetes. https://kubernetes.io.

[Lakshman and Malik, 2010] Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

[Leitao et al., 2010] Leitao, J., Pereira, J., and Rodrigues, L. (2010). Gossip-based broadcast. In *Handbook of Peer-to-Peer Networking*, pages 831–860. Springer.

[LevelDB,] LevelDB. https://github.com/google/leveldb.

[Maia et al., 2014] Maia, F., Matos, M., Vilaça, R., Pereira, J., Oliveira, R., and Riviere, E. (2014). Dataflasks: epidemic store for massive scale systems. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 79–88. IEEE.

[Mathur et al., 2007] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. (2007). The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33.

[Maymounkov and Mazieres, 2002] Maymounkov, P. and Mazieres, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer.

[Mockapetris and Dunlap, 1988] Mockapetris, P. and Dunlap, K. J. (1988). *Development of the domain name system*, volume 18. ACM.

[Muthitacharoen et al., 2002] Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B. (2002). Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44.

[MySQL,] MySQL. https://mysql.com.

[Napster,] Napster. https://napster.com/.

[Osadzinski, 1988] Osadzinski, A. (1988). The network file system (nfs). *Computer Standards & Interfaces*, 8(1):45–48.

[Picconi et al., 2005] Picconi, F., Sens, P., et al. (2005). Pastis: A highly-scalable multi-user peer-to-peer file system. In *European Conference on Parallel Processing*, pages 1173–1182. Springer.

[Popek et al., 1990] Popek, G. J., Guy, R. G., Page, T. W., and Heidemann, J. S. (1990). Replication in ficus distributed file systems. In *[1990] Proceedings. Workshop on the Management of Replicated Data*, pages 5–10. IEEE.

[PostgreSQL,] PostgreSQL. https://postgresql.org/.

[Protobuf,] Protobuf. https://developers.google.com/protocol-buffers.

[Ratnasamy et al., 2001] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. (2001). *A scalable content-addressable network*, volume 31. ACM.

[Rhea et al., 2004] Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. (2004). Handling churn in a dht.

[Riak,] Riak. https://riak.com/products/riak-kv/.

[Ripeanu, 2001] Ripeanu, M. (2001). Peer-to-peer architecture case study: Gnutella network. In *Proceedings first international conference on peer-to-peer computing*, pages 99–100. IEEE.

[Riviere and Voulgaris, 2011] Riviere, E. and Voulgaris, S. (2011). Gossip-based networking for internet-scale distributed systems. In *International Conference on E-Technologies*, pages 253–284. Springer.

[RocksDB,] RocksDB. http://rocksdb.org/.

[Rowstron and Druschel, 2001] Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer.

[Sandberg et al., 1985] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. (1985). Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130.

[Sarkar, 2019] Sarkar, S. (2019). A scalable artificial intelligence data pipeline for accelerating time to insight. In *2019 Storage Developer Conference (SDC19)*. Western Digital.

[Shapiro et al., 2011] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011). Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer.

[Singla and Rohrs, 2002] Singla, A. and Rohrs, C. (2002). Ultrapeers: Another step towards gnutella scalability .

[Sittón-Candanedo et al., 2019] Sittón-Candanedo, I., Alonso, R. S., García, Ó., Muñoz, L., and Rodríguez-González, S. (2019). Edge computing, iot and social computing in smart energy scenarios. *Sensors*, 19(15):3353.

[Stoica et al., 2001] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160.

[Szeredi,] Szeredi, M. File system in user space. https://github.com/libfuse/libfuse.

[Taher et al., 2019] Taher, N. C., Mallat, I., Agoulmine, N., and El-Mawass, N. (2019). An iot-cloud based solution for real-time and batch processing of big data: Application in healthcare. In *2019 3rd International Conference on Bio-engineering for Smart Technologies (BioSMART)*, pages 1–8. IEEE.

[Tarasov et al., 2015] Tarasov, V., Gupta, A., Sourav, K., Trehan, S., and Zadok, E. (2015). Terra incognita: On the practicality of user-space file systems. In *7th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*.

[Tarasov et al., 2016] Tarasov, V., Zadok, E., and Shepler, S. (2016). Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12.

[Tarkoma et al., 2011] Tarkoma, S., Rothenberg, C. E., and Lagerspetz, E. (2011). Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155.

[Terry et al., 1995] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. (1995). Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, volume 95, pages 172–182.

[Van Renesse et al., 2003] Van Renesse, R., Birman, K. P., and Vogels, W. (2003). Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM transactions on computer systems (TOCS)*, 21(2):164–206.

[Van Renesse et al., 1998] Van Renesse, R., Minsky, Y., and Hayden, M. (1998). A gossip-style failure detection service. In *Middleware'98*, pages 55–70. Springer.

[Vangoor et al., 2017] Vangoor, B. K. R., Tarasov, V., and Zadok, E. (2017). To {FUSE} or not to {FUSE}: Performance of user-space file systems. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 59–72.

[Voulgaris et al., 2005] Voulgaris, S., Gavidia, D., and Van Steen, M. (2005). Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and systems Management*, 13(2):197–217.

[Voulgaris et al., 2003] Voulgaris, S., Jelasity, M., and Van Steen, M. (2003). A robust and scalable peer-to-peer gossiping protocol. In *International Workshop on Agents and P2P Computing*, pages 47–58. Springer.

[Weikum and Vossen, 2001] Weikum, G. and Vossen, G. (2001). *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier.

[Weil et al., 2006] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., and Maltzahn, C. (2006). Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association.

[Xively,] Xively. https://xively.com/.

[Zhang et al., 2015] Zhang, B., Mor, N., Kolb, J., Chan, D. S., Lutz, K., Allman, E., Wawrzynek, J., Lee, E., and Kubiatowicz, J. (2015). The cloud is not enough: Saving iot from the cloud. In *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*.

[Zhao et al., 2004] Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., and Kubiatowicz, J. D. (2004). Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53.

A

APÊNDICES

A.1 APÊNDICE 1 - RESULTADOS DE MONITORIZAÇÃO DE OPERAÇÕES DE LEITURA (1º CONJUNTO DE EXPERIÊNCIAS)

			Cliente		Nodos de Armaz.	
			CPU(%)	RAM(MiB)	CPU(%)	RAM(MiB)
Consumo de leituras	leitura-seq-4k	random-4k	10,30%	413,45 MiB	43,29%	310,85 MiB
		random-8k	8,09%	419,19 MiB	44,31%	314,98 MiB
		random-16k	6,09%	403,23 MiB	31,08%	310,38 MiB
		smart-64k	22,54%	406,91 MiB	17,73%	311,85 MiB
		smart-96k	24,3%	409,17 MiB	20,63%	312,87 MiB
		smart-128k	17,4%	408,8 MiB	15,97%	312,20 MiB
	leitura-seq-32k	random-4k	11,55%	408,8 MiB	45,21%	306,86 MiB
		random-8k	9,28%	403,14 MiB	43,05%	306,88 MiB
		random-16k	7,82%	404,88 MiB	33,34%	307,23 MiB
		smart-64k	22,23%	406,23 MiB	16,72%	304,88 MiB
		smart-96k	24,46%	407,54 MiB	18,27%	305,78 MiB
		smart-128k	16,93%	405,89 MiB	16,81%	311,93 MiB
	leitura-aleat-4k	random-4k	9,36%	407,22 MiB	6,86%	305,85 MiB
		smart-4k	4,98%	408,08 MiB	6,44%	313,31 MiB
	leitura-aleat-32k	random-4k	10,1%	408,15 MiB	40,29%	306,21 MiB
		random-8k	8,99%	406,14%	43,33%	307,45 MiB
		random-16k	9,04%	404,5 MiB	36,58%	308,02 MiB
		smart-16k	6,65%	403,61 MiB	7,37%	306,11 MiB
		smart-32k	5,78%	408,70 MiB	8,37%	313,62 MiB