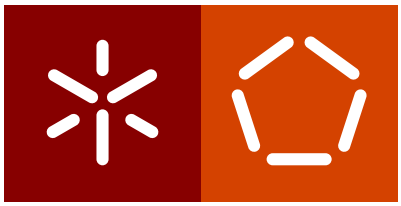
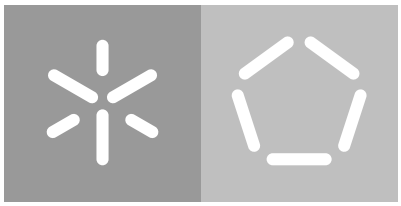


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Diogo Luzio Leitão

RSafeFS: Sistema de Ficheiros Modular para Armazenamento Remoto

Fevereiro 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Diogo Luzio Leitão

RSafeFS: Sistema de Ficheiros Modular para Armazenamento Remoto

Dissertação de mestrado
Mestrado Integrado em Engenharia Informática

Dissertação orientada por
João Tiago Medeiros Paulo
José Orlando Pereira

Fevereiro 2021

AGRADECIMENTOS

A realização desta dissertação não seria possível sem a ajuda de várias pessoas, as quais eu gostaria de expressar a minha gratidão.

Ao meu orientador, Professor Doutor João Paulo, por me ter guiado e motivado durante todo este processo, e por ter estado disponível sempre que foi necessário. Ao meu co-orientador, Professor Doutor José Orlando Pereira, por tornar a realização desta dissertação possível.

Ao Ricardo Macedo, por toda a ajuda na prestada na implementação do protótipo, na análise de resultados, e por fim na escrita da dissertação. A todo o Grupo de Sistemas Distribuídos do HASLab, pela forma como me acolheram, e pela disponibilidade demonstrada em ajudar.

Por último, dirijo um agradecimento especial a toda a minha família pelo apoio incondicional durante todo o meu percurso académico.

ABSTRACT

File systems are widely used for storing digital information, as they offer abstractions that allow data to be intuitively separated and organized through files and directories, according to the requirements of applications and users. The continuous growth of data volume and complexity leads to the constant evolution of these systems. However, the complexity of integration of new features and lack of continuous support, leads to many file systems not being adopted in practice.

In this sense, stackable file systems have emerged, which allow the development of complex file systems, providing existing systems with new functionalities through independent processing layers. Despite this, the development of these systems presents some challenges, namely in terms of speed of implementation, portability, and resilience, since they are developed in kernel. In this way, later solutions emerged that allowed the development of file systems in user space, thus mitigating some of the problems identified in the development of this type of file systems. However, these solutions have not been properly explored in the development of remote file systems.

Therefore, this dissertation presents RSafeFS, a platform that extends the SafeFS system to allow developing modular, flexible and extensible remote file systems in user space. The proposed solution enables extensible remote file system implementations that adjust to the requirements of different types of applications and storage workloads. It was then necessary to develop a layer that would allow an RSafeFS instance to operate as a system server, and a communication layer, based on remote procedure calls (RPCs), to allow interoperability between client and server instances. To demonstrate the ease of integration of new features, taking advantage of the modularity and flexibility of RSafeFS, the developed prototype was equipped with two layers of caching, namely data and metadata, which aim to improve system peformance. The results obtained with this prototype reveal that the file systems developed through RSafeFS obtain performances comparable to remote storage solutions based on FUSE. Furthermore, with the processing layers developed it is possible to adjust the system to different types of workloads, allowing, for example, to improve system performance by 1.5× in certain workloads.

Keywords: storage, remote, modular, flexible, extensible

RESUMO

Os sistemas de ficheiros são atualmente uma das soluções mais utilizadas para o armazenamento de informação digital, pois oferecem abstrações que permitem separar e organizar de forma intuitiva os dados através ficheiros e diretorias, segundo os requisitos das aplicações e dos utilizadores. O contínuo crescimento do volume e complexidade de dados leva à constante evolução destas soluções. Contudo, a complexidade de integração de novas funcionalidades e falta de suporte contínuo, leva a que muitos dos sistemas de ficheiros desenvolvidos não sejam adotados.

Neste sentido, surgiram os sistemas de ficheiros empilháveis, que permitem desenvolver sistemas de ficheiros complexos, dotando sistemas já existentes com novas funcionalidades através de camadas de processamento independentes. Apesar disto, o desenvolvimento destes sistemas apresenta alguns desafios, nomeadamente a nível da rapidez de implementação, portabilidade, e resiliência, uma vez que são desenvolvidos ao nível do *kernel*. Desta forma, mais tarde, surgiram soluções que permitiram desenvolver sistemas de ficheiros em espaço de utilizador, mitigando assim alguns dos problemas identificados no desenvolvimento deste tipo sistemas de ficheiros. Contudo, estas soluções não têm sido devidamente exploradas no desenvolvimento de sistemas de ficheiros remotos.

Esta dissertação apresenta o RSafeFS, uma plataforma que estende o sistema SafeFS para permitir desenvolver sistemas de ficheiros remotos modulares, flexíveis e extensíveis em espaço de utilizador. Foi então necessário desenvolver uma camada que permitisse a uma instância RSafeFS operar como um servidor do sistema, e meios de comunicação, baseados em protocolos remotos (RPCs), para permitir a interoperabilidade entre instâncias cliente e servidor. Desta forma, a solução proposta permite desenvolver soluções de armazenamento remotas extensíveis e adaptáveis a requisitos de diferentes tipos de aplicações e cargas de trabalho. Para demonstrar a facilidade de integração de novas funcionalidades, tirando partido da modularidade e flexibilidade do RSafeFS, o protótipo desenvolvido foi dotado com duas camadas de *caching*, nomeadamente de dados e metadados, que procuram melhorar o desempenho do sistema. Os resultados obtidos com este protótipo, revelam que os sistemas de ficheiros desenvolvidos através do RSafeFS obtêm desempenhos comparáveis com os de soluções de armazenamento remotos baseadas em FUSE. Ainda, com as camadas de processamento desenvolvidas é possível ajustar o sistema a diferentes tipos de cargas de trabalho, permitindo, por exemplo, melhorar o desempenho do sistema em $1.5\times$ em determinadas cargas de trabalho.

Palavras-chave: armazenamento, remoto, modular, flexível, extensível

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.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE EXCERTOS DE CÓDIGO

LISTA DE ABREVIATURAS

CIFS Common Internet File System. 18

DNS Domain Name System. 18
DOS Disk Operating System. 17

FIFO First In First Out. 14
FUSE Filesystem in Userspace. 8

IDL Interface Definition Language. 37

LAN Local Area Network. 18, 19
LRU Least Recently Used. 14

MRU Most Recently Used. 14

NetBIOS Network Basic Input/Output System. 18
NFS Network File System. 16
NLM Network Lock Manager. 16, 17
NVMe Non-Volatile Memory. 50

ONC RPC Open Network Computing Remote Procedure Call. 23

POSIX Portable Operating System Interface. 1

RPC Remote Procedure Calls. 3, 16, 23, 24, 34, 36, 37
RTT Round Trip Time. 50

SMB Server Message Block. 17–19
SSD Solid State Drive. 50

TCP Transmission Control Protocol. 24

UDP User Datagram Protocol. 24

VFS Virtual File System. 11, 21, 22, 27, 32, 33

XDR External Data Representation. 24

1

INTRODUÇÃO

Com a expansão da era digital, cada vez mais é necessário encontrar soluções práticas e eficientes de forma a garantir a persistência e rápido acesso a grandes conjuntos de dados. Por exemplo, as empresas, hoje em dia, estão a alavancar os seus negócios na análise de informação para melhorarem a experiência dos clientes, abrir novos mercados e aumentar a produtividade dos trabalhadores e processos. Com o evoluir dos anos, o volume de dados a armazenar, bem como a sua complexidade, cresceram exponencialmente, e tem tendência a continuar. Um estudo recentemente realizado pela *International Data Corporation* prevê que até 2025 a quantidade de dados gerado a nível global aumentará para 175 Zettabytes face aos 33 ZB gerados em 2018 [58].

Uma das abstrações mais utilizadas para armazenar informação é o ficheiro, pois permite separar e organizar intuitivamente os dados segundo as necessidades dos utilizadores e aplicações. Esta abstração, que é fornecida sobre dispositivos de armazenamento orientados ao bloco, é possível através da utilização de sistemas de ficheiros. Para além da organização de dados em ficheiros, estes sistemas oferecem também a noção de diretorias, que permitem aos utilizadores agruparem vários ficheiros. As diretorias podem também permitir hierarquias, ou seja, as diretorias podem conter subdiretorias, permitindo assim estruturas mais complexas na organização da informação.

Um utilizador de um sistema de ficheiros, interage com o sistema através de operações definidas pela interface Portable Operating System Interface (POSIX) [75] - família de *standards* que mantém a compatibilidade entre diferentes sistemas operativos e aplicações. Nesta interface, entre várias operações, estão definidas as operações que permitem abrir e fechar ficheiros, ler e modificar dados de ficheiros, gerir diretorias e atualizar metadados. Estas operações são depois recebidas e realizadas pelo sistema de ficheiros. Ao nível do sistema operativo, cada ficheiro é composto por dados e metadados, e os pedidos sobre os ficheiros originam acessos aos metadados e a diferentes blocos de dados.

Atualmente, existem vários sistemas de ficheiros locais (p.e., Ext4, XFS, ZFS, Btrfs [48, 68, 30, 59]), cada um com diferentes características e formas de organizar os seus dados. Por isso, alguns sistemas de ficheiros têm melhor desempenho do que outros, alguns oferecem mais garantias de segurança, outros oferecem maior proteção contra a corrupção de dados

e outros têm melhor escalabilidade e suporte para uma capacidade de armazenamento superior.

Os sistemas de armazenamento local têm como desvantagem não possibilitar o acesso e armazenamento de dados por clientes remotos, limitando assim a partilha de informação entre diferentes utilizadores. Estas limitações levaram ao desenvolvimento de sistemas de ficheiros remotos (p.e., NFS, Samba [61, 25]) que permitem aos utilizadores acederem a ficheiros armazenados remotamente através da rede. A arquitetura destes sistemas é dividida em duas partes: o cliente e o servidor. O servidor armazena toda a informação de todos os clientes. O cliente vê o sistema de ficheiros exportado pelo servidor como um sistema local e consegue operar sobre os ficheiros remotos através da rede.

De notar que todos os sistemas de ficheiros referidos anteriormente respeitam a interface POSIX, tornando-se possível a utilização transparente dos mesmos por parte de diferentes aplicações, ou seja, as aplicações não precisam de ser modificadas para utilizarem diferentes sistemas de ficheiros. Ainda, de referir que isto é verdade tanto para sistemas de ficheiros locais como para sistemas de ficheiros remotos.

1.1 PROBLEMA E OBJETIVOS

A maioria dos sistemas de ficheiros atuais são implementados em *kernel-space* o que dificulta consideravelmente a adição de novas funcionalidades (p.e., *caching*, cifragem, compressão, deduplicação). De facto, é reconhecido pela literatura que a implementação destas funcionalidades em *user-space* é cada vez mais desejável por questões de rapidez de implementação, acesso a bibliotecas de alto-nível, e até mesmo de resiliência [69]. Estas razões motivaram o aparecimento de plataformas para desenvolver sistemas de ficheiros POSIX em *user-space*, sendo o FUSE [18], a plataforma mais utilizada atualmente [74].

No entanto, quer as implementações tradicionais quer as baseadas na plataforma FUSE são tipicamente monolíticas, limitando assim, uma vez mais, o desenvolvimento e composição de novas funcionalidades de armazenamento que permitam uma resposta mais eficiente aos requisitos de diferentes tipos de aplicações e cargas de trabalho.

De forma a ultrapassar este desafio, o estado da arte atual tem-se focado na proposta de novas soluções, baseadas em *user-space*, que permitem agilizar o desenvolvimento de funcionalidades de armazenamento. Em particular, o SafeFS [54] é um sistema de ficheiros baseado em FUSE, que apresenta uma arquitetura modular e flexível. O desenho do mesmo pretende simplificar e acelerar o processo de implementação de novas funcionalidades de armazenamento e a composição das mesmas. Assim, este sistema permite que utilizadores personalizem o seu armazenamento de dados segundo os diferentes objetivos que pretendem atingir (p.e., cifrar, deduplicar, comprimir), escolhendo funcionalidades de armazenamento que ofereçam as funcionalidades desejadas pelas suas aplicações. Contudo, o SafeFS está

desenhado para gerir e armazenar dados localmente, impossibilitando assim a sua adoção para cenários de armazenamento remoto.

Esta dissertação tem assim como objetivo redesenhar o sistema de ficheiros SafeFS, de forma a estender a sua flexibilidade e modularidade para sistemas de armazenamento remoto. Ao alavancar os princípios do SafeFS para um ambiente remoto, torna possível a utilização e implementação de novas funcionalidades sobre os dados (p.e., cifragem, compressão, *caching*) de acordo com diferentes objetivos e requisitos identificados pelas aplicações e administradores de sistemas (p.e., reduzir espaço de armazenamento, proteger informação sensível, maior desempenho).

1.2 CONTRIBUIÇÕES

De forma a concretizar o objetivo proposto, esta dissertação apresenta um conjunto de contribuições distribuídas em três fases: desenho de uma arquitetura de um sistema de ficheiros remoto modular e flexível; desenvolvimento de um protótipo deste novo sistema; e avaliação extensiva do protótipo sobre diferentes cargas de trabalho. Em maior detalhe:

- Como primeira contribuição, esta dissertação propõe um novo desenho de uma arquitetura modular, flexível, e extensível para um sistema de ficheiros remoto. Este sistema denomina-se RSafeFS, e avança o estado da arte ao apresentar uma plataforma que permite desenvolver sistemas de ficheiros modulares e flexíveis com capacidades de armazenamento remoto. Comparativamente com os sistemas de ficheiros remotos existentes, os quais são implementados em *kernel-space*, este desenho vem colmatar a dificuldade de implementação, integração e combinação de novas funcionalidades de armazenamento.

- Como segunda contribuição, é apresentado um protótipo do sistema RSafeFS. O protótipo do RSafeFS é dividido em duas partes - cliente e servidor -, que individualmente seguem o desenho original do SafeFS. Como novidade, esta dissertação propõe uma nova camada de comunicação baseada em protocolos remotos - Remote Procedure Calls (RPC). Esta camada de comunicação pode implementar diferentes protocolos de comunicação, e utilizar diferentes sistemas de comunicação. Por exemplo, o protótipo apresentado suporta as plataformas gRPC [15] e Cap'n Proto [3]. Ainda, de forma a melhorar o desempenho do protótipo e mostrar a sua modularidade, são desenvolvidas novas camadas de funcionalidade de armazenamento (*caches* de dados e metadados), que procuram reduzir o número de operações enviadas para o dispositivo de armazenamento físico.

- Como terceira contribuição, é realizada uma avaliação experimental detalhada do protótipo do sistema RSafeFS. A mesma compreende a análise do desempenho e dos

recursos utilizados pelo sistema, quando sujeito a testes micro e macro. Nos testes micro é avaliado o comportamento do sistema sobre operações isoladas sobre dados e metadados. Nos testes macro é avaliado o desempenho do sistema sobre cenários mais realistas, sendo compostos por conjuntos de operações (sobre dados e metadados) e diferentes padrões de acesso. De forma a tornar a avaliação compreensiva, são também contemplados outros sistemas de armazenamento remoto, nomeadamente o NFS e FUSE-NFS [11], que servem de comparação com o RSafeFS. Os resultados demonstram que os desempenhos dos sistemas de ficheiros desenvolvidos através do RSafeFS são comparáveis com os de soluções existentes baseadas em FUSE, para uma grande maioria das cargas experimentais complementadas, e que podem ser melhorados através de camadas de processamento.

1.3 Estrutura da dissertação

A dissertação está organizada da seguinte forma. No capítulo 2 é realizado um levantamento do estado da arte atual sobre sistemas de ficheiros modulares e sistemas de armazenamento remoto. No capítulo 3 é apresentada a arquitetura do RSafeFS, descrevendo o papel de todos os componentes, bem como a forma como estes permitem dotar o sistema SafeFS com capacidades de armazenamento remoto. O capítulo 4 descreve a implementação dos componentes principais da arquitetura proposta, bem como das novas funcionalidades de armazenamento utilizadas para melhorar o seu desempenho. No capítulo 5 é feita uma avaliação completa e detalhada do protótipo implementado, onde é analisado o seu desempenho quando sujeito a diferentes cargas de trabalho. Por fim, no capítulo 6 são feitas as reflexões finais sobre o trabalho realizado e são discutidos pontos de trabalho futuro.

2

TRABALHO RELACIONADO

Nesta dissertação, apresentamos o RSafeFS, um sistema de ficheiros modular remoto. Desta forma, neste capítulo é feito um levantamento dos principais sistemas de ficheiros modulares (secção 2.1) e remotos (secção 2.2), sendo também realizada uma descrição detalhada da sua arquitetura e funcionalidade.

2.1 SISTEMAS DE FICHEIROS MODULARES

Para responder ao contínuo crescimento do volume e complexidade dos dados, os sistemas de ficheiros estão em constante evolução. Por exemplo, para reduzir o espaço de armazenamento, sistemas de ficheiros como Btrfs e ZFS fazem compressão de dados [59, 30]. Contudo, a adoção de alguns destes sistemas de ficheiros é impraticável, dada a complexidade de integração de novas funcionalidades e falta de suporte contínuo. Por exemplo, o StegFS [27, 49] é um sistema de ficheiros que estende o Ext2 [33] para oferecer funcionalidades de criptografia e esteganografia sobre os dados. No entanto, devido a erros de implementação e com o desenvolvimento do sistema de ficheiros Ext3 [73], este sistema foi descontinuado.

Como solução para estes problemas, surgem os sistemas de ficheiros empilháveis, que permitem construir sistemas de ficheiros complexos através de várias camadas de processamento de dados [41]. Os sistemas de ficheiros empilháveis são sistemas que não armazenam eles próprios os dados, em vez disso utilizam outros sistemas ficheiros (p.e., Ext4, NFS) para garantir esse armazenamento [66]. Neste tipo de sistemas, o sistema de ficheiros empilhável é designado de sistema de ficheiros superior, e o sistema de ficheiros utilizado para persistir os dados de sistema de ficheiros inferior.

No sistema de ficheiros superior, cada camada compreende um mecanismo de processamento isolado a aplicar sobre os dados (p.e., compressão, cifragem). Ao encapsular cada um dos mecanismos de forma isolada, este tipo de sistemas permite a construção de novas funcionalidades para responder aos requisitos das várias aplicações. Cada camada é empilhada em cima de outra, seguindo uma interface comum, que garante que não há restrições na ordem das camadas, podendo estas estarem dispostas por qualquer ordem. Permitindo assim uma configuração flexível da pilha de camadas. Nestes sistemas os pedidos fluem de

camada em camada até todas realizarem processamento e o pedido chegar ao sistema de ficheiros inferior, ou até o processamento em alguma das camadas ser abortado por algum erro.

Os sistemas de ficheiros empilháveis podem ser categorizados em duas vertentes no que diz respeito ao seu desenvolvimento: *kernel* e *user-space*.

2.1.1 *Soluções em kernel-space*

Os sistemas de ficheiros empilháveis desenvolvidos em *kernel* aplicam este conceito estendendo a interface *vnode* [46]. Inicialmente, os sistemas de ficheiros estavam completamente integrados no sistema operativo, e as chamadas de sistema (p.e., *open*, *read*, *close*) invocavam diretamente as operações dos sistemas de ficheiros. Esta arquitetura impossibilitava o uso de vários/diferentes sistemas de ficheiros. A introdução do *vnode* proporcionou uma camada de abstração que separou o sistema operativo dos sistemas de ficheiros.

O *vnode* é uma estrutura utilizada no *kernel* dos sistemas Unix, para representar ficheiros, diretorias, ou outros objetos do *namespace* de um sistema de ficheiros (p.e., *sockets*, *FIFOs*). Esta estrutura contém um conjunto de operações (p.e., *vn_open*, *vn_read*, *vn_close*) que podem ser realizadas num *vnode* dentro de um sistema de ficheiros. Estas operações são definidas pelo sistema de ficheiros a que o objeto pertence, e são invocadas quando as aplicações realizam as respetivas chamadas de sistemas. Como referido anteriormente, esta interface permite separar as operações de sistemas de ficheiros genéricos da implementação de sistemas de ficheiros específicos, permitindo assim, às aplicações acesso transparente a diferentes sistemas de ficheiros.

Para modularizar funcionalidades de sistemas de ficheiros, o conceito de *vnode* foi estendido para permitir empilhar *vnodes* [60, 41, 80, 83]. Esta extensão do *vnode*, permite que um objeto *vnode*, que normalmente se relaciona com os sistemas de ficheiros, possa interagir com outros *vnodes*. Com o empilhamento de *vnodes*, várias interfaces de *vnodes* podem existir, e estas podem chamar-se umas às outras em sequência, isto é, uma determinada operação do nível de empilhamento N invoca a operação correspondente do nível $N + 1$, e assim por diante. Em cada nível (ou camada), as operações e os dados podem ser modificados para garantirem a funcionalidade pretendida para o sistema de ficheiros.

A Figura 1 ilustra a arquitetura do sistema de ficheiros Wrapfs [82], que utiliza o conceito de *vnodes* empilháveis para criar um sistema de ficheiros modular. O Wrapfs foi desenvolvido para facilitar o desenvolvimento de outros sistemas de ficheiros através de *templates* empilháveis, procurando assim libertar os programadores de detalhes do sistema operativo. Neste sistema, as chamadas de sistema realizadas pelas aplicações (p.e., *read*) são convertidas em chamadas ao nível do *vnode* (p.e., *vn_read*), que invocam as chamadas equivalentes do sistema Wrapfs (p.e., *wrapfs_read*). O Wrapfs, depois de aplicar a funcionalidade pretendida

sobre os dados (p.e., compressão, deduplicação), invoca novamente as operações genéricas do *vnode*, que por sua vez, invoca as suas respectivas operações específicas do sistema de ficheiros do nível inferior.

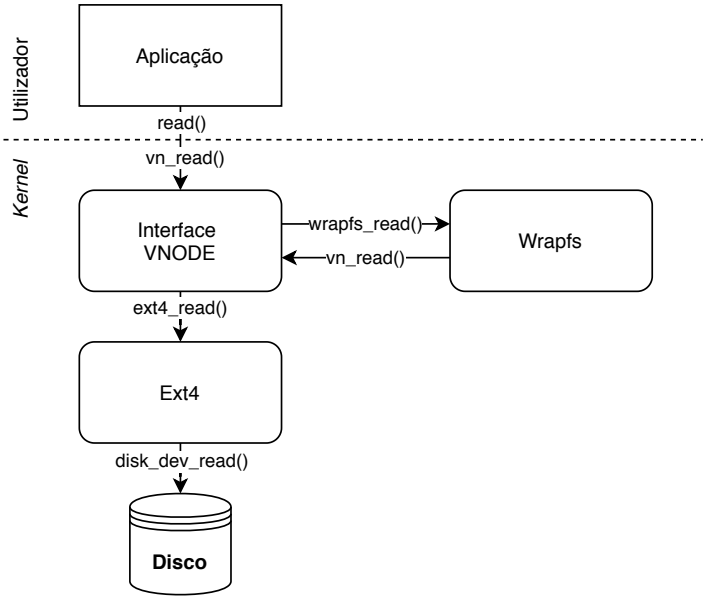


Figura 1: Arquitetura de um sistema de ficheiros empilhável que estende a interface *vnode*.

O Cryptfs [81] é um sistema de ficheiros desenvolvido a partir do Wrapfs que, através de uma camada de cifragem, procura introduzir confidencialidade e privacidade de dados num sistema de ficheiros. Este sistema pode ser empilhado sobre sistemas de ficheiros locais ou remotos, que garantem a persistência dos dados cifrados. Contudo está limitado ao uso de apenas um algoritmo de cifragem de dados, o Blowfish [62]. O Cryptfs foi desenvolvido como um exemplo de prova de conceito do que os sistemas de ficheiros empilháveis podem oferecer. O seu sucessor, o NCryptfs [78], é um sistema de ficheiros mais flexível que permite aos utilizadores ajustar o nível de segurança de acordo com os requisitos das aplicações. Este sistema inclui suporte para múltiplo utilizadores, múltiplas chaves, múltiplas cifras e múltiplos métodos de autenticação.

Uma das principais limitações destes sistemas ficheiros empilháveis é a fraca portabilidade entre plataformas, pois a sua implementação está dependente das características dos sistemas operativos [82]. Por exemplo, a interface *vnode*, para além de diferir entre diferentes sistemas operativos, também pode diferir entre diferentes versões de *kernel* do mesmo sistema operativo. Nesse sentido surge o FiST [84, 79], uma linguagem de alto nível para definir sistemas de ficheiros empilháveis, e que gera módulos de sistemas de ficheiros para várias plataformas.

Contudo, a principal desvantagem deste tipo de sistemas de ficheiros empilháveis é o seu desenvolvimento em *kernel*. Um ambiente de desenvolvimento hostil onde o progresso

é lento, a depuração é difícil, e erros simples podem danificar os sistemas. Ainda, o desenvolvimento no *kernel* requer uma compreensão profunda dos aspectos internos do sistema, resultando em muito tempo gasto pelo programadores a familiarizarem-se com os detalhes deste.

2.1.2 *Soluções em user-space*

O desenvolvimento de sistemas de ficheiros em *user-space* trazem vários benefícios comparativamente a sistemas de ficheiros desenvolvidos em *kernel*, nomeadamente, ao nível da rapidez de implementação, manutenção, versatilidade e portabilidade [69]. Em maior detalhe:

RAPIDEZ DE IMPLEMENTAÇÃO Os programadores têm ao seu dispor várias ferramentas que permitem melhorar/acelerar o desenvolvimento de *software* (p.e., depuradores, ferramentas de *tracing* e *profiling*). Erros de implementação não causam falhas de todo o sistema, apenas provocam falha de uma aplicação, que normalmente não implicam reinicialização do sistema. Os programadores não estão limitados às linguagens de programação orientadas para o desenvolvimento de sistemas em *kernel*, em vez disso, podem utilizar várias linguagens de alto nível (p.e., C++, Rust, Java, Go), de acordo com os objetivos pretendidos (p.e., desempenho, facilidade de desenvolvimento).

PORTABILIDADE Desenvolver código compatível com múltiplas plataformas é mais fácil em *user-space* do que em *kernel-space*. Um exemplo prático disso é o Arla [77] - um cliente do sistemas de ficheiros distribuídos AFS [45, 44] -, que através do desenvolvimentos em duas componentes, uma em *user-space* e outra em *kernel-space*, permitiu que apenas 10% do código ficasse dependente do sistema operativo, facilitando assim a implementação e a portabilidade do cliente.

BIBLIOTECAS Existem várias bibliotecas disponíveis em *user-space*, que podem ser utilizadas para melhorarem o desempenho dos sistemas de ficheiros. Por exemplo, técnicas de *prefetching*, utilizadas para melhorar o desempenho na leitura de dados, podem recorrer a algoritmos baseados em inteligência artificial para se adaptarem a cargas de trabalho de certas aplicações.

O Filesystem in Userspace (FUSE) [18] é a plataforma mais utilizada para desenvolver sistemas de ficheiros em *user-space* [74, 69]. Esta plataforma possibilita que programas em espaço de utilizador, executados por utilizadores sem quaisquer privilégios especiais, consigam exportar um sistema de ficheiros, ao estilo POSIX [75], para *kernels* Linux, sem que para isso seja necessário produzir código ao nível do *kernel*.

A Figura 2 ilustra a arquitetura do FUSE, com os dois componentes que o constituem em destaque - o módulo do *kernel* (fuse.ko) e uma biblioteca em espaço de utilizador, com o

nome de *libfuse*, que permite a comunicação com o já referido módulo. O módulo FUSE em *kernel* regista o *device file* /dev/fuse, que serve como interface entre a biblioteca em *user-space* e o *kernel*. A aplicação em *user-space* lê os pedidos FUSE através do *device file* registado, processa-os, e depois responde, escrevendo a resposta no mesmo *device file*

Como é possível observar na figura, um sistema de ficheiros desenvolvido em FUSE não substitui um sistema de ficheiros desenvolvido em *kernel*, sendo por isso necessário utilizar outros subsistemas, como sistemas de ficheiros locais (p.e., Ext4, XFS, Btrfs), para persistir os dados. Possibilitando assim o desenvolvimento de sistemas de ficheiros empilháveis, onde a componente superior do sistema é desenvolvida em *user-space*.

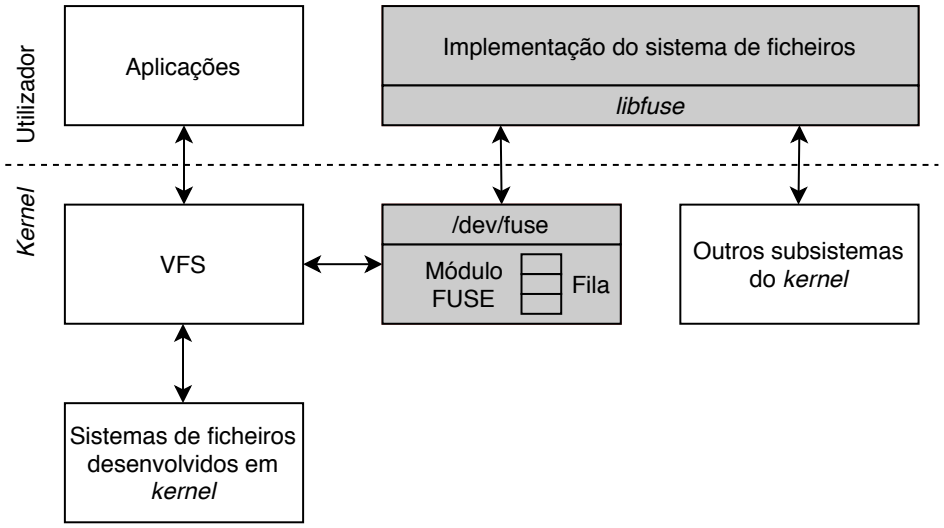


Figura 2: Arquitetura do FUSE.

A implementação de um sistema de ficheiros em *user-space* baseado em FUSE é feita por várias etapas. Inicialmente é desenvolvida uma aplicação, que implementa a interface FUSE recorrendo à *libfuse*. Para isso, a aplicação implementa *callbacks* relativos às operações POSIX principais sobre sistemas de ficheiros (operações descritas na Tabela 1). Estas operações de *callback* incluem as operações orientadas aos dados a serem acedidos/armazenados, por exemplo, através das escritas (*write*) e leituras (*read*), e também operações orientadas aos metadados (p.e., *getattr*, *chmod*). Nestas operações de *callback* são implementadas as funcionalidades pretendidas pelos sistemas de ficheiros desenvolvidos em FUSE. Por exemplo, num sistema de ficheiros que utilize o FUSE para realizar compressão de dados é esperado que essa funcionalidade seja implementada na operação de escrita (*write*), e a descompressão na operação de leitura (*read*).

Tabela 1: Operações FUSE da versão 2. Estas operações refletem a API síncrona de alto nível do FUSE, onde o processamento de um pedido termina quando a operação de *callback* termina, e as *callbacks* utlizam nomes e caminhos de ficheiros em vez de *inodes* para identificarem os ficheiros. Salvo especificação em contrário, todas as operações retornam um número zero ou um número positivo em caso de sucesso, ou um valor negativo selecionado de *errno.h* se ocorrer um erro.

Operações	**Descrição**
getattr(path, stbuf) $\rightarrow$ *res*	Preenche a estrutura *stbuf* (*struct stat*) com os atributos do ficheiro (p.e., tamanho, tipo e modo) identificado pelo caminho *path*.
access(path, mode) $\rightarrow$ *res*	Verifica a acessibilidade do ficheiro *path* para as permissões de acesso indicadas em *mode*.
create(path, mode, fi) $\rightarrow$ *res*	Cria o ficheiro *path* com o modo de abertura presente na estrutura *fi* (*struct fuse_file_info*) e com as permissões de acesso *mode*.
open(path, fi) $\rightarrow$ *res*	Abre o ficheiro *path* com o modo de abertura presente na estrutura *fi* (*struct fuse_file_info*).
read(path, buf, size, offset, fi) $\rightarrow$ *res*	Lê *size* bytes de dados do ficheiro *path* a partir de *offset* para *buf*. Retorna o número de bytes lidos, ou zero se o *offset* for igual ou superior ao tamanho do ficheiro.
write(path, buf, size, offset, fi) $\rightarrow$ *res*	Escreve *size* bytes de dados de *buf* para o ficheiro *path* a partir de *offset*. Retorna o número de bytes escritos.
flush(path, fi) $\rightarrow$ *res*	Fecha o ficheiro *path*.
release(path, fi) $\rightarrow$ *res*	Liberta dados/recursos alocados ao ficheiro *path*. Esta operação ocorre quando todos os descritores de um ficheiro foram fechados.
fsync(path, isdatasync, fi) $\rightarrow$ *res*	Persiste dados modificados do ficheiro *path* para o disco. Se *isdatasync* é diferente de zero, apenas devem ser persistidos os dados, caso contrário devem ser persistidos dados e metadados.
unlink(path) $\rightarrow$ *res*	Remove o ficheiro *path*.
mkdir(path, mode) $\rightarrow$ *res*	Cria a diretoria *path* com as permissões de acesso *mode*.
rmdir(path) $\rightarrow$ *res*	Remove a diretoria *path*.
rename(path_from, path_to) $\rightarrow$ *res*	Muda o nome do objeto *path_from* do sistema de ficheiros para *path_to*.
chmod(path, mode) $\rightarrow$ *res*	Muda as permissões do ficheiro *path* para *mode*.
readdir(path, buf, filler, offset, fi) $\rightarrow$ *res*	Preenche a estrutura *buf*, utilizando a função *filler*, com as entradas da diretoria *path* a partir de *offset*.

Depois de implementadas as operações de *callback*, segue-se a montagem do sistema de ficheiros. O sistema de ficheiros é montado sobre um *namespace*, desta forma, todos

os pedidos destinados a este *namespace* serão reencaminhados para o sistema de ficheiros desenvolvido em FUSE. Durante este processo regista-se a existência de um *handler* de operações no módulo do *kernel*, que permite que a partir desse momento possam ser trocados dados relativos a chamadas de sistema com o mesmo módulo, através do *device file* /dev/fuse.

A partir deste momento, o sistema de ficheiro baseado em FUSE está pronto a receber pedidos das aplicações. Todas as chamadas de sistema, efetuadas pelas aplicações, são enviadas para o Virtual File System (VFS). O VFS é um sistema de ficheiros virtual que fornece uma abstração global de todo *namespace* do sistema operativo. Para cada pedido, o VFS verifica o caminho e valida a qual sistema de ficheiros é direcionado, reencaminhado para o mesmo. No caso do sistema de ficheiros FUSE, este está registado como qualquer outro sistema de ficheiros. Desta forma, os pedidos destinados à diretoria que o sistema de ficheiros FUSE compreende serão reencaminhados para módulo FUSE, que os coloca uma fila para serem processados.

O componente em *user-space* retira o pedido da fila em *kernel*, lendo o *device file*. Com o pedido em *user-space*, é-lhe aplicada a respetiva função de *callback* que aplicará as funcionalidades previamente implementadas (p.e., *caching*, compressão, cifragem). Após aplicadas as funcionalidades sobre os dados, o pedido é reencaminhado para o sistema de ficheiros (p.e, Ext4, Btrfs), que está a ser utilizado pelo sistema de ficheiros desenvolvido em FUSE, para persistir os dados. Depois da componente em *user-space* obter resposta do sistema que garante a persistência dos dados, escreve a resposta ao pedido, que será enviada em sentido oposto, através do mesmo *device file*.

Existem vários sistemas de ficheiros desenvolvidos em *user-space* que utilizam FUSE para adicionarem novas funcionalidades a sistemas de ficheiros. Por exemplo, o CryFS [5] e EncFS [9] são sistemas de ficheiros que adicionam funcionalidades de privacidade e confidencialidade de dados através da cifragem de dados. O LessFS [17] oferece deduplicação - método para reduzir espaço de armazenamento através da eliminação de dados redundantes - e compressão com os algoritmos LZO [19] ou QuickLZ [23]. O Catfs [4] implementa funcionalidades de *caching* de dados com a semântica *write-through* (escritas são feitas de forma síncrona, tanto na *cache* como no sistema que persiste os dados) e *read-ahead* (leituras fazem *prefetching* de dados).

Contudo estes sistemas de ficheiros não se enquadram no contexto desta dissertação porque não são modulares, ou seja, não permitem ser ajustáveis ou extensíveis. Para colmatar estes problemas surgiu o SafeFS [54], uma plataforma para implementar sistemas de ficheiros flexíveis, modulares e extensíveis baseados em FUSE. A sua arquitetura permite empilhar camadas independentes, cada uma com a sua funcionalidade (p.e., compressão, cifragem, *caching*). Como todas as camada expõem uma interface idêntica à API fornecida pela biblioteca FUSE, estas camadas podem ser integradas com outros sistemas de ficheiros

desenvolvidos em FUSE, e empilhadas por diferentes ordens. Cada configuração de empilhamento resulta em sistemas de ficheiros com diferentes características, adequados a diferentes aplicações e cargas de trabalho. Contudo, a flexibilidade e modularidade deste sistema está limitada localmente onde o sistema de ficheiros está montado, ou seja, quando utilizado com camadas/sistemas (p.e., GCSF, dbxfs, s3fs [13, 6, 24]) que garantem a persistências dos dados em serviços de armazenamento baseado na nuvem (p.e., *Google Drive*, *Dropbox*, *Amazon S3* [14, 7, 1]), esta modularidade e flexibilidade não é oferecida no lado dos servidores.

Atualmente o SafeFS é considerado estado da arte no que toca a sistemas de ficheiros modulares em espaço de utilizador, tendo sido já utilizado para desenvolver o sistema TrustFS [40]. Uma plataforma segura, programável e modular para implementar sistemas de ficheiros com funcionalidades orientadas ao conteúdo (p.e., deduplicação, compressão) que executam em ambientes de execução confiáveis oferecidos pelo Intel SGX [37]. Esta plataforma procura reduzir o esforço de reimplementação de funcionalidades orientadas ao conteúdo dos sistemas atuais que utilizam o SGX para oferecerem garantias ao nível da segurança, utilizando a modularidade e flexibilidade do SafeFS.

Uma vez que esta dissertação pretende dotar o SafeFS com mecanismos de armazenamento remoto, vamos agora detalhar sobre a sua arquitetura e funcionalidade.

2.1.3 *SafeFS*

Como referido anteriormente, o SafeFS é uma plataforma, criada em FUSE, que permite construir sistemas de ficheiros flexíveis, modulares e extensíveis. A sua abordagem, baseada nos princípios de armazenamento definido por *software* [71, 47] - paradigma de armazenamento que reorganiza a pilha de E/S para separar o controlo e os fluxos de dados em dois planos de funcionalidade (*plano de dados* e *plano de controlo*) -, permite empilhar camadas independentes que implementem funcionalidades de armazenamento específicas. Estas camadas (p.e., compressão, deduplicação, replicação) podem ser empilhadas por diferentes ordens, bem como integradas com outro sistemas de ficheiros também baseados em FUSE. As diferentes configurações de empilhamento levam a sistemas de ficheiros com características diferentes, adequados a diferentes requisitos. Esta plataforma, com a arquitetura ilustrada na Figura 3, foi desenvolvida de forma responder aos seguintes objetivos:

EFICÁCIA reduzir o custo de implementação de novos sistemas de ficheiros, utilizando para isso camadas independentes, empilháveis e reutilizáveis.

COMPATIBILIDADE permitir integrar e incorporar, de forma simples, sistemas de ficheiros baseados em FUSE como camadas individuais.

FLEXIBILIDADE configurar a ordem e combinação de camadas de forma a ajustar o sistema ao requisitos das aplicações.

FACILIDADE DE USO ser transparente e utilizável como qualquer outro sistema de ficheiros construído em FUSE.

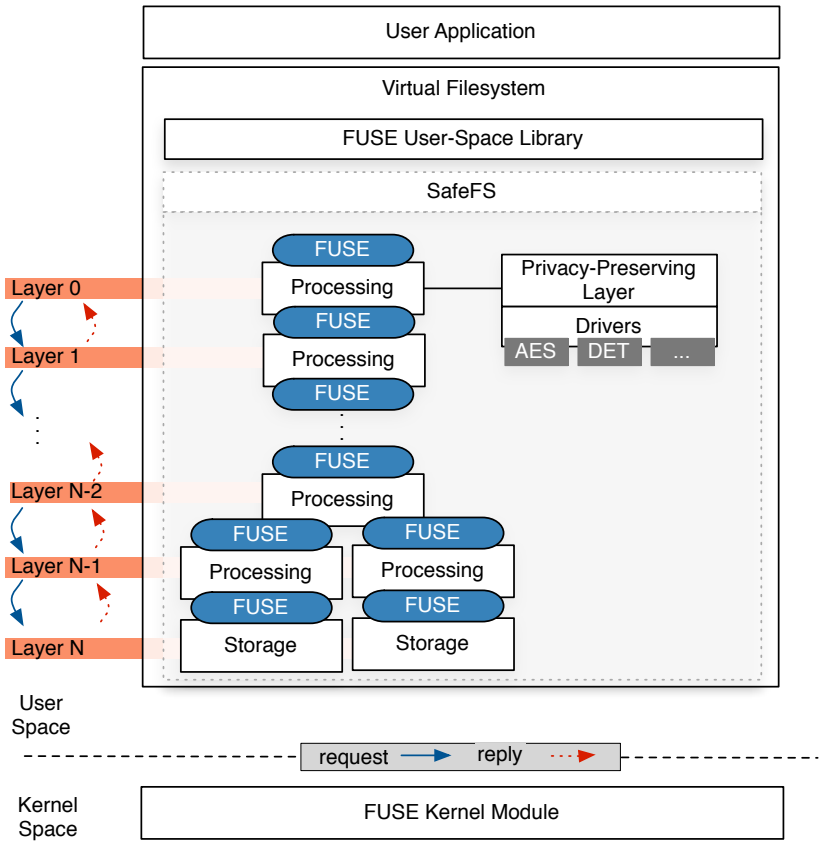


Figura 3: Arquitetura do SafeFS (imagem retirada de [54]).

Como em outros sistemas de ficheiros desenvolvidos em FUSE, e como explicado na secção anterior, as operações relacionadas com o sistema de ficheiros são intercetadas pelo módulo FUSE do *kernel*, e depois redirecionadas pela biblioteca FUSE no espaço do utilizador para o SafeFS. Cada operação intercetada é então processada pelas várias camadas que combinadas resultam na implementação de um sistema de armazenamento. Cada camada tem uma tarefa específica, e pode ser atribuída a uma das duas categorias de camadas do SafeFS: armazenamento ou processamento. As camadas de processamento manipulam os dados e/ou metadados, e reencaminham a operação para as camadas seguintes. Neste tipo de camadas inserem-se, por exemplo, as camadas de compressão, de cifra, ou de *caching*. As camadas de armazenamento têm como função persistir os dados e os metadados no meio de armazenamento designado, nos quais se incluem discos e serviços de armazenamento baseado na nuvem (p.e., *Google Drive*, *Dropbox*). Todas as camadas expõem uma interface idêntica à fornecida pela API da biblioteca FUSE, podendo assim as camadas serem empi-

lhadas por qualquer ordem. Para garantir que os dados são efetivamente persistidos em disco, a última camada do SafeFS corresponde a uma camada de armazenamento.

Segundo a arquitetura deste sistema as operações são ordenadamente transmitidas pelas várias camadas, de modo a que a cada camada apenas receba operações da camada imediatamente anterior e envie operações para a camada imediatamente posterior. Porém, a ordem das camadas é extremamente relevante no desempenho que o sistema pode oferecer. A ordenação arbitrária ou errada de camadas de processamento podem degradar o desempenho do sistema. Por exemplo, uma camada criptográfica seguida de uma camada de compressão traduz-se numa configuração que prejudica o desempenho do sistema [34]. Esta combinação de camadas irá resultar numa compressão ineficiente, pois os algoritmos de compressão dependem de padrões nos dados para reduzir o tamanho ocupado por estes. Padrões estes que são destruídos pela camada de criptografia, que transforma os dados em conteúdo indistinguível de conteúdo aleatório.

Algumas camadas podem conter diferentes algoritmos que podem ser configurados com diferentes definições, embora estes não alterem o fluxo de execução das operações. Por exemplo, uma camada que faz *caching* de blocos dados, pode ter diferentes algoritmos (p.e., Least Recently Used (LRU), Most Recently Used (MRU), First In First Out (FIFO)) para selecionar qual o bloco ser substituído quando a *cache* está cheia, e se pretende adicionar um novo bloco. Como forma de promover a reutilização de camadas e a modularidade, o SafeFS inclui a noção de *driver*. Assim, cada camada pode utilizar diferentes *drivers*, que apenas têm de cumprir uma determinada API definida pela camada. No caso da *cache* de dados, utilizando a noção de *drivers*, a lógica da *cache* seria comum a todas as *caches* de dados (i.e, os processos de alocação, cópia e remoção de blocos seriam os mesmos). Para selecionar os blocos a substituir quando a *cache* estiver cheia, seriam utilizadas as operações definidas pela API do *driver*.

Os *drivers* são carregados durante a montagem do sistema de ficheiros, e podem ser alterados sem necessitar de recompilar o sistema de ficheiros, desde que a nova configuração se mantenha compatível com a versão anterior. Por exemplo, ainda utilizando o exemplo da *cache* de dados, diferentes políticas de substituição são sempre compatíveis com versões anteriores. Já para uma camada que faça cifragem de dados, diferentes técnicas criptográficas vão impedir a compatibilidade com sistemas anteriores.

O SafeFS é configurado através de um ficheiro de configuração, que é lido durante a montagem do sistema de ficheiros. Neste ficheiro de configuração, o administrador de sistemas, para compor o sistema de ficheiros, seleciona e ordena as camadas que implementam às funcionalidades pretendidas, e configura cada uma das camadas.

2.2 SISTEMAS DE FICHEIROS REMOTOS

Os sistema de ficheiros locais (p.e., Ext4, XFS, Btrfs) são muito otimizados e com várias funcionalidades (p.e., compressão, desfragmentação, deduplicação). Contudo, não possibilitam o acesso e armazenamento de dados por parte de clientes remotos. Esta funcionalidade é pretendida por clientes que procuram ter dados replicados em diferentes servidores; cópias de segurança num sistema secundário, para recuperação dos dados em caso de falhas no sistema primário; ou partilha de informação entre computadores na mesma rede.

Inicialmente, estas necessidades eram satisfeitas por aplicações que implementavam protocolos de transferência de ficheiros, como o FTP [57], para transferirem ficheiros inteiros entre computadores através da rede. O FTP permite que clientes, para além de transferirem ficheiros para o servidor, possam criar, remover e mudar o nome de ficheiros no servidor, bem como manipular (p.e., criar, remover) diretorias deste. No entanto, não permite que o conteúdo de ficheiros seja manipulado diretamente pelas aplicações. Isto faz com que as aplicações para manipularem o conteúdo de ficheiros transfiram os ficheiros na sua totalidade para o disco local antes destes poderem ser lidos ou modificados. Esta necessidade de transferir ficheiros traz vários inconvenientes. Com os ficheiros remotos a precisarem de ser transferidos para serem modificados/lidos, estes ficheiros não podem ser abertos diretamente pelas aplicações. Assim, as aplicações que foram desenvolvidas para acederem diretamente a ficheiros de um sistema de ficheiros local precisam de ser modificadas e recompiladas para poderem operar neste modelo. Ainda como inconveniente, os ficheiros têm de ser transferidos para o disco local para serem lidos, e transferidos, novamente, para o servidor se forem modificados. Como consequência disto, é necessário espaço de armazenamento no disco local para armazenar os ficheiros remotos.

Para colmatar estes problemas, foram desenvolvidos sistemas de ficheiros remotos que permitem acesso e armazenamento transparente a ficheiros remotos. Estes sistemas de armazenamento removem a necessidade de transferir ficheiros na sua totalidade para poderem ser lidos ou manipulados. Os ficheiros permanecem no servidor e são manipulados diretamente no servidor a pedido dos clientes. Esta manipulação no local onde os ficheiros são armazenados traz vários benefícios, principalmente se forem pequenas modificações. Por exemplo, para escrever na forma de *append* num ficheiro, apenas é necessário enviar os dados para realizar a operação *append*. Enquanto que com os protocolos de transferência o ficheiro precisava de ser transferido nos dois sentidos. Transferências estas altamente penalizadoras para ficheiros de grandes dimensões.

Comparativamente com os protocolos de transferência de ficheiros, os sistemas de ficheiros remotos apresentam várias vantagens: os ficheiros remotos são transparentes para o cliente, ou seja, as aplicações não precisam de ser modificadas ou recompiladas para os poderem usar; se as aplicações no cliente apenas precisam de uma parte do ficheiro, apenas essa parte

precisa de ser transferida; como os sistemas de ficheiros remotos acedem diretamente aos ficheiros do servidor, o ficheiro está sempre atualizado, assumindo que não há incoerências causada por *caches*; o cliente não precisa de ter espaço de armazenamento disponível para aceder aos ficheiros remotos; e as aplicações podem utilizar mecanismos de controlo de concorrência para evitar que vários clientes modifiquem ficheiros em simultâneo.

Os sistemas de ficheiros remotos seguem o modelo cliente-servidor, e são constituídos por três componentes: o cliente, o servidor, e o protocolo de comunicação. O cliente fornece às aplicações acesso e armazenamento remoto a ficheiros, para isso, reencaminha os pedidos locais para o um servidor remoto, e espera pela sua resposta. O servidor implementa as operações de um servidor de ficheiros remoto, que acedem ao armazenamento local (p.e., ficheiros num sistema de ficheiros local). Esta componente recebe os pedidos do cliente, processa-os, e envia as respostas de volta para o cliente. O protocolo de comunicação define um conjunto de procedimentos que seguem o padrão pedido-resposta. Estes procedimentos são definido pelos seus argumentos e resultados, e os seus efeitos, ou seja, definem como o cliente e o servidor devem comunicar para o sistema funcionar corretamente. A comunicação nestes sistemas de armazenamento é baseada em Remote Procedure Calls (RPC) [29], que têm como objetivo simplificar a definição, organização e implementação de serviços remotos. RPC fornece um mecanismo para um cliente invocar um procedimento que parece ser local, mas que na realidade é executado noutra máquina da rede.

O Network File System (NFS) foi um dos primeiros sistemas de ficheiros remotos a surgir, e foi concebido para simplificar a partilha de ficheiros numa rede com máquinas heterogéneas. Além disso, tinha também como o objetivo ser transparente, proporcionando acesso a ficheiros remotos sem a necessidade de modificar ou recompilar as aplicações que o usam [61, 51]. Desde então, tem estado em constante desenvolvimento, contando já com várias versões, sendo as versões v3 e v4 as mais utilizadas atualmente.

Um dos objectivos iniciais do desenho do protocolo NFSv3 era fazer com que cada pedido fosse independente, para que um servidor não precisasse manter estado entre pedidos de clientes. Não manter estado no servidor permite que o processo de recuperação no caso de faltas seja muito simples. Por exemplo, quando um servidor falha, o cliente apenas tem de reenviar os pedidos até que obtenham resposta, e o servidor não realiza qualquer processo para recuperar da falha. Quando é um cliente que falha, tanto no cliente como no servidor não é preciso realizar qualquer processo de recuperação.

Uma consequência desta abordagem é a ausência de operações para abrir e fechar ficheiros no protocolo NFSv3, pois estas operações implicariam que o servidor mantivesse estado, por exemplo, quais os ficheiros abertos pelos clientes. Ainda, pela mesma razão, o protocolo NFS não oferece procedimentos para fazer a gestão de controlo de acessos concorrentes a ficheiros. Para controlar acessos concorrentes a ficheiros no sistema NFS é preciso utilizar um serviço externo, que implementa o protocolo Network Lock Manager (NLM) [32], e que

por sua vez mantém estado sobre os clientes. O protocolo NLM utiliza um modelo de gestão de concorrência similar ao POSIX, que permite a uma aplicação no cliente bloquear uma região do ficheiro definida por um *offset* e um comprimento.

Mais recentemente surgiu o NFSv4 [53, 64], que procurou introduzir várias melhorias no protocolo NFS. Um dos problemas identificados nas versões anteriores do NFS é a elevada latência das operações quando utilizado em redes de longa distância. Para resolver este problema introduziram-se os procedimentos compostos, que permitem incorporar várias operações relacionadas num único pedido a ser enviado para o servidor, diminuindo assim o número de operações realizadas entre clientes e servidores. Ainda, neste sentido de diminuir o número de pedidos entre cliente e servidor, as operações que inicialmente procuravam por um nome (de um objeto do sistema de ficheiros) numa diretoria, procuram agora por um caminho completo numa diretoria.

Como referido anteriormente, nas versões inicias do NFS o servidor não mantinha estado. Esta abordagem permite simplificar o processo de recuperação no casos de falhas. No entanto, isto traz problemas na implementação de algumas funcionalidades (p.e., controlo de concorrência e partilha de ficheiros) em alguns sistemas operativos, nomeadamente no sistema operativo Windows. Para resolver estes problemas, foram adicionadas ao protocolo NFS as operações para abrir e fechar ficheiros, que implicam introduzir estado no servidor. A operação para abrir um ficheiro, além de abrir o ficheiro, fornece uma operação atómica para procurá-lo, criá-lo, se necessário, e garantir acesso exclusivo. A operação para fechar um ficheiro, além de fechar o ficheiro, liberta qualquer estado que esteja associado ao ficheiro. Com a introdução de estado no protocolo NFS, foi também possível remover a dependência do protocolo NLM. Desta forma, o protocolo NFSv4 integra também operações que controlam o acesso concorrente a ficheiros.

Realizar *caching* de dados e metadados no cliente é essencial para um bom desempenho, por isso, nesta versão, o servidor NFS pode delegar *caching* seletivo a um cliente para otimizar as operações de acesso e armazenamento sobre o mesmo. Isto é, o servidor cede o controlo das actualizações de ficheiros a um cliente durante um período através de uma delegação. Ainda, o cliente não tem forma de pedir uma delegação, a decisão de concedê-la é feita unicamente pelo servidor, através da análise de padrões de acesso a ficheiros. Por exemplo, se um ficheiro estiver a ser escrito por vários clientes, o servidor, muito provavelmente, não irá conceder nenhuma delegação a nenhum dos clientes. Já quando um ficheiro só estiver a ser manipulado por um cliente, a possibilidade de-lhe ser atribuída uma delegação é elevada.

O Server Message Block (SMB) é outro protocolo de comunicação, que permite partilhar ficheiros entre sistemas Disk Operating System (DOS) através da rede [43]. O protocolo, inicialmente desenvolvido pela IBM, segue o modelo cliente-servidor, e foi concebido para permitir gerir e aceder recursos remotos, tais como ficheiros e impressoras [63]. Com este

protocolo as aplicações conseguem, por exemplo, abrir, ler, escrever e remover ficheiros num servidor remoto.

Sendo um protocolo, define um conjunto de procedimentos/comandos, e não um sistema em particular. Desde que apresentado pela IBM tem estado em constante desenvolvimento por parte da Microsoft, contando já com várias versões, variantes e implementações.

A primeira versão do protocolo exigia que a comunicação utilizasse a API Network Basic Input/Output System (NetBIOS) para a comunicação entre cliente e servidor [42]. NetBIOS é um serviço para PCs que expõe uma API que permite às aplicações comunicarem entre si através de uma rede local (LAN). Este serviço é responsável por três tarefas: registo e resolução de nomes; gestão da sessão que é utilizada durante uma ligação; e envio de datagramas (pacotes enviados numa rede não confiável) que não requerem uma comunicação com ligação/conexão [67]. Contudo, o NetBIOS é um serviço ineficiente porque envolve muita comunicação, e várias transmissões em *broadcast*.

Mais tarde, a Microsoft desenvolveu uma variante do SMB - o Common Internet File System (CIFS) -, que permite que a comunicação seja feita diretamente sobre TCP/IP. A remoção da dependência do NetBIOS trouxe várias vantagens: simplificou o transporte do tráfego SMB; removeu a utilização de *broadcasting* como meio de resolução de nomes; e permitiu normalizar o Domain Name System (DNS) como meio para resolução de nomes.

Nesta primeira versão é introduzida a noção de *opportunistic lock* (*OpLock*) como um mecanismo dos clientes fazerem *caching* para melhorar o desempenho [42]. Os *OpLocks* são um mecanismo de semelhante às delegações do NFSv4. Ou seja, um cliente que adquiriu um *OpLock* pode modificar diretamente os dados que tem em *cache* sem necessitar de enviar essas operações para o servidor. As delegações do NFSv4 diferem dos *OpLocks* no sentido em que as delegações são atribuídas pelos servidores, enquanto que os *OpLocks* são solicitados pelos clientes.

A primeira grande revisão do protocolo SMB, designada de SMB2 [26], foi fornecida pela Microsoft, quando lançou os sistemas operativos Windows Vista e Windows Server 2008. Um dos problemas identificados nas versões anteriores, é o excesso de comunicação entre o cliente e o servidor, que por sua vez prejudica o desempenho do sistema que implementa o protocolo. Para resolver este problema foram feitas várias otimizações. A principal otimização está relacionada com a simplificação do protocolo, nas versões iniciais eram definidos mais do que cem comandos e subcomandos. Nesta nova versão o número de comandos foram reduzidos para dezanove. Foram também introduzidos os pedidos compostos, que permitem que múltiplos pedidos sejam enviados como um único pedido pela rede, e ainda leituras e escritas com maiores dimensões.

O SMB3 [26] foi lançado quando surgiu o Windows 8 e o Windows Server 2012, e focou-se em melhorar a segurança e o desempenho. Por exemplo, através de encriptação de ponto-a-ponto, e de múltiplos canais. Este último, combina a largura de banda de várias placas

de rede, e permite os fluxos de dados sejam divididos para transferências de dados mais rápidas.

O SMB é um protocolo de comunicação utilizado sobretudo no sistema operativo Microsoft Windows. Para permitir que clientes com o sistema operativo Windows consigam aceder a recursos de servidores com sistemas operativos baseados em Unix foi desenvolvido o Samba. O Samba [25] é um sistema, em que as várias partes implementam o protocolo SMB. Ao utilizar este protocolo o Samba permite que os máquinas com sistemas operativos baseados um Unix comuniquem com máquinas com o sistema operativo Microsoft Windows e outros clientes e servidores que também suportem o protocolo SMB [35]. Este sistema, tem como principal objetivo garantir a interoperabilidade de sistemas Windows e Unix que estejam na mesma rede e que queiram partilhar recursos.

Existem ainda outros sistemas de armazenamento que permitem acesso remoto a ficheiros como o HDFS [65], Ceph [76] e Lustre [31]. Contudo, estes estão fora espectro dos sistemas de ficheiros remotos, uma vez que apresentam arquiteturas distribuídas, e oferecem garantias adequadas a cenários de larga escala, como escalabilidade, replicação de dados e tolerância a faltas.

No contexto desta dissertação, como demonstrado nos próximos capítulos, inspiramo-nos e comparamos o sistema desenvolvido com o NFSv3. A escolha recaiu sobre o NFS porque este é o sistema com melhor integração nos sistemas Unix. Relativamente à versão do NFS, a escolha recaiu sobre a versão 3, pois esta é uma versão mais matura e menos complexa. Além disso, estudos recentes revelam que a introdução de estado no protocolo NFSv4, que implica mais comunicação entre cliente e servidor, prejudica o desempenho do sistema quando utilizado em redes com pouca latência (p.e., LAN) [36]. Ainda, os procedimentos compostos, que procuram reduzir a comunicação entre cliente e servidor, e assim poderem compensar a comunicação feita para manter o estado no servidor não têm o desempenho esperado.

Na seguinte secção é feito um estudo mais aprofundado do NFS, onde é detalhada a sua arquitetura e funcionamento. Daqui em diante, a não ser que referido explicitamente, usaremos o termo NFS para nos referirmos ao NFSv3.

2.2.1 *NFS*

Tendo em conta as características anteriormente referidas, os princípios e objetivos do NFS são:

AGNÓSTICO DO *HARDWARE* E DO SISTEMA OPERATIVO Os protocolos de comunicação usados devem ser agnósticos do sistema operativo, para permitir que um servidor NFS possa fornecer ficheiros para vários tipos de clientes. Para além disso, estes protocolos devem também ser simples e consumir poucos recursos, possibilitando a implementação

do cliente em diferentes tipos de máquinas (desde simples computadores pessoais a servidores empresariais).

RECUPERAÇÃO A FALTAS Clientes e servidores devem ser capazes de recuperar facilmente de falhas nas máquinas ou problemas de rede. Ou seja, não deve ser necessário realizar nenhum processo de recuperação para que o sistema volte a estar funcional após a recuperação.

ACESSO TRANSPARENTE Fornecer um sistema que permita às aplicações acederem a ficheiros remotos de forma semelhante ao acesso local, sem haver necessidade de reimplementação por parte das aplicações ou utilização de bibliotecas adicionais.

Como referido anteriormente, o NFS é um sistema de armazenamento que fornece acesso e armazenamento transparente a ficheiros remotos. A arquitetura do sistema NFS, ilustrada na Figura 4, consiste em três módulos principais: o cliente, o servidor e o protocolo de comunicação entre cliente e servidor.

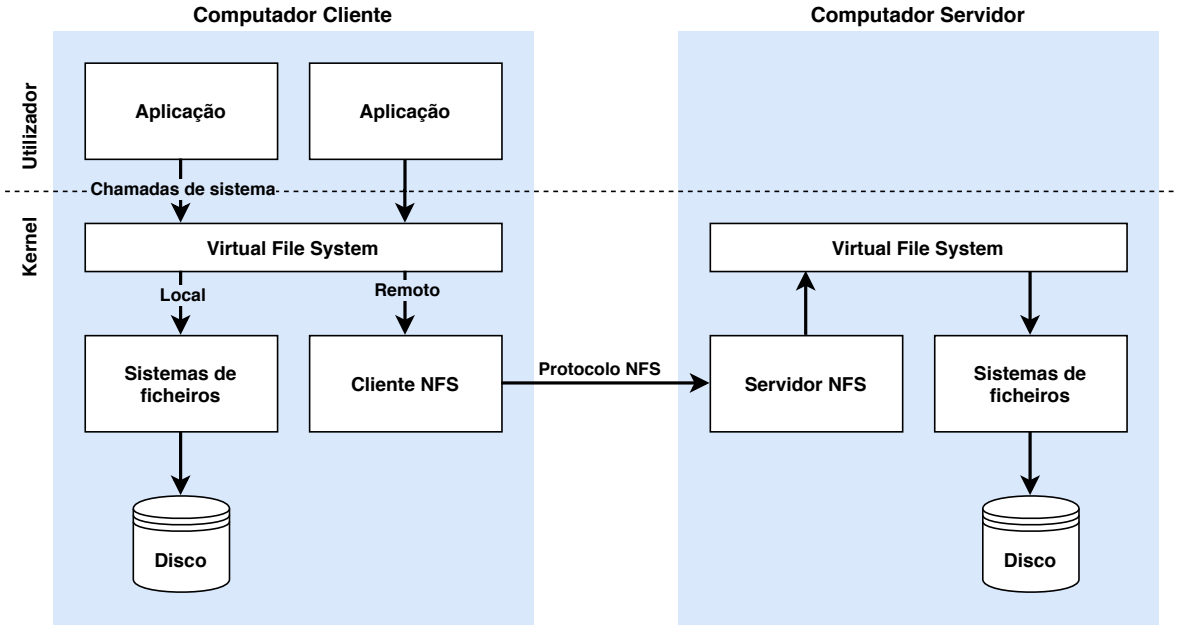


Figura 4: Arquitetura do NFS.

Cliente

O módulo cliente NFS é responsável por enviar pedidos (p.e., leitura e/ou escrita de dados) para o servidor e esperar pela sua resposta. Além disso, tem também de garantir que a interação entre aplicações e o NFS é transparente (i.e., as aplicações não precisam de ser modificadas para garantir o seu funcionamento).

Para garantir esta transparência, o módulo cliente NFS, desenvolvido em *kernel*, está registado no VFS. Este registo, garante que as aplicações no cliente acedem a ficheiros remotos através de chamadas de sistema definidas pela interface POSIX. Assim, as aplicações invocam as chamadas de sistema (p.e, *open*, *read*, *close*, *mkdir*), que são submetidas no VFS para acederem aos ficheiros armazenados no servidor. Por sua vez, o cliente NFS recebe os pedidos do VFS e executa os procedimentos necessários (do protocolo NFS) para responder a esses pedidos.

A interação do módulo cliente NFS com o VFS é idêntica à de outros sistemas de ficheiros, ou seja, transfere blocos dados de e para o servidor, e faz *caching* desses blocos na memória local sempre que possível.

Para verificar se os dados mantido em *cache* ainda estão válidos, quando uma aplicação abre o ficheiro, o cliente envia um pedido ao servidor para obter os atributos do ficheiro (p.e., tempo de modificação e tamanho do ficheiro) que contém esses dados. Se o tempo de modificação no servidor coincidir com o tempo de modificação dos atributos em *cache* no cliente, o cliente assume que a *cache* ainda está válida. Se os tempos de modificação entre servidor e a *cache* do cliente não coincidirem, o cliente assume que o ficheiro foi modificado, e que os dados mantidos em *cache* já não estão válidos.

Contudo, este método não consegue identificar a validade dos dados se o cliente modificar os dados que tem em *cache*. Por exemplo, se o cliente escrever numa parte de um ficheiro, os dados mantidos em *cache* para a restante parte provavelmente ainda são válidos. Mas o cliente não tem como confirmar essa informação, porque a operação de escrita do ficheiro no servidor modifica os atributos desse ficheiro, nomeadamente o tempo de modificação.

Neste sentido, para ajudar os clientes a determinar quando os dados da *cache* estão inválidos, o NFS introduz a noção de *cache* com fraca consistência (*weak cache consistency*). Para isto, na resposta às operações que modificam dados e/ou metadados (p.e, *write*, *mkdir*, *remove*) são enviadas duas versões dos atributos do ficheiro: imediatamente antes da operação ser realizada, e imediatamente depois da operação ser realizada. Se o tempo de modificação recebido nos atributos que antecederam a operação coincidirem com os atributos mantidos na *cache*, então a *cache* do cliente está válida, e o cliente deve atualizar os atributos armazenados na *cache* para os novos atributos recebidos. Caso contrário assume que informação que tem em *cache* já não está válida.

Ainda assim, com vários clientes em diferentes máquinas a acederem em simultâneo aos mesmos ficheiros, este modelo não garante verdadeira coerência da *cache*. Por exemplo, um cliente pode ter modificado um ficheiro localmente mas ainda não ter realizada a persistência dos novos dados para o servidor. E mesmo que o tenha feito, um outro cliente só verifica o tempo de modificação quando um ficheiro é aberto, ou quando o tempo da validade dos atributos em *cache* é ultrapassado. Desta forma, se um cliente modificar dados depois deste

ficheiro ter sido aberto por outro cliente, a *cache* deste outro cliente vai ficar incoerente até que do tempo de validade dos atributos seja ultrapassado.

Para melhorar o desempenho, o cliente realiza também *caching* de atributos de ficheiros e diretorias. Estes atributos mantêm-se válidos no cliente durante um certo período de tempo, e são utilizados para responder a outros pedidos que ocorram durante esse período, evitando assim o envio desses pedidos para o servidor. Tal como no mecanismo de *caching* de dados anteriormente descrito, este modelo não garante a coerência das *caches* quando múltiplos clientes modificam o mesmo ficheiro. Isto porque, alterações realizadas por outros clientes só são visíveis depois do período de validade dos atributos mantidos em *cache* serem ultrapassados. Até lá, os clientes têm uma visão incoerente dos dados do servidor.

No que diz respeito ao tipo de clientes do NFS, estes podem ser síncronos ou assíncronos. Globalmente, a principal diferença entre estes dois tipos de cliente é na forma como a operação de escrita de dados é gerida. No cliente síncrono do NFS qualquer escrita gera uma operação *write* para o servidor, que só responde depois de garantir que os dados modificados pela operação foram persistidos no disco. Este tipo de cliente oferece mais coerência entre *caches* de clientes, mas em contra partida acrescenta um custo significativo no desempenho.

O cliente assíncrono utiliza a noção de escritas assíncronas seguras para implementar o modelo assíncrono. As escritas assíncronas seguras combinam os procedimentos *write* e *commit* para garantir que não há perdas de informação. Assim, para cada escrita (*write*) para o servidor, este pode responder logo ao cliente, mesmo sem garantir que os dados foram persistidos em disco. A operação *commit* proporciona ao cliente uma forma de persistir os dados para disco (no servidor) de pedidos assíncronos anteriores, e detectar se é necessária a retransmissão de dados. Ainda, no cliente assíncrono do NFS nem todas as escritas efetuadas pelas aplicações geram operações *write* para o servidor. Isto permite agrupar dados de operações consecutivas, diminuindo assim o número de pedidos enviados para o servidor. O envio destas operações é atrasado até que aconteça algum dos seguintes eventos: o sistema precise da memória alocada para armazenar os dados destas operações; a aplicação faça explicitamente operações que causem *flush* dos dados (p.e., *sync* ou *fsync*), a aplicação feche o ficheiro (onde o *flush* é feito transparentemente); ou quando são feitas operações para controlar acessos concorrentes ao ficheiro [20].

Para melhorar o desempenho do sistema NFS na leitura de dados, o cliente utiliza também o mecanismo *read-ahead*. Este mecanismo faz *prefetching* de dados de ficheiros, de modo a que, quando posteriormente acedidos, o seu conteúdo seja lido a partir da memória e não através da rede, resultado em latências de acesso a ficheiros muito mais baixas.

Servidor

O módulo servidor NFS é responsável por receber os pedidos dos clientes, e realizar processamento necessário para lhes responder. Para isso, submete operações para o VFS,

que por sua vez realiza operações num sistema de ficheiros (p.e., Ext4, XFS) que garante a persistência dos dados.

O módulo servidor NFS está implementado no *kernel* como um módulo, e permite servir múltiplos clientes (NFS), contudo apenas os que estão devidamente autenticados serão atendidos. Adicionalmente, o servidor NFS suporta também mecanismos de autenticação mais robustos, recorrendo, por exemplo, a Kerberos [50], e também possibilita a cifragem de dados de forma a garantir privacidade dos mesmos.

Como o servidor NFS não mantêm estado sobre os clientes - para simplificar o processo de recuperação do servidor -, quando este realiza uma operação tem de persistir os dados modificados em disco antes de devolver os resultados. Por exemplo, num pedido de escrita síncrono (operação *write*), não só o bloco de dados, mas também quaisquer blocos indiretamente modificados e o bloco que contém o *inode* - objecto que representa toda a informação necessária ao *kernel* para manipular um ficheiro ou diretoria, na qual se inclui os metadados - devem ser persistidos se tiverem sido modificados. A única exceção a isto são a escritas assíncronas seguras, porque a operação *commit* permite que o cliente recupere de possíveis falhas do servidor.

Apesar disto, o servidor suporta os dois modelos de operação. O modelo síncrono assegura que o servidor só responde aos pedidos depois de garantir que os dados modificados foram persistidos em disco. Este modelo fornece mais garantias de persistência, mesmo que o servidor fique temporariamente indisponível (p.e., após uma falha) ou a ligação entre cliente e servidor falhe. No modelo assíncrono, o servidor responde ao cliente NFS logo que tenha processado o pedido e que este tenha sido enviado para o sistema de ficheiros local. Ou seja, este modelo não garante que os dados modificados foram persistidos antes de responder ao cliente NFS [10]. Ao seguir esta abordagem é possível melhorar o desempenho do sistema, contudo, oferece menos garantias de persistência dos dados, correndo o risco de perder informação aquando de uma falha. Importa referir que este modelo de operação assíncrono oferece garantias mais fracas do que do protocolo *standard* NFS, porque o cliente assume sempre que os dados modificados foram persistidos, ou seja, o cliente não tem um comportamento diferente quando o servidor utiliza este modelo para poder recuperar de possíveis falhas do servidor.

Comunicação

A comunicação entre os módulos cliente NFS e servidor NFS é realizada através do sistema de RPC da Sun [72], também designado por Open Network Computing Remote Procedure Call (ONC RPC). Este sistema de RPC foi desenvolvido para simplificar a definição, organização e implementação de serviços remotos. As chamadas de procedimento remoto são síncronas, isto é, a aplicação cliente bloqueia até que o servidor tenha completado a chamada e devolvido os resultados. Estas características fazem com que o sistema de RPC

seja fácil de utilizar e de compreender porque se comporta como um procedimento local [61]. Este sistema de RPC permite ser configurado para utilizar os protocolos User Datagram Protocol (UDP) [55] ou Transmission Control Protocol (TCP) [56]. Quando utilizado com UDP (um protocolo *stateless*) a ligação (em condições normais) minimiza o tráfego na rede. O servidor NFS envia ao cliente um *cookie* depois do cliente estar autorizado a aceder à diretoria partilhada. Este *cookie* é um valor aleatório armazenado no servidor, e é transmitido juntamente com os pedidos RPC enviados pelos clientes. O servidor NFS pode ser reiniciado sem afetar os clientes e o *cookie* permanece intacto após a reinicialização. Contudo, como o UDP é *stateless*, se o servidor falhar inesperadamente, os clientes continuam a saturar a rede com pedidos para o servidor. Por esta razão, o protocolo TCP passou a ser utilizado por padrão no NFS [20].

Os dados do protocolo NFS nas mensagens RPC devem ser representados num formato que possa ser compreendidos tanto pelo cliente como pelo servidor. Para isso, o RPC utiliza uma camada de interoperabilidade designada de External Data Representation (XDR) [38], que assegura que os componentes do NFS representam os dados da mesma forma. O XDR encarrega-se de converter os tipos de dados para uma representação comum, para que todas as arquitecturas possam interoperar e partilhar sistemas de ficheiros. Esta camada, define, por exemplo, o tamanho, ordem de bytes e alinhamento dos tipos de dados básicos (p.e., *string*, *integer*, *union*, *boolean* e *array*). Estruturas mais complexas podem ser construídas a partir dos tipos de dados básicos XDR.

A escolha de um protocolo de comunicação *stateless* tem duas implicações no desenho e implementação do NFS. Primeiro, cada pedido definido no protocolo NFS tem de descrever completamente a operação a ser executada. Por exemplo, quando é escrito um bloco de dados, o pedido de escrita (operação *write*) deve identificar qual é o ficheiro, o *offset*, e o tamanho dos dados a serem escritos. Isto é claramente diferente da chamada de sistema *write*, onde os dados são escritos para onde o descritor do ficheiro aponta. Assim, para o servidor NFS o estado contido por num descritor de ficheiro não existe, e este assume que o cliente tem toda a informação de estado necessária para realizar os pedidos. Segundo, os pedidos devem ser idempotentes, ou seja, se um cliente enviar o mesmo pedido mais do que uma vez os resultados devem ser equivalentes. Por exemplo, a leitura de um bloco de dados é uma operação idempotente, os mesmos dados são devolvidos como resultado de cada operação. Contudo, nem todas as operações são idempotentes. Por exemplo, a operação que remove um ficheiro não consegue ser realizada duas vezes, porque a segunda tentativa de remover o ficheiro falhará se a primeira for bem sucedida. Para resolver este problema, o servidor NFS implementa uma *cache* que mantém um registo de operações recentemente realizadas. Assim, se um pedido estiver nesta *cache*, a operação não é executada.

Tabela 2: Operações do servidor NFS (versão 3 do protocolo NFS, simplificada).

Operações	**Descrição**
lookup(dirfh, name) → fh, attr	Devolve o identificador do ficheiro *name* na diretoria *dirfh*, e os atributos do ficheiro e da diretoria.
getattr(fh) → attr	Devolve os atributos do ficheiro (p.e., tamanho do ficheiro, tempo da última modificação, ID do utilizador dono, etc) identificado por *fh*.
setattr(fh, attr) → attr	Define os atributos do ficheiro (p.e., modo, ID de utilizador, ID de grupo, tamanho e tempos de acesso e modificação) identificado por *fh*, e devolve os novos atributos do ficheiro.
access(fh, mask) → mask, attr	Devolve uma máscara de bits codificada com as permissões de acesso do cliente para o ficheiro fh, e os atributos desse ficheiro
create(dirfh, name, attr) → fh, attr	Cria o ficheiro *name* na diretoria *dirfh* com atributos *attr*, e devolve o identificador do ficheiro criado, os seus atributos, e os atributos da diretoria onde foi criado.
remove(dirfh, name) → attr	Remove o ficheiro *name* da diretoria *dirfh*, e devolve os atributos da diretoria de onde foi removido.
read(fh, offset, count) → data, attr	Devolve até *count* bytes de dados do ficheiro *fh* a partir de *offset*, e os atributos do ficheiro.
write(fh, offset, count, data) → attr	Escreve *count* bytes de dados no ficheiro *fh* a partir de *offset*, e devolve os atributos do ficheiro depois da operação de escrita.
rename(dirfh, name, todirfh, toname) → attr	Muda o nome do ficheiro *name* na diretoria *dirfh* para *toname* na diretoria *todirfh*, e devolve os atributos das duas diretorias.
mkdir(dirfh, name, attr) → fh, attr	Cria uma nova diretoria *name* com atributos *attr* na diretoria *dirfh*, e devolve identificador da diretoria criada, os seus atributos, e os atributos da diretoria onde foi criada.
rmdir(dirfh, name) → attr	Remove a diretoria *name* da diretoria *dirfh*, e devolve os atributos da diretoria de onde foi removida.
readdir(dirfh, cookie, count) → entries, attr	Devolve até *count* bytes de entradas da diretoria *dirfh*. Cada entrada contém o nome do ficheiro, o seu identificador e um apontador para próxima entrada da diretoria, designado de *cookie*. Este *cookie* é utilizado para as seguintes operações *readdir* começarem a ler a partir da entrada seguinte. Devolve também os atributos da diretoria pai.
fsstat(fh) → fsstats	Devolve informações do sistema de ficheiros (p.e., tamanho do bloco, número de blocos livres) para o sistema de ficheiros que contém um ficheiro *fh*.

Como é possível verificar na Tabela 2, todas as operações definidas no protocolo NFS [32] utilizam *file handles* (*fh*) para identificar os ficheiros ou diretorias sobre os quais serão realizadas. Estes identificadores não têm qualquer significado para os clientes, mas contêm

toda a informação necessária para o servidor conseguir distinguir e identificar os ficheiros alvos das operações. O identificador é gerado, pelo servidor, através do número do *inode* do ficheiro combinado com o identificador do sistema de ficheiros e com o número de geração do *inode*.

O cliente recebe o primeiro *file handle* para o sistema de ficheiros remoto durante a sua montagem, e este identificador refere-se à diretoria raiz do sistema de ficheiros exportado pelo servidor. Novos identificadores são enviados do servidor para o cliente nos resultados da operações *lookup*, *create* e *mkdir*. No sentido contrário, do cliente para o servidor, são enviados como argumentos das restantes operações.

Para reduzir o número de operações transmitidas entre cliente e servidor relacionadas com metadados (p.e, *getattr*), todas as operações do protocolo NFS, além do resultado dessa operação, devolvem também os atributos dos ficheiros/diretorias. Por exemplo, na operação de escrita que escreve N bytes num ficheiro, na resposta enviada pelo servidor, para além do número de bytes escritos nesse ficheiro são também devolvidos os atributos do ficheiro (p.e., tamanho, tempos de acesso e modificação do ficheiro).

Fluxo dos pedidos no NFS

Para demonstrar o comportamento do sistema NFS, vamos exemplificar o fluxo de um pedido de leitura (operação *read*) desde a aplicação no cliente até ao sistema de ficheiros utilizado no servidor para persistir os dados.

Como referido anteriormente, todas as operações utilizam *file handles* para identificarem os ficheiros. Então, antes da operação *read* ser realizada é preciso localizar o ficheiro de onde vão ser lidos os dados, e obter o respetivo *file handle*. O primeiro *file handle*, da diretoria exportada pelo servidor, é obtido durante a montagem do sistema de ficheiros no cliente (Figura-5-①). O *file handle* do ficheiro de onde vão ser lidos os dados, é obtido durante a chamada de sistema *open* realizada pela aplicação para abrir o ficheiro. Como os ficheiros são localizados relativamente a um *file handle* inicial, através de uma sequência de operações de *lookup*, podem ser necessários vários pedidos (Figura-5-②③) para obter o identificador de um ficheiro. Depois de obtido o identificador do ficheiro segue-se a operação de leitura (Figura-5-④).

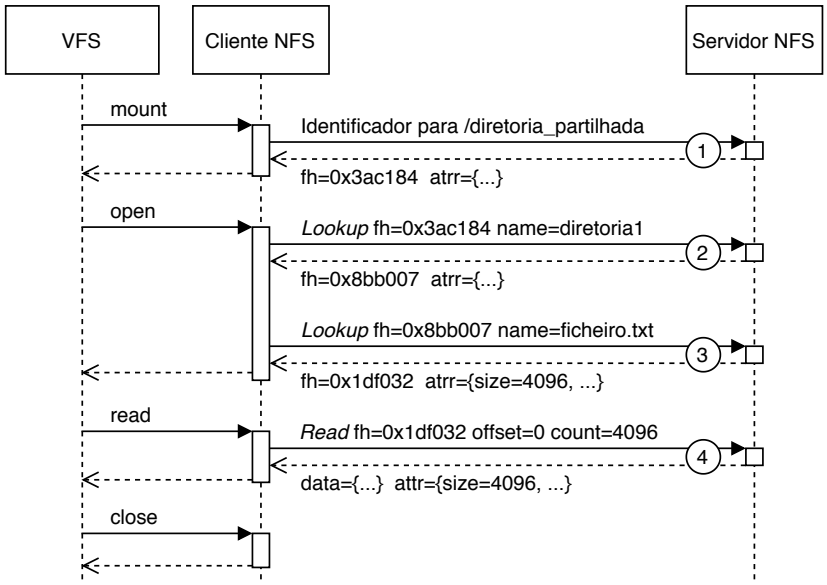


Figura 5: Fluxo de operações entre os módulos cliente NFS e servidor NFS.

O fluxo da operação inicia-se com uma aplicação a submeter uma operação de leitura para o VFS através da chamada de sistema *read*. O VFS, após validar o caminho indicado na operação, reencaminha o pedido para o módulo *kernel* cliente NFS. Este módulo envia então um pedido de leitura (operação *read* do protocolo NFS) através da rede para o servidor NFS, e espera pela resposta.

O pedido chega ao servidor através do módulo *kernel* servidor NFS. Analisando o *file handle* recebido no pedido enviado pelo cliente, o módulo servidor NFS identifica que ficheiro que pretende ser lido pelo cliente, e submete uma operação de leitura para o VFS sobre esse mesmo ficheiro. O VFS envia a operação para o sistema de ficheiros utilizado para persistir os dados (p.e., Ext4, Btrfs), que por sua vez lê os blocos pretendidos pelo cliente do disco.

Depois de lidos os dados do disco no servidor, a resposta segue o caminho oposto do pedido, passando pela rede até chegar à aplicação que submeteu o pedido no VFS do cliente.

2.3 Discussão

O contínuo crescimento do volume e complexidade de dados leva à constante evolução dos sistemas de ficheiros. Contudo, a complexidade de integração de novas funcionalidades e falta de suporte contínuo, leva a que muitos destes sistemas de ficheiros não sejam adotados. A solução para este problema passa pelos sistemas de ficheiros empilháveis, que permitem dotar sistemas de ficheiros já existentes com novas funcionalidades implementas sob a forma de camadas independentes. O principal problema destes sistemas é o seu desenvolvimento

em *kernel*, que traz vários inconvenientes, nomeadamente, dificuldade na portabilidade entre sistemas operativos e demora no desenvolvimento.

Para colmatar esses problemas surge o SafeFS, um sistema que utiliza a plataforma FUSE para desenvolver sistemas de ficheiros empilháveis. Este sistema permite que administradores e utilizadores de sistemas configurem/personalizem o sistema de armazenamento de acordo com as suas necessidades, através de camadas de funcionalidade desenvolvidas no espaço de utilizador. Com estas características, o SafeFS apresenta-se como estado da arte no que diz respeito aos sistemas de ficheiros modulares.

Relativamente aos sistemas de ficheiros remotos, o NFS fornece uma arquitetura baseada no modelo cliente-servidor, e permite aceder e armazenar ficheiros remotos de forma transparente. Ou seja, as aplicações que utilizam o NFS não precisam de ser modificadas, nem necessitam de utilizar bibliotecas adicionais para armazenar e aceder a ficheiros remotos.

Contudo surgem dois problemas. Por um lado, embora o SafeFS seja modular e extensível, não possibilita o armazenamento remoto. Por outro lado, embora o NFS possibilite armazenamento remoto, é difícil estendê-lo com novas funcionalidades e torná-lo adaptável para diferentes casos de uso, uma vez que está desenvolvido em *kernel*.

Desta forma, esta dissertação pretende avançar o estado da arte atual apresentado o RSafeFS, um sistema de ficheiros que conjuga as primitivas de modularidade e flexibilidade do SafeFS com as capacidades de armazenamento remoto do NFS.

3

ARQUITETURA

Neste capítulo é apresentado o sistema de ficheiros remoto RSafeFS, que propõe uma arquitetura modular, flexível, e extensível, para permitir acesso e armazenamento remoto em sistemas Unix. Esta arquitetura visa melhorar as limitações dos sistemas de ficheiros modulares e sistemas de ficheiros remotos. Nomeadamente, relativamente ao primeiro tipo de soluções, o sistema RSafeFS acrescenta a funcionalidade de operações remotas. Enquanto que no sistema SafeFS o armazenamento era feito localmente, o RSafeFS permite que múltiplos clientes, cada um configurado de acordo com o seu caso de uso (i.e., configurações das camadas) comuniquem com um servidor remoto para aceder e armazenar ficheiros. Relativamente ao segundo tipo de sistemas, enquanto fornecendo as mesmas funcionalidades de serviço remoto que o NFS, o RSafeFS acrescenta um ambiente modular e flexível, permitindo a configuração dos módulos cliente e servidor de forma a atingir requisitos de diferentes aplicações (p.e., desempenho, segurança, redução de espaço de armazenamento).

3.1 ARQUITETURA DO RSAFEFS

O RSafeFS é um sistema de ficheiros remoto que alavanca as propriedades de modularidade e flexibilidade do SafeFS, de forma a permitir o desenvolvimento de sistemas de ficheiros remotos ajustados aos requisitos das aplicações. De forma a possibilitar a construção de sistemas de ficheiros remotos e flexíveis, o desenho do sistema RSafeFS assenta sobre as seguintes propriedades:

EXTENSÍVEL deve permitir adicionar novas camadas de funcionalidade, tanto no cliente como no servidor, ao contrário do SafeFS onde esta propriedade apenas é oferecida no cliente. Ainda, a camada de comunicação deve permitir implementar diferentes protocolos de comunicação.

FLEXÍVEL deve ser possível configurar a pilha de camadas de acordo com diferentes objetivos, tanto ao nível do cliente como do servidor. Por exemplo, um administrador de sistemas deve poder configurar o servidor de acordo com objetivos orientados ao sistema que está a oferecer. Isto é, se o sistema estiver a ser utilizado para fazer *backup*, deverá

querer, por exemplo, redução de espaço, cifragem e redundância. No caso do cliente, este também deve poder ser configurado com os objetivos pretendidos pelo mesmo, por exemplo, cifragem para segurança e *caching* para melhorar o desempenho.

TRANSPARENTE deve ser transparente, como qualquer sistema de ficheiros baseado em FUSE, tanto no cliente como no servidor. Garantindo assim que as aplicações que utilizam a interface POSIX para aceder e armazenar ficheiros não necessitem de ser modificadas, e que não precisem de utilizar bibliotecas adicionais para utilizarem o RSafeFS.

COMPATÍVEL deve permitir a integração de sistemas de ficheiros baseados em FUSE como camadas independentes, uma vez que estende o SafeFS.

Este sistema, tal como o SafeFS, é um sistema que segue os princípios de armazenamento definido por *software*. Em particular, em vez de construir mecanismos monolíticos e acopulados, o RSafeFS parte estes mecanismos em camadas e permite que os sistemas de ficheiros (por si construídos) possam ser ajustados e programados segundo um conjunto de objetivos.

A arquitetura do novo sistema, apresentada na Figura 6, é baseada no modelo cliente-servidor, e apresenta três componentes principais: o cliente, o servidor e a camada de comunicação.

O cliente RSafeFS é responsável por enviar os pedidos realizados pelas aplicações para o servidor, mas antes disso aplica as camadas de funcionalidades (p.e., *caching*, cifragem) que constituem a instância do cliente. Analogamente ao SafeFS, e como em outros sistemas de ficheiros baseados em FUSE, o cliente RSafeFS processa as operações intercetadas pelo módulo FUSE do *kernel*. Cada operação intercetada é então processada pelas várias camadas, que tal como no SafeFS podem ser empilhadas por qualquer ordem. Para garantir o acesso e armazenamento remoto, a última camada deve corresponder a uma camada que permita comunicar com o servidor.

O servidor RSafeFS é responsável por receber os pedidos dos clientes (RSafeFS), processá-los através das camadas de processamento (p.e., compressão, deduplicação) e armazenamento que compõem a sua instância, e depois responder aos clientes.

A camada de comunicação do RSafeFS é composta dois componentes: o cliente RPC e o servidor RPC. Como os nomes dos componentes indicam, estes componentes comunicam através de chamadas de procedimentos remotos, porque, tal como no NFS, este mecanismo permite simplificar o desenvolvimento de serviços remotos. O servidor RPC está na instância do RSafeFS que atua como servidor, e o cliente RPC na que atua como cliente. Os pedidos que chegam ao cliente de RPC são traduzidos para operações do protocolo de comunicação estabelecido, e depois são enviadas para o servidor de RPC. Este protocolo define o conjunto de chamadas de procedimento remoto que permitem ao cliente efetuar as operações da API do FUSE na máquina servidor.

A integração da componente cliente RPC no RSafeFS é realizada através do desenvolvimento de uma camada que encapsula o comportamento do cliente de RPC. A camada cliente RPC pertence à categoria das camadas de armazenamento pois não comunica com outras camadas através da API do FUSE, precisando assim de ser sempre a última camada quando utilizada numa instância do RSafeFS. Esta camada, é então responsável por receber os pedidos da camada superior, prepará-los de forma a serem enviados para o servidor, e esperar pelas respostas vindas deste, entregando-as depois à camada superior. Antes da resposta ser entrega à camada superior, o cliente RPC transforma a resposta recebida do servidor na correspondente resposta segundo a API do FUSE.

Como referido anteriormente, o servidor RPC está encarregue de receber os pedidos dos clientes. Ao contrário da instância cliente RSafeFS em que o processamento das camadas é gerado pela biblioteca FUSE, na instância servidor RSafeFS esse processamento deve ser gerido pela componente servidor de RPC, invocando as *callbacks* registadas na primeira camada dessa instância. Assim, esta componente depois de receber os pedidos dos clientes tem de construir as operações correspondentes aos pedidos, cumprindo a API do FUSE, para estas poderem ser transmitidas pelas camadas seguintes do servidor. Depois de processadas pela restantes camadas o servidor prepara a resposta obtida e envia-a ao cliente através do protocolo de comunicação estabelecido.

O RSafeFS utiliza também a noção de *drivers*, introduzida pelo SafeFS, para promover a reutilização de camadas e a modularidade. Por exemplo, neste sistema, esta noção de *drivers* é utilizada para permitir configurar clientes e servidores RPC com diferentes protocolos de comunicação a serem utilizados. Desta forma, é possível dotar o sistema com novos protocolos de comunicação a serem escolhidos consoante os requisitos das aplicações e de instalação.

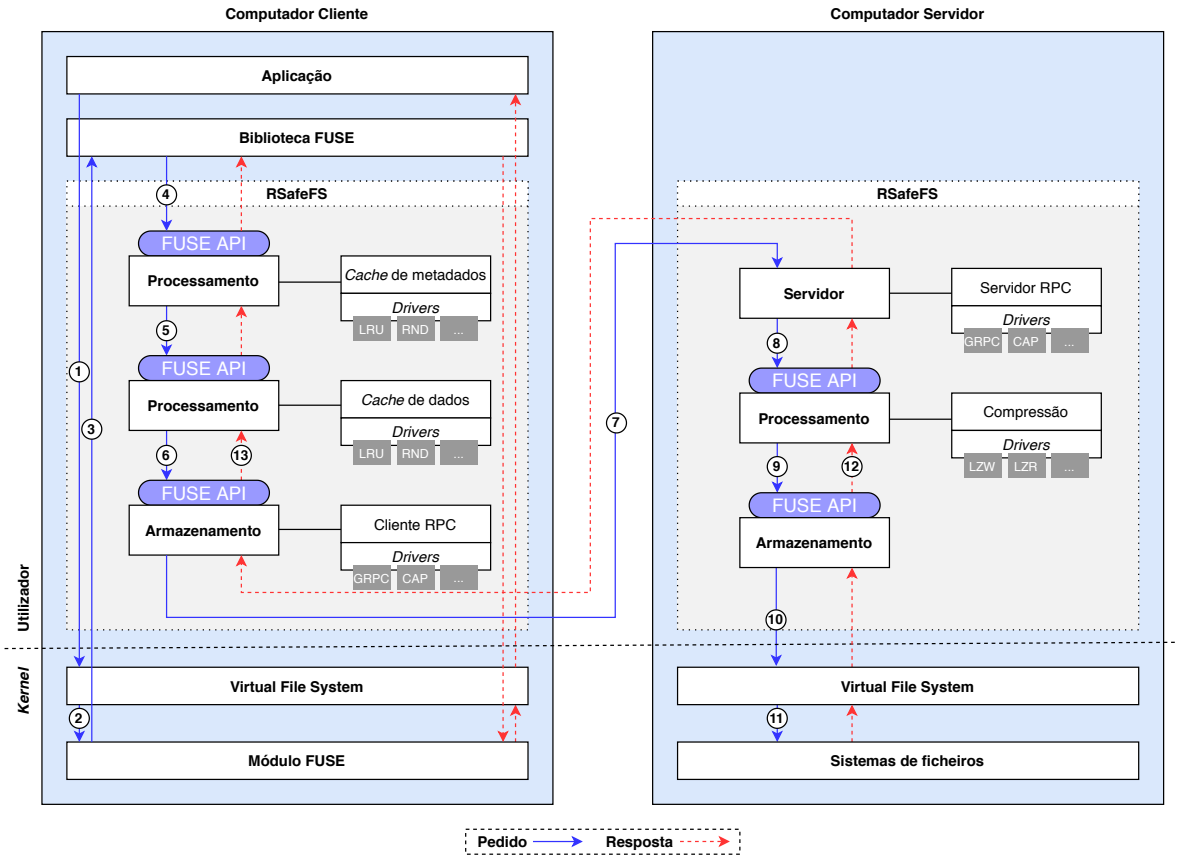


Figura 6: Arquitetura do RSafeFS.

3.2 FLUXO DAS OPERAÇÕES NO RSAFEFS

Para demonstrar o comportamento do sistema RSafeFS, vamos exemplificar o fluxo de um pedido desde a aplicação do cliente até ao sistema de ficheiros local no servidor. Para tal, consideremos uma operação de leitura (*read*) no sistema RSafeFS composto pelo cliente e servidor. O cliente compreende duas camadas de processamento, nomeadamente uma *cache* de metadados e uma *cache* de dados. Estas camadas procuram melhorar o desempenho do sistema reduzindo o número de pedidos a serem enviados para o servidor. Para isso, mantêm informação (dados e metadados) em memória para responder aos pedidos das aplicações. O servidor compreende uma camada de processamento, nomeadamente uma camada de compressão que reduz o espaço de armazenamento necessário para persistir os dados.

O fluxo inicia-se com a submissão de um pedido de leitura (*read*) no VFS por parte de uma aplicação (Figura-6-①). De seguida, o VFS verifica e valida o caminho do pedido, reencaminhando-o depois para o módulo FUSE (*kernel*) (Figura-6-②), que por sua vez irá redirecionar o pedido para a biblioteca FUSE no espaço de utilizador (Figura-6-③).

Neste ponto, a operação entra em contacto com a primeira camada do RSafeFS, no módulo cliente (Figura-6-④). A primeira camada do RSafeFS no módulo cliente, como referido anteriormente, é uma *cache* de metadados. Como o pedido realizado pela aplicação é uma operação orientada a dados, e esta camada é orientada a metadados, nesta camada não é feito nenhum processamento, sendo o pedido imediatamente enviado para a próxima camada (Figura-6-⑤). Nesta camada, cuja funcionalidade é fazer *caching* de dados, é primeiro verificado se os dados se encontram em memória: se for o caso, a camada retorna imediatamente (uma vez que os dados já se encontravam em memória); caso contrário continua o seu caminho, avançando para a próxima camada (Figura-6-⑥). Esta camada, que encapsula o cliente de RPC, tem como função serializar os dados de modo a identificar a operação a ser realizada no servidor, e efetuar a chamada do procedimento remoto. Durante esta fase, o pedido é enviado pela rede entre o cliente e o servidor (Figura-6-⑦).

O pedido chega ao módulo servidor do RSafeFS através do servidor de RPC, onde é feita a desserialização do pedido, de acordo com a chamada remota invocada pelo cliente, e transformado no correspondente pedido de leitura, seguindo a API do FUSE. O pedido é então enviado para a primeira camada de processamento no servidor (Figura-6-⑧), ou seja, para a camada de compressão. A camada de compressão, antes de enviar o pedido para a seguinte camada (Figura-6-⑨), ajusta o tamanho de dados pedidos para a leitura, pois os dados foram comprimidos quando foram escritos. A camada de armazenamento submete o pedido de leitura para o VFS (Figura-6-⑩), que reencaminha o pedido para um sistema de ficheiros local (Figura-6-⑪), onde os dados são lidos do disco.

Depois de obtida a resposta ao pedido de leitura, esta é propagada no sentido inverso ao descrito anteriormente, ou seja, passando pela rede, e pelas várias camadas de processamento, até chegar à aplicação. O servidor de RPC terá então de serializar a resposta, e a camada cliente de RPC de realizar o processo contrário. Durante este caminho inverso, as camadas poderão, se necessário, realizar processamento. Por exemplo, no lado do servidor, a camada de compressão (Figura-6-⑫) fará a descompressão dos dados; no lado do cliente, a *cache* de dados (Figura-6-⑬) mantém em memória os dados que fazem parte da resposta ao pedido de leitura.

As operações de escrita são tratadas de forma semelhante, isto é, o fluxo destas operações é análogo ao das operações de leitura, apesar de um processamento diferente por parte das camadas. Por exemplo, no cliente RSafeFS, enquanto que na camada de *caching* de dados, as operações de leitura procuram ser respondidas por dados mantidos em memória, nas operações de escrita não é realizado qualquer processamento; no servidor RSafeFS, a camada de compressão realiza o processamento contrário ao realizado na operação leitura, ou seja, em vez de realizar a descompressão de dados comprime-os, segundo a implementação do *driver*.

4

IMPLEMENTAÇÃO

Conforme explicado no capítulo 3, o sistema RSafeFS permite construir sistemas de ficheiros remotos com propriedades modulares, flexíveis e adaptáveis a diferentes casos de uso. Neste capítulo, será feita uma descrição detalhada da implementação dessa arquitetura.

Inicialmente é feita uma descrição da implementação global do sistema RSafeFS. De seguida, são apresentadas as *frameworks* de RPC que foram integradas como *drivers* na camada de comunicação do sistema e é explicado o seu funcionamento em detalhe. Ainda, é feita uma descrição detalhada de implementação de outras camadas de processamento consideradas no RSafeFS, e que foram utilizadas na avaliação experimental do protótipo. Todas as implementações aqui apresentadas foram desenvolvidas em C++.

4.1 RSAFEFS

Quando uma instância RSafeFS (cliente ou servidor) é inicializada, é necessário compor a pilha de camadas que a constituem. Este processo necessita que cada camada, além de implementar as operações (ou um subconjunto das operações) da API do FUSE, implemente também mais duas funções: *init* e *clean*.

A função *init* é invocada na inicialização do sistema, e serve para inicializar uma camada, ou seja, inicializa metadados, carrega configurações, e especifica para que camada deve enviar/redirecionar os seus pedidos. Esta função recebe como argumentos: uma estrutura com as operações de *callback* definidas pela camada inferior, e uma estrutura com metadados para parametrizar/configurar a camada. As operações definidas na estrutura de *callbacks* são utilizadas pela camada, durante a execução do sistema, para invocar as operações da camada inferior, fazendo assim, com que os pedidos sejam transmitidos entre camadas.

A função *clean* é invocada quando o sistema termina, e serve para libertar recursos associados à camada, por exemplo, para libertar estruturas de dados alocadas em memória.

Na montagem do sistema de ficheiros no lado do cliente, e na inicialização do servidor RSafeFS é interpretado um ficheiro de configuração passado como argumento à aplicação que inicia o sistema. O ficheiro de configuração é utilizado para definir as camadas utilizadas no cliente e servidor, e quais as definições que cada uma destas assume. Este ficheiro segue

o formato *INI*, que cumpre uma estrutura baseada em secções. A primeira secção do ficheiro define as camadas que compõem uma instância, e a ordem em que as camadas são declaradas nesta secção define a disposição das camadas do sistema. Assim, para uma correta disposição, no ficheiro de configuração devem ser declaradas primeiro as camadas inferiores. No resto das secções do ficheiro são configuradas cada uma das camadas, de acordo com as suas definições.

O Excerto 4.1 apresenta um exemplo de um ficheiro utilizado para configurar um cliente RSafeFS. No ficheiro de configuração do cliente são declaradas duas camadas, apresentadas na primeira secção do ficheiro (secção *layers*). A primeira camada declarada (*rpc_client*) compreende o cliente RPC, e a segunda (*metadata_cache*) uma *cache* de metadados. Nas restantes secções estão especificadas as definições da cada camada. Por exemplo, para a camada que compreende o cliente de RPC é especificado que o endereço do servidor de RPC é *10.0.0.1:50051*, e que o cliente deve utilizar modelo de operação assíncrono com a *framework* de comunicação gRPC.

```
[layers]
rpc_client = 0
metadata_cache = 1

[rpc_client]
server_address = 10.0.0.1:50051
framework = grpc
mode = async

[metadata_cache]
time_out = 30
replacement = lru
```

Excerto de Código 4.1: Exemplo de um ficheiro de configuração de um cliente RSafeFS, com uma camada de processamento

Um servidor RSafeFS é configurado da mesma forma que um cliente RSafeFS. O Excerto 4.2 apresenta um exemplo de um ficheiro utilizado para configurar uma instância servidor. Nesta configuração, o servidor RSafeFS não contempla nenhuma camada de processamento. Este apenas é composto por uma camada de armazenamento local, configurada na secção *local* com o caminho da diretoria para onde vão ser reencaminhados os pedidos dos clientes. Por sua vez, o servidor RPC é configurado na secção *rpc_server* através de quatro parâmetros, nomeadamente com o endereço que os clientes devem estabelecer ligação, a *framework* de comunicação utilizada, e através de dois parâmetros relacionados com o números de *threads* que devem atender os pedidos dos clientes.

```
[layers]
local = 0
rpc_server = 1

[local]
path = /exported/directory

[rpc_server]
server_address = *:50051
framework = grpc
n_queues = 2
n_pollers_per_queue = 2
```

Excerto de Código 4.2: Exemplo de um ficheiro de configuração para um servidor RSafeFS, apenas com uma camada de armazenamento

Para garantir a comunicação entre clientes e servidores RSafeFS foram desenvolvidas duas camadas, nomeadamente cliente RPC e servidor RPC. No âmbito desta dissertação, foram também desenvolvidas camadas de *caching* de dados e metadados, que visam, sobretudo, melhorar o desempenho dos clientes, no entanto, devido à modularidade do RSafeFS, as camadas desenvolvidas são compatíveis de serem utilizadas tanto no cliente como no servidor. No sentido de promover a reutilização e a modularidade destas camadas, os algoritmos de substituição, que definem quais os objetos a substituir quando estas *caches* estiverem cheias, foram implementados segundo a noção de *drivers* do RSafeFS.

4.2 FRAMEWORKS DE RPC

Para permitir acesso e armazenamento remoto no sistema RSafeFS, é necessário assegurar a camada de comunicação entre os módulos cliente e servidor. Para isso, assumimos que a comunicação seria efetuada através de RPC, porque, como referido anteriormente, este modelo de comunicação permite simplificar a definição, organização e implementação de serviços remotos. Em detalhe, o sistema de RPC subjacente oculta detalhes importantes, ao nível da codificação e descodificação de parâmetros e resultados, e da preservação da semântica necessária para a invocação de um procedimento. Ainda, esta abordagem oferece suporte direto ao modelo cliente-servidor pretendido pelo RSafeFS, com servidores que oferecerem um conjunto de operações através de uma interface de serviço e clientes que invocam essas operações como se fossem locais. Após uma revisão das *frameworks* de RPC atualmente disponíveis pela comunidade, selecionamos a gRPC[15] e a Cap'n Proto RPC [3].

O gRPC é uma *framework* de RPC de alto de desempenho, desenvolvida pela Google, sobre o protocolo de comunicação HTTP/2 [28]. Esta *framework* é uma ferramenta RPC

universal com continuo suporte da comunidade (i.e., mantém-se sempre atual e em constante desenvolvimento) e pode ser ser utilizada com múltiplas linguagens de programação (p.e., C++, Go, Java, Python). Além da Google, que utiliza o gRPC para desenvolver os seus produtos, nomeadamente os produtos baseados na nuvem e APIs externas (p.e., *Google Cloud Pub/Sub API*, *Google Bigtable API*, *Google Speech-to-Text API*), o gRPC é também utilizado por várias empresas (p.e., Netflix, Dropbox, Spotify, Cisco) para desenvolverem os seus produtos.

Esta *framework* permite construir serviços com o modelo de comunicação síncrona e assíncrona, e para além do habitual modelo pedido-resposta suporta ainda fluxos bidirecionais, permitindo, desta forma, cenários em que cliente e servidor enviam fluxos de dados de forma assíncrona. O gRPC utiliza por padrão Protocol Buffers [22] como Interface Definition Language (IDL) e como formato de dados das mensagens trocadas entre cliente e servidor. Protocol Buffers são um mecanismo extensível, neutro em termos de linguagem e de plataforma, para a serialização de dados estruturados. Quando utilizado no gRPC permite que a comunicação entre o cliente e o servidor possa ser eficiente serializada, permitindo assim, uma comunicação eficiente. Além destas características, o gRPC oferece também mecanismos como, autenticação, cifragem, balanceamento de carga e monitorização, que não são explorados no âmbito desta dissertação.

O Cap'n Proto é uma *framework* de RPC, desenvolvida pela Sandstorm, que tem como principal característica um mecanismo designado *time-travel*. Este mecanismo permite que os resultados de uma chamada RPC sejam devolvidos instantaneamente, mesmo antes do servidor receber a pedido inicial. Contudo, os resultados só podem ser utilizados como parte de um novo pedido a ser enviado para o servidor. Se o cliente quiser utilizar os resultados, sem ser para encadear procedimentos remotos, tem de esperar que estes sejam feitos no servidor. Esta *framework* utiliza para representação de dados um formato neutro em memória, que não envolve etapas de codificação/descodificação. Permitindo assim, acesso direto a dados serializados.

Porém, dadas as características do FUSE, não é possível tirar proveito deste encadeamento de procedimentos oferecido pelo Cap'n Proto, porque as operações FUSE estão dependentes umas das outras. Por exemplo, uma operação de leitura de dados só ocorre se a operação para abrir o ficheiro foi bem sucedida. Ainda, do lado do cliente só é possível concluir uma operação se o servidor realmente lhe responder.

Contudo, antes de passarmos à integração imediata destas *frameworks* na camada de comunicação do RSafeFS, decidimos que deveria ser feita uma primeira fase de avaliação quanto à sua funcionalidade e desempenho. Assim, inicialmente foram desenvolvidas integrações de FUSE com diferentes *frameworks* de RPC e avaliados os seus desempenhos para um conjunto extenso de testes, que serão apresentados no capítulo 5. Os resultados para estas integrações revelam que a integração de FUSE com o gRPC obteve, na maioria dos

testes, melhor desempenho que a integração com Cap'n Proto. Esta diferença de desempenho deve-se à implementação *single-threaded* da integração com Cap'n Proto e da utilização de fluxos bidirecionais na integração com gRPC.

Desta forma, na implementação da componente de comunicação no RSafeFS, utilizamos a *framework* gRPC. Importa referir que, embora tenhamos utilizado gRPC para construir a componente de comunicação, o desenho e arquitetura modular do RSafeFS permite a integração de outras *frameworks* de RPC. As próxima secções (4.3 e 4.4), relativas aos componentes de comunicação, apresentam a implementação tendo em conta as características do gRPC.

4.3 Camada de comunicação RSafeFS - Servidor

Como referido no capítulo anterior, a camada de comunicação do servidor RSafeFS é responsável por receber os pedidos dos clientes, e enviar-lhes as respostas depois das camadas que constituem a instância do servidor concluírem as respetivas operações. Para garantir estas propriedades, esta camada é implementada encapsulando um servidor de RPC. Assim, o servidor de RPC recebe os pedidos dos clientes RSafeFS e simula o ambiente criado pela biblioteca FUSE para as camadas que compõem a instância do servidor RSafeFS, invocando as operações registadas na primeira camada.

Esta componente implementa o serviço definido pelo protocolo de comunicação estabelecido entre cliente e servidor. Este protocolo define as chamadas de procedimento remoto das operações FUSE, bem como a estrutura das várias mensagens pedido e reposta. O Excerto de Código 4.3 apresenta o protocolo de comunicação definido para operação *read*, através da sintaxe *Protocol Buffers*. Para esta operação, o serviço define que o cliente envia uma mensagem do tipo *ReadRequest* e que o servidor responde através de uma mensagem do tipo *ReadReply*.

Como anteriormente apresentado na Tabela 1, aquando da apresentação da API do FUSE, a operação *read* recebe como argumentos: o caminho para o ficheiro de onde vão ser lidos os dados, o tamanho dos dados a serem lidos, um *offset* da leitura em relação ao início do ficheiro, e uma estrutura do FUSE (*struct fuse_file_info*). Para esta operação ser realizada no servidor, estes parâmetros têm de ser recebidos no pedido enviado para o servidor. Assim, a mensagem que define o pedido *ReadRequest* é composta pelo caminho do ficheiro (*path*), pelo tamanho dos dados a serem lidos (*size*), pelo *offset*, e pela mensagem *StructFuseFileInfo*, que mapeia estrutura recebida como argumento da operação *read* do FUSE. No sentido contrário, na resposta ao pedido, a mensagem *ReadReply* é composta pelo resultado da operação de leitura (*result*), e pelo dados lidos por essa operação (*buf*).

```
service FuseOps {
    ...
    rpc Read (ReadRequest) returns (ReadReply) {}
    ...
}

message ReadRequest {
    string path = 1;
    uint64 size = 2;
    int64 offset = 3;
    StructFuseFileInfo info = 4;
}

message ReadReply {
    int32 result = 1;
    bytes buf = 2;
}

message StructFuseFileInfo {
    int32 flags = 1;
    ...
    uint64 fh = 10;
    uint64 lock_owner = 11;
}
```

Excerto de Código 4.3: Protocolo definido entre cliente e servidor para a operação *getattr* do FUSE

A comunicação (e respetiva implementação) é realizada através da API assíncrona do gRPC. Ao contrário da API síncrona, em que os pedidos são tratados com pouca transparência por um conjunto de *threads* internas do gRPC, onde efetivamente não é possível saber que *threads* estão a ser utilizadas para responder aos pedidos; a API assíncrona oferece um controlo preciso na gestão de execução de tarefas, permitindo, por exemplo, escolher quantas e que *threads* devem ser utilizadas para responder a cada operação do serviço, permitindo assim melhorar o desempenho do sistema.

Neste modelo de comunicação assíncrona os pedidos são recebidos pelo servidor através de uma *CompletionQueue* - uma fila de eventos - que notifica as *threads* responsáveis por responderem a estes pedidos. Depois de recebido um pedido, a *thread* responsável pela execução deste faz a sua desserialização, cria os argumentos esperados pela operação FUSE para esse tipo de pedido (p.e., o *buffer* para onde devem ser lidos dados numa operação de leitura) e invoca a correspondente operação da primeira camada do servidor RSafeFS. Depois desta camada responder ao servidor de RPC, dá-se a serialização da resposta que é

de seguida enviada para o cliente RSafeFS, que por sua vez implementa a camada cliente de RPC.

Esta componente da camada de comunicação pode ser configurada através do ficheiro de configuração do RSafeFS, onde é possível definir o número de *CompletionQueues* e de *threads* utilizadas pelo servidor de RPC. Permitindo assim ajustar o servidor RSafeFS aos recursos da máquina, bem como às cargas esperadas no sistema.

4.4 Camada de comunicação RSafeFS - Cliente

Como descrito na secção 3.2, a camada de comunicação de um cliente RSafeFS, envia pedidos (e espera pela sua resposta) por RPC. De modo a atingir diferentes garantias de coerência de dados e desempenho, nesta camada, foram implementados dois tipos de cliente, conforme indicado no resto desta secção.

4.4.1 *Cliente síncrono*

O cliente síncrono, limita-se a enviar os pedidos para o servidor e aguardar pela sua resposta. Na sua implementação é utilizado diretamente o *stub* que fornece as chamadas para os métodos implementados pelo servidor.

Uma vez que a camada de comunicação RSafeFS do lado do cliente, após a submissão de um pedido, precisa de esperar pela resposta do servidor (antes de responder à camada superior), esta é implementada com a API síncrona do gRPC.

As operações de escrita e leitura são realizadas através de fluxos bidirecionais do gRPC, que permitem obter melhor desempenho quando comparadas com as chamadas unidirecionais. Este ganho no desempenho é justificado pela eliminação da sobrecarga de criar um novo pedido HTTP/2 por cada chamada entre cliente e servidor, utilizando em vez disso uma ligação HTTP/2 já existente. Assim, para cada operação *open* sobre ficheiros distintos, o cliente abre um fluxo bidirecional, que só é fechado na operação *release*, operação esta que sinaliza que já não há referências para o ficheiro aberto. Enquanto o ficheiro estiver aberto todas as operações de leitura e escrita são feitas através deste fluxo.

4.4.2 *Cliente assíncrono*

Para a maioria das operações FUSE, o cliente assíncrono apresenta uma implementação semelhante à do cliente síncrono, apenas modificando a implementação das operações *write*, *flush* e *fsync*. Este cliente foi inspirado no cliente assíncrono do NFS, que atrasa o envio de operações de escrita para o servidor até que aconteçam determinados eventos (p.e., até um

limite de memória ocupada, ou a aplicação faça explicitamente operações que causem *flush* dos dados) [20].

No cliente implementado, os dados relativos às operações de escrita são armazenados temporariamente em memória conservando a ordem com que estas foram realizadas. Assim, para cada operação de escrita a camada cliente de RPC responde logo à camada superior indicando sucesso da operação depois de garantir que os dados foram armazenados em memória. Estes são mantidos em memória até que seja feito um pedido de persistência de dados para o servidor, que é realizado quando um dos seguintes eventos ocorre: operações *fsync* e *flush*, que são invocadas explicitamente pelas aplicações ou na invocação explícita ou implícita da operação *close*, que fecha um determinado descritor de ficheiro; atingido o limite máximo de memória que pode ser utilizada pelo cliente; ou, quando não ocorrem novas escritas durante um certo período de tempo estabelecido.

Quando os dados são enviados para o servidor, o cliente procura agrupar, tanto quanto possível, dados de várias escritas consecutivas sobre o mesmo ficheiro (até um determinado tamanho), diminuindo assim o número de mensagens transmitidas entre o cliente e o servidor.

O envio dos dados para o servidor é feito através da API assíncrona do gRPC, porque a API síncrona implica que a *thread* que faz a chamada do procedimento remoto fique bloqueada até que chegue uma resposta do servidor. Caso contrário, o acesso à memória do cliente (utilizada para armazenar os pedidos) ficaria bloqueado, não permitindo assim guardar em memória as novas escritas geradas pelas aplicações. Como consequência, esta abordagem tornar-se-ia num ponto de contenção no sistema e diminuiria o desempenho do sistema.

Quando a aplicação fecha um ficheiro, o cliente (assíncrono) do RSafeFS envia para o servidor pedidos de escrita que estejam pendentes, e espera que estes sejam persistidos pelo servidor. Isto permitir que o cliente reporte erros às aplicações, ocorridos no servidor durante as escritas, através da resposta à chamada de sistema *close*. Ainda, isto assegura, tal como no NFS, que com o este tipo de cliente as aplicações têm a garantia que depois de voltar a abrir um ficheiro as aplicações têm acesso a dados anteriormente modificados.

De forma a possibilitar o ajuste de diferentes parâmetros no modo de operação assíncrono, o RSafeFS exporta quatro configurações para este cliente, nomeadamente: a capacidade máxima de dados mantidos em memória; o tamanho máximo do bloco de dados a ser enviado para o servidor, que é criado no processo que procura agrupar escritas consecutivas do mesmo ficheiro; o número de *threads* que operam com a API assíncrona do gRPC; e ainda um fator relativo ao limite máximo de memória utilizada, que procura, através de operações de *flush* periódicas, diminuir o tempo em que a memória do cliente está bloqueada para garantir que o limite máximo de memória não é ultrapassado.

4.5 CAMADAS DE *caching*

De forma a otimizar o desempenho do sistema RSafeFS, e das aplicações que o usam, foram implementadas diferentes camadas de *caching*. Estas camadas, têm como objetivo reduzir o número de operações que chegam às camadas de armazenamento, mantendo para isso informação necessária para responder aos pedidos em memória. Desta forma, foram implementadas duas camadas de *caching*, uma orientada a metadados e outra a dados.

4.5.1 *Cache de Metadados*

A camada de *caching* de metadados procura reduzir o número de operações sobre metadados que chegam às camadas de armazenamento. Ao fazê-lo, o RSafeFS introduz vários benefícios: (1) se o pedido conseguir ser atendido pelos metadados armazenados em memória, evitamos a travessia do pedido pelas várias camadas de processamento e armazenamento, e claro, pela camada de comunicação; (2) quando utilizada no cliente, reduzimos o número de invocações submetidas ao servidor RSafeFS; (3) como consequência dos itens anteriores, e tal como apresentado no capítulo 5, há um impacto (positivo) significativo no desempenho do RSafeFS, quando comparado com outros sistemas de armazenamento remoto.

A leitura dos metadados é feita na operação *getattr*, que procura pelos atributos de um objeto no sistema de ficheiros. Esta operação é intercetada por esta camada, e os metadados que procura são verificados com os metadados armazenados em memória. Se o metadados estiverem na *cache* e ainda forem válidos, ou seja, desde a inserção ainda não foi ultrapassado um certo período de tempo, o pedido é respondido diretamente por esta camada. Se os metadados não estiverem na *cache*, ou se estiverem inválidos, o pedido é enviado para a seguinte camada de processamento. Depois de recebida a resposta da próxima camada, e no caso desta indicar sucesso na realização da operação, os metadados são inseridos na *cache* para esta poder atender de forma eficiente futuros pedidos.

Embora esta camada intercepte várias operações (p.e, *chmod*, *mkdir*, *rename*), a sua maioria é usada para manter coerência da *cache*, invalidando possíveis metadados associados aos alvos das operações. Por exemplo, assumamos a operação *create*, que é realizada quando uma aplicação pretende criar um ficheiro. Para garantir a coerência da *cache*, os metadados da diretoria onde este ficheiro vai ser criado são removidos da *cache*, pois estes metadados irão ser modificados caso esta operação tenha sucesso.

A gestão dos metadados da *cache*, quando são realizadas operações que modifiquem o seu conteúdo, pode ser feita de duas formas: esperar pela resposta da camada inferior e fazer a atualização aos metadados na própria *cache*; ou invalidar os metadados do objeto associado a esta operação, removendo-os da *cache*. Apesar da primeira solução poder apresentar melhor

desempenho, optou-se pela segunda solução, uma vez que no caso de haver múltiplos clientes, invalidar mais vezes a cache traz maior coerência entre *caches* de diferentes clientes.

Esta camada é configurada através de dois parâmetros, nomeadamente, a capacidade máxima, em bytes, da *cache*; e o limite máximo, em segundos, que cada conjunto de metadados é considerado válido após a sua inserção na *cache*. E ainda através de um *driver*, que permite escolher qual a política de substituição utilizada pela *cache* quando esta atinge a sua capacidade máxima.

Nesta *cache* de metadados, tal como acontece na *cache* do NFS [20], metadados modificados por outros clientes só são invalidados quando o tempo de validade dos metadados mantidos em memória expira, ou seja, alterações que ocorram no servidor durante esses intervalos não são detetados pelos clientes. Isto faz com que em certas situações, por exemplo, quando ocorrem operações concorrentes (de diferentes clientes) sobre o mesmo ficheiro, possa haver incoerência da *cache*. Para garantir outras formas de coerência da *cache*, que também não são garantidas pelo NFS, o servidor teria, por exemplo, de tomar a iniciativa de atualizar os clientes (que estão a fazer *caching*) com novos metadados, quando detetasse que estes tinham sido modificados. Para isto, o servidor necessitava de manter estado relativo aos clientes que mantêm uma ligação ativa com ele, bem como detetar possíveis falhas destes clientes.

Comparativamente à *cache* de atributos do NFS, que é atualizada por respostas que incluem os atributos do ficheiros, que como anteriormente observado (Tabela 2) são a maioria [39]; a *cache* de metadados desenvolvida para o RSafeFS não é atualizada desta forma, porque a comunicação entre camadas é realizada através da API do FUSE, onde os atributos dos ficheiros não integram as respostas aos pedidos.

4.5.2 *Cache de Dados*

A camada de *caching* de dados tem como objetivo reduzir o número de operações de leitura de dados enviadas paras as camadas de armazenamento mantendo em memória, durante um certo período de tempo, blocos de dados que já foram lidos e que poderão ser lidos num futuro próximo. Esta camada procura assim aumentar o débito das operações de leitura de dados, relaxando a coerência dos dados, nomeadamente no caso de ficheiros partilhados entre clientes.

Esta camada oferece as mesmas garantias de coerência *caching* de dados que o NFS [52], onde os dados mantidos em *cache* no cliente são revalidados na abertura de ficheiros. Esta revalidação é feita comparando o tempo de modificação do ficheiro no servidor, com o tempo de modificação associados aos blocos mantidos na *cache* no cliente.

Desta forma, a camada de *caching* de dados do RSafeFS interceta três operações, nomeadamente, as operações *open*, *read* e *release*. As operações de escrita não são intercetadas por esta camada, e não são servidas por esta *cache*, para garantir a sua persistência em disco. Durante

a intercepção da operação *open*, esta camada, para além de reencaminhar para a seguinte camada a operação para abrir o ficheiro, envia também uma operação *getattr* para obter o tempo de modificação do ficheiro presente nos atributos do ficheiro. Sendo depois este tempo de modificação do ficheiro mantido em memória na *cache* até que a operação *release* sobre o ficheiro seja realizada, operação esta que ocorre quando não há mais referências para ficheiros abertos.

A operação *read* intercetada - responsável pela leitura de dados -, tem como argumentos o caminho do ficheiro de onde deverão ser lidos os dados, o tamanho dos dados a serem lidos, e o *offset* da leitura em relação ao início do ficheiro. No processamento desta operação, antes da leitura do bloco pedido pela operação é necessário identificar e verificar se este existe na *cache*. Cada bloco é composto por um conjunto de bytes contíguos num ficheiro, que é identificado através do caminho do ficheiro a que pertence e do seu *offset* neste. Para eficientemente localizar cada bloco no ficheiro, é-lhe atribuído um identificador que é calculado através de divisão do *offset* pedido pelo tamanho máximo do bloco da *cache*. Por sua vez, como ilustrado na Figura 7, cada bloco na *cache* é identificado através da composição do identificador do ficheiro (caminho do ficheiro) com o identificador do bloco.

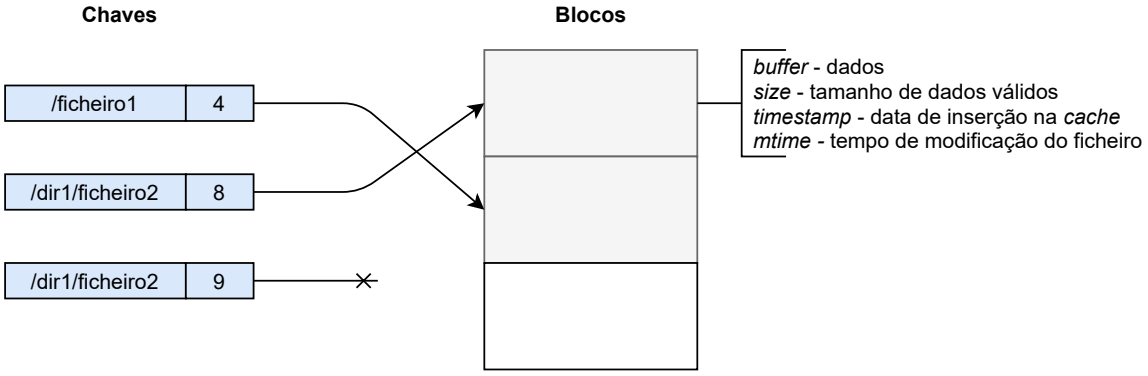


Figura 7: Mapeamento dos blocos na *cache* de dados.

Se o bloco estiver na *cache* for válido, ou seja se o tempo de modificação do bloco for igual ao tempo de modificação obtido na operação *open*, e se desde a inserção na *cache* ainda não foi ultrapassado um certo período de tempo, a operação de leitura é realizada lendo os dados necessários para responder ao pedido do bloco mantido em memória. Caso contrário, é necessário trazer para memória os dados persistidos em disco, relativos a esta operação. Neste caso, é enviada uma operação de leitura para a camada inferior, que, por sua vez, em vez de ler o tamanho pedido na operação de leitura é feita uma operação de leitura em que o tamanho é o bloco máximo de bloco da *cache*. Com isto, aproveitamos para ler (e armazenar em memória) dados adicionais que possam vir a ser utilizados por futuras operações.

Analogamente à *cache* de metadados, a *cache* de dados também permite ser configurada através de um *driver* que faz a gestão da substituição dos blocos quando esta atinge a sua

capacidade máxima. Ainda, esta camada também é configurável através de três parâmetros: o limite máximo de memória utilizada pela camada, em bytes; o tamanho dos blocos que são armazenados na *cache*; e o período em que os blocos são considerados válidos desde a sua inserção.

Contudo importa referir que, de forma análoga à *cache* do cliente NFS, quando esta camada de *caching* é utilizada num cliente RSafeFS, operações em simultâneo sobre ficheiros podem gerar incoerência desta *cache*. Por exemplo, se o cliente *A* após abrir um ficheiro verificar que os dados mantidos em *cache* ainda estão válidos, as futuras operações de leitura são respondidas pela *cache*. Entretanto, se o cliente *B* abrir o mesmo ficheiro e realizar operações de escrita neste, o cliente *A* vai ter a sua *cache* incoerente, que o levará a ler dados que já poderão ter sido modificados.

Por isso, caso sejam realizadas operações de leituras e escritas sobrepostas num único ficheiro, devem ser utilizados mecanismos de controlo de concorrência para evitar problemas de coerência da *cache*. Ainda, para mais rapidamente serem detetadas alterações dos ficheiros no servidor, a *cache* de dados do RSafeFS pode ser configurada com períodos de validade dos blocos mais pequenos.

4.5.3 *Políticas de substituição*

Uma *cache* considera-se eficiente se conseguir obter taxas de acerto elevadas, ou seja, se o objeto procurado for encontrado com muita frequência na *cache*. Diferentes políticas de substituição garantem diferentes níveis de eficiência. As políticas de substituição (ou algoritmos de substituição) definem que objeto deve ser removido da *cache* quando esta atinge a capacidade máxima, e se pretende adicionar um novo objeto. Assim, algoritmos que consigam prever qual o objeto que está na *cache* e que vai ser utilizado futuramente, conseguirão evitar que este seja removido, tornando a *cache* mais eficiente. Contudo, nem sempre é possível utilizar as políticas de substituição mais eficientes, pois estas normalmente necessitam de mais informação sobre a utilização dos objetos, que por sua vez, se traduz na utilização de mais recursos.

Desta forma, para ambas as *caches* (dados e metadados), foram implementadas duas políticas de substituição que diferem na utilização de recursos. A política de substituição aleatória seleciona aleatoriamente um objeto para ser removido, por isso, não necessita de manter qualquer informação sobre os objetos. A política de substituição menos utilizado recentemente mantém informação sobre os objetos que estão a ser utilizados, para depois remover o objeto que não é utilizado à mais tempo.

Nesta última política, a informação é gerida através de duas estruturas auxiliares, nomeadamente, com uma lista de identificadores de objetos armazenados na *cache*, e com uma estrutura que mapeia os identificadores dos objetos em iteradores para a lista. Quando

um objeto da *cache* é acedido, primeiramente, através do seu identificador é procurado o respetivo iterador da lista. Se este identificador for encontrado no mapa, com base no iterador, o identificador é removido da lista de identificadores. Depois o identificador é inserido no inicio da lista, e o respetivo iterador é inserido no mapa. Assim garante-se que os objetos utilizados mais recentemente estão no inicio da lista, desta forma, quando é necessário escolher um objeto para ser removido, retira-se o último identificador da lista, pois este é o objeto menos recentemente utilizado.

Para padrões de acesso aleatórios a política de substituição de menos utilizados recentemente não traz vantagem em relação à de substituição aleatória, contudo, quando os padrões de acesso das aplicações acedem com mais frequência a dados recentemente acedidos, esta política garante uma gestão mais eficiente da cache, possibilitando assim um melhor desempenho.

As políticas de substituição foram implementadas seguindo a noção de *drivers* do RSafeFS, procurando assim reutilizar código e manter a modularidade. A API dos *drivers* destas camadas definem três operações que permitem fazer a gestão das *caches* segundo diferentes políticas de substituição, e seguem as assinaturas apresentadas no Excerto de Código 4.4).

```
void touch(const T &t);
void remove(const T &t);
std::optional<T> select();
```

Excerto de Código 4.4: Assinatura das operações do driver

A operação *touch* informa o *driver* que um dado objeto da *cache* foi utilizado, podendo assim o *driver* fazer, se necessário, a gestão com base nos objetos que estão a ser utilizados. A operação *remove* informa o *driver* que um dado objeto foi removido da *cache*, devendo por isso sair da gestão do *driver*. Por fim, a operação *select* seleciona um objeto segundo a política de substituição utilizada pelo *driver*.

4.6 Camada de armazenamento local

Para uma comparação justa com o NFS, que irá ser realizada no próximo capítulo, foi necessário desenvolver uma camada de armazenamento que ofereça as mesmas garantias que o NFS. Isto é, que a camada de armazenamento garanta que os dados modificados pelas operações são persistidos antes de responder à camada superior, garantindo assim que, quando utilizada pelo servidor RSafeFS, este garante ao cliente que não há perdas de informação no caso de falhas (temporárias) do servidor.

Esta garantia é uma característica importante do servidor NFS, pois com o seu incumprimento o cliente não teria forma de saber quando seria seguro descartar dados associados às operações pedidas. Isto é, se o servidor não persistisse os dados antes de responder ao

cliente, o cliente teria de manter em memória dados que lhe permitam retransmitir o pedido, caso ocorra alguma falha do servidor. Por exemplo, na operação *write* onde são escritos dados num ficheiro, se o servidor não garantisse que os dados são persistidos antes de responder ao cliente, este, para garantir que não haveria perdas de dados no caso de falhas temporárias do servidor, teria de manter em memória dados associados à operação. Ainda, o cliente não saberia quando poderia remover de memória esses dados sem recorrer a outras operações do servidor.

As operações do protocolo NFS que implicam que o servidor persista os dados modificados são: *write* (quando definida com a opção *file_sync*), *create*, *mkdir*, *symlink*, *mknod*, *remove*, *rmdir*, *rename*, *link*, e *commit* [32]. Desta forma, as operações análogas FUSE: *mknod*, *mkdir*, *unlink*, *rmdir*, *symlink*, *rename*, *link*, *create* e *write* foram implementadas de forma a oferecerem as mesmas garantias. Assim, quando as respostas a estas operações são enviadas para os clientes, é garantido que as operações correspondentes foram concluídas no servidor, e que os dados modificados foram persistidos em disco.

5

AVALIAÇÃO EXPERIMENTAL

Definida a arquitetura e implementação do protótipo do RSafeFS, é necessário realizar uma avaliação completa e detalhada, de forma a perceber o possível impacto na construção de sistemas de armazenamento remotos, bem como eventuais gargalos no desempenho do RSafeFS e do próprio FUSE. Deste modo, neste capítulo é feita uma análise dos resultados obtidos num conjunto de micro e macro testes, percebendo assim qual o desempenho do sistema, a sua aplicabilidade em ambientes realistas, e as suas limitações.

5.1 METODOLOGIA

Para se efetuar uma avaliação completa do desempenho foram selecionados um conjunto de testes micro e macro. A descrição de cada um dos testes utilizado durante a avaliação experimental é apresenta na Tabela 3.

Os micro testes focam-se em cargas de trabalho regulares que submetem sempre o mesmo tipo de operações, e podem ser divididos em duas categorias relativamente ao tipo de operações submetidas: dados ou metadados. Os micro testes orientados a dados realizam operações de escrita ou leitura, sobre um determinado tipo de acesso, podendo este ser aleatório ou sequencial. Os micro testes que submetem operações orientadas a metadados realizam operações de criação (*create*), remoção (*delete*), leitura (*read*), e verificação de estado (*stat*) de ficheiros. Relativamente aos macro testes, estes incluem um conjunto de operações sobre dados e metadados, com diferentes tipos de acesso, e visam representar casos de uso reais, nomeadamente servidores e-mail (*mail-server*), web (*web-server*), e ficheiros (*file-server*).

De forma a simplificar a identificação das cargas de trabalho irão ser utilizadas as seguintes siglas: *rand* e *seq* para representar padrões de acesso aleatório e sequenciais, respetivamente; *read* e *write* para operações de leitura e escrita, respetivamente; e *Xth*, *Yf* e *Zk* para representar, respetivamente, o número de *threads* utilizadas, o número de ficheiros acedidos, e o tamanho de bloco utilizada pelas as operações de E/S.

Tabela 3: Descrição das cargas de trabalho.

Tipo		Nome	Descrição
micro	dados	*seq-read-Xth-Xf*	*X threads* [1, 16] lêem sequencialmente *X* ficheiros [1, 16]; quando *X* é 1, 1 *thread* lê sequencialmente 1 ficheiro pré-alocado de 32 GiB; quando *X* é 16, cada uma das 16 *threads* lê sequencialmente de um ficheiro distinto pré-alocado de 2 GiB.
		rand-read-Xth-Xf	*X threads* [1, 16] lêem aleatoriamente *X* ficheiros [1, 16]; quando *X* é 1, 1 *thread* lê aleatoriamente 1 ficheiro pré-alocado de 32 GiB, quando *X* é 16, cada uma das 16 *threads* lê aleatoriamente de um ficheiro distinto pré-alocado de 2 GiB.
		seq-write-Xth-Xf	*X threads* [1, 16] escrevem sequencialmente *X* ficheiros [1, 16]; quando *X* é 1, 1 *thread* cria e escreve sequencialmente 1 ficheiro de 32 GiB; quando *X* é 16, cada uma das 16 *threads* escreve sequencialmente um ficheiro distinto de 2 GiB.
		rand-write-Xth-Xf	*X threads* [1, 16] escrevem aleatoriamente *X* ficheiros [1, 16]; quando *X* é 1, 1 *thread* escreve aleatoriamente para 1 ficheiro pré-alocado de 32 GiB; quando *X* é 16, cada uma das 16 *threads* escreve aleatoriamente para um ficheiro distinto pré-alocado de 2 GiB.
	metadados	*create-Xth*	*X threads* [1, 16] criam 1 milhão de ficheiros de 4 KiB distribuídos por 1000 diretorias.
		delete-Xth	*X threads* [1, 16] removem 150 mil ficheiros pré-alocados de 4 KiB distribuídos por 150 diretorias.
		read-Xth	*X threads* [1, 16] lêem 750 mil ficheiros pré-alocados de 4 KiB distribuídos por 25 diretorias.
		stat-Xth	*X threads* [1, 16] obtêm o estado de 500 mil ficheiros pré-alocados de 4 KiB distribuídos por 500 diretorias.
macro		*file-server*	Carga de um servidor de ficheiros simulada através de 50 *threads*, que manipulam 75 mil ficheiros distribuídos por 150 diretorias.
		mail-server	Carga de um servidor de e-mails simulada através de 16 *threads*, que manipulam 100 mil ficheiros distribuídos por 100 mil diretorias.
		web-server	Carga de um servidor web simulada através de 100 *threads*, que manipulam 100 mil ficheiros distribuídos por 100 diretorias.

* Para as cargas de trabalho com operações orientadas a dados, foram também testados tamanhos de bloco de 4 e 1024 KiB.

Para cada configuração de teste foram realizadas três execuções, e calculada a média e o desvio padrão. Cada execução foi limitada a um período máximo de execução de 20 minutos, exceto na carga *read-Xth*, que na tentativa de estabilizar a média, o limite foi

aumentado para 30 minutos. As cargas foram geradas através do gerador de cargas de trabalho *filebench* (v1.5-alpha3) [70], e durante o seu período de execução foram recolhidos os recursos utilizados (p.e., percentagem de utilização de CPU, memória utilizada, e largura de banda ao nível do disco e rede), com recurso à ferramenta *dstat* (v0.7.3) [8].

5.2 Ambiente Experimental

Relativamente ao ambiente de testes utilizado, recorreu-se a duas máquinas, uma a operar como cliente RSafeFS e outra como servidor RSafeFS. Recorreu-se também a dois cenários de avaliação diferentes, um em que o servidor incluía um disco Solid State Drive (SSD), e um outro em que o servidor dispõe de um disco Non-Volatile Memory (NVMe).

Ao nível de *hardware*, as máquinas configuradas como cliente e como servidor com disco SSD tinham as seguintes especificações: um Intel Core i3-4170 CPU @ 3.70 GHz, com 2 cores físicos e 4 lógicos; 16 GiB RAM 2×(8 GiB DIMM DDR3 Synchronous 1600 MHz), 119 GiB SATA-III SAMSUNG MZ7LN128HCHP-000H1 SSD e um Intel 10G X550T NIC. A máquina com o disco NVMe tinha as seguintes especificações: um Intel Core i3-7100 CPU @ 3.90 GHz, 2 cores físicos e 4 lógicos; 16 GiB (8 GiB DIMM DDR4 Synchronous Unbuffered (Unregistered) 2400 MHz + 8 GiB DIMM DDR4 Synchronous Unbuffered (Unregistered) 2667 MHz), Samsung SSD 860 EVO 250 GB + Samsung SSD 970 EVO Plus 250 GB e um Intel 10G X550T NIC. Apesar das máquinas possuírem 16 GiB de RAM, a quantidade de RAM disponível para o sistema operativo foi limitada a 8 GiB para acelerar o aquecimento da *cache*. A ligação da rede partilhada pelas máquinas foi feita usando um *switch* 10GbE.

Ao nível de *software*, a máquina cliente e a máquina com o disco SSD utilizavam como sistema operativo o Ubuntu 18.04.3 LTS para x86-64, com a versão 4.15.0-70-generic do *kernel* Linux. A máquina equipada com o disco NVMe, utilizava como sistema operativo Ubuntu 18.04.4 LTS Linux para x86-64, com a versão 4.15.0-101-generic do *kernel* Linux. O sistema de ficheiros local utilizado pelos servidores para persistirem os dados foi o Ext4, configurado com a definições padrão.

Numa avaliação preliminar dos recursos disponíveis foi possível verificar que a configuração de armazenamento do servidor com disco SSD foi capaz de atingir débitos até 525 MiB/s em leitura sequencial e 145 MiB/s em escrita sequencial. Já o servidor com disco NVMe foi capaz de atingir taxas de transferência até 2.98 GiB/s em leitura sequencial e 1.39 GiB/s em escrita sequencial. Avaliando a rede, no ambiente com disco SSD foi medido um Round Trip Time (RTT) de 0.261 ms e um débito de 9.42 Gb/s entre máquinas; no ambiente com disco NVMe o RTT foi de 0.343 ms e o débito 9.41 Gb/s. O RTT foi medido através da ferramenta *ping* [21] e o débito através do *iperf3* (v3.1.3) [16].

Para as implementações do RSafeFS e das integrações de FUSE com as *frameworks* de RPC foram ainda utilizadas as seguintes bibliotecas instaladas com as seguintes versões: FUSE

(v2.9.7) [18], Cap'n Proto (v0.8.0) [3], gRPC (v1.27.2) [15], Protocol Buffers (v3.11.2) [22] e Boost (v1.72.0) [2]. Como compilador foi utilizado o gcc (v7.4.0) [12].

5.3 Grupos de Comparação e Configurações Experimentais

Para além da avaliação do desempenho do protótipo do sistema RSafeFS, considerou-se também o NFS. Uma vez que o RSafeFS permite construir sistemas de ficheiros remotos, a comparação com o NFS permite perceber quais as suas vantagens, desvantagens, e quais os avanços que este traz para o estado da arte atual.

Na Tabela 4 apresentamos os casos de estudo considerados nesta avaliação. Nestes casos de estudo estão também presentes as integrações de FUSE com as *frameworks* de RPC selecionadas (gRPC e Cap'n Proto), permitindo assim analisar o seu desempenho e selecionar qual a mais adequada a integrar na camada de comunicação do RSafeFS. Estes casos de estudo estão divididos em grupos, segundo o tipo teste e cliente do sistema (síncrono ou assíncrono). Assim, com esta divisão, é possível efetuar comparações justas entre cenários de teste pertencentes ao mesmo grupo.

Tabela 4: Grupos de comparação utilizados durante a avaliação experimental.

#	Cenários de teste	Grupos					
		Micro testes				Macro testes	
		Dados		Metadados			
		Síncrono	Assíncrono	Síncrono	Assíncrono	Síncrono	Assíncrono
1	NFS	✓	✓	✓	✓	✓	✓
2	FUSE-NFS	✗	✓	✗	✓	✗	✓
3	FUSE+Capnp	✓	✗	✓	✗	✓	✗
4	FUSE+gRPC	✓	✓	✓	✓	✓	✓
5	RSafeFS	✓	✓	✓	✓	✓	✓
6	RSafeFS-mc-lru-10s-100MiB	✗	✗	✓	✓	✗	✗
7	RSafeFS-mc-rnd-10s-100MiB	✗	✗	✓	✓	✗	✗
8	RSafeFS-mc-lru-60s-300MiB	✗	✗	✓	✓	✓	✓
9	RSafeFS-mc-rnd-60s-300MiB	✗	✗	✓	✓	✓	✓
10	RSafeFS-dc-lru-30s-1GiB-128KiB	✗	✗	✗	✗	✓	✓
11	RSafeFS-dc-rnd-30s-1GiB-128KiB	✗	✗	✗	✗	✓	✓

Ainda, com o objetivo de observar o impacto no desempenho introduzido pelo FUSE no protocolo NFS, foi utilizado um cenário cuja implementação do cliente NFS é feita através do FUSE (fuse-nfs [11]). Este sistema é de código aberto e apresenta-se como a solução padrão para sistemas de ficheiros baseados em FUSE usarem acesso e armazenamento remoto. Como este sistema apenas suporta o modelo assíncrono, este cenário apenas está presente nos grupos dos clientes assíncronos.

Na integração de FUSE com a *framework* Cap'n Proto RPC não foi implementado o cliente assíncrono. Isto porque, este tipo de cliente necessita que *threads* em segundo plano enviem pedidos para o servidor, contudo, o *stub* desta *framework* de RPC não permite ser partilhado por várias *threads*. Desta forma, este cenário de teste só está presente nos grupos de clientes síncronos.

De seguida, descrevem-se as características e as configurações mais relevantes de cada cenário de teste presente na Tabela 4:

Cenário 1. Cliente e servidor comunicam através da versão protocolo NFSv3. O cliente é configurado com as definições padrão, e o servidor é configurado com a opção *sync* que garante que este só responde ao cliente depois dos dados modificados serem persistidos. Apesar da opção *async* poder melhorar o desempenho, trata-se de uma violação do protocolo, pois permite o servidor responda antes dos dados estarem persistidos, que pode levar a perdas de dados ou à corrupção dos mesmos. Por isso, a configuração com o servidor assíncrono não será alvo de avaliação.

Cenário 2. Cliente NFS implementado através de FUSE. Cliente e servidor comunicam através da versão 3 do protocolo NFS, e o servidor NFS é configurado com as mesmas propriedades que o cenário anterior.

Cenário 3. Integração de FUSE com a *framework* de RPC Cap'n Proto, com implementação *single-threaded* do cliente.

Cenário 4. Integração de FUSE com a *framework* gRPC, com implementação *multi-threaded* do cliente e do servidor.

Cenário 5. Instâncias cliente e servidor RSafeFS sem camadas de processamento adicionais. A camada de comunicação é implementada através da *framework* gRPC, que permite implementação *multi-threaded* do cliente e servidor.

Cenário 6. Cliente do RSafeFS com uma camada de *caching* de metadados, com 100 MiB de armazenamento máximo, atributos válidos durante 10 segundos, e com uma política que substitui os atributos menos utilizados recentemente.

Cenário 7. A mesma configuração que o cenário anterior mas com a camada de *caching* com uma política de substituição aleatória.

Cenário 8. Cliente do RSafeFS com uma camada de *caching* de metadados, com 300 MiB de armazenamento máximo, atributos válidos durante 60 segundos, e com uma política que substitui os atributos menos utilizados recentemente.

Cenário 9. A mesma configuração que o cenário anterior mas com a camada de *caching* com uma política de substituição aleatória.

Cenário 10. Cliente do RSafeFS com um camada de *caching* de dados, com 1 GiB de armazenamento máximo, com os blocos de dados de 128 KiB válidos durante

30 segundos, e com uma política que substitui os blocos de dados menos utilizados recentemente.

Cenário 11. A mesma configuração que o cenário anterior mas com a camada de *caching* com uma política de substituição aleatória.

5.4 RESULTADOS E OBSERVAÇÕES GLOBAIS

Nesta avaliação experimental foi medido o débito de cada sistema apresentado anteriormente, para todas as cargas de trabalho. O débito é apresentado em MiB/s para os micro testes orientados a dados, e operações por segundo (ops/s) para os restantes cenários. As tabelas apresentadas nesta secção apresentam o desempenho absoluto para o NFS e o desempenho relativo para os restantes cenários de teste. Ainda, para cada tabela, as células seguem um esquema de cores, dividido em cinco categorias, que visa reportar o desempenho relativo:

(1) Azul: a diferença relativa de desempenho é superior a 5%.
(2) Verde: a diferença relativa de desempenho está compreendida entre]-5%, 5%].
(3) Amarelo: a diferença relativa do desempenho está compreendia entre]-5%, -25%].
(4) Laranja: a diferença relativa do desempenho está compreendida entre]-25%, -50%].
(5) Vermelho: a diferença relativa do desempenho é inferior a -50%.

Para as tabelas que apresentam os resultados de débito, uma percentagem negativa indica que o desempenho do sistema a comparar é pior que o NFS. Para as tabela que apresentam os resultados dos recursos computacionais utilizados, nomeadamente CPU, memória, e rede, valores negativos indicam que o sistema reportou um menor consumo destes recursos. Estes últimos resultados são apresentados no Anexo A.

De forma a simplificar a leitura das observações presentes nas seguintes secções, sempre que estas mencionam desempenho, referem-se ao débito, a não ser que explicitamente mencionada a métrica referente.

5.4.1 *Micro testes*

Dados

As Tabelas 5 e 6 apresentam os débitos de cada cenário de teste com clientes síncrono e assíncrono, respetivamente, sobre os micro testes orientados aos dados, nos ambientes com as máquinas servidores equipadas com discos SSD e NVMe.

Tabela 5: Resultados do débito dos clientes síncronos para micro testes orientados a dados.

		SSD				NVMe			
		NFS MiB/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**NFS** MiB/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %
seq-read	1th-1f-4k	520.1	−28.9	−23.2	−25.0	1092.2	−72.3	−56.3	−56.1
	1th-1f-1024k	520.1	−28.9	−23.2	−25.0	1092.2	−75.3	−53.1	−53.1
	16th-16f-4k	526.1	−36.5	−2.7	−2.2	1089.6	−73.1	−6.0	−9.8
	16th-16f-1024k	524.1	−34.9	−0.8	−0.8	1121.2	−73.7	−11.4	−10.5
rand-read	1th-1f-4k	18.7	−6.8	−30.6	−31.0	19.2	−21.8	−7.1	−10.9
	1th-1f-1024k	254.4	−20.5	−15.0	−16.9	414.7	−30.5	+6.3	+9.7
	16th-16f-4k	193.8	−90.4	−19.0	−19.2	214.3	−93.1	−19.0	−20.0
	16th-16f-1024k	544.9	−52.2	+10.5	+12.2	1236.3	−74.7	−6.0	−6.0
seq-write	1th-1f-4k	0.5	+0.0	+0.0	+0.0	0.6	−16.7	−16.7	−11.7
	1th-1f-1024k	59.6	−81.2	−81.9	−80.8	64.4	−78.5	−79.3	−79.3
	16th-16f-4k	1.8	−72.2	+76.1	+59.4	2.4	−79.2	+72.1	+73.8
	16th-16f-1024k	97.4	−88.9	−63.3	−65.4	239.1	−94.0	−65.1	−70.1
rand-write	1th-1f-4k	0.6	−16.7	+0.0	+0.0	0.6	−9.5	−9.5	−4.8
	1th-1f-1024k	43.9	−75.2	−68.1	−73.3	45.5	−73.8	−72.7	−73.2
	16th-16f-4k	2.3	−78.5	+67.4	+63.1	2.6	−77.8	+84.0	+81.7
	16th-16f-1024k	73.6	−85.4	−45.6	−45.8	99.0	−88.7	−42.1	−44.3

A seguir, são enumeradas as principais observações dos resultados da Tabela 5, bem como das Tabelas 13, 14, 15, que demonstram o consumo de recursos computacionais nas máquinas durante a execução das cargas de trabalho nos cenários de teste com cliente síncrono.

■ Observação 1. O desempenho varia entre cenários de teste, cargas de trabalho e ambientes experimentais. Globalmente, o NFS obtém melhor desempenho que os restantes sistemas na maioria das cargas de trabalho. Relativamente ao NFS, a degradação máxima de desempenho é de 94.0% na carga *seq-write-16th-16f-1024k* com sistema FUSE+Capnp, no ambiente com disco NVMe, e o ganho máximo de desempenho é de 84.0% na carga *rand-write-16th-16f-4k* no sistema FUSE+gRPC, também no ambiente com disco NVMe.

■ Observação 2. A integração de FUSE com o Cap'n Proto RPC tem pior desempenho que a integração com gRPC na maioria das cargas. A diferença de desempenho média entre entre estes sistemas, considerando todas as cargas de trabalho em ambos os ambientes (SSD e NVMe), é de -41.6%.

■ Observação 3. A diferença de desempenho entre FUSE+gRPC e RSafeFS não é significativa. O RSafeFS apresenta uma diferença média de desempenho, em relação ao sistema FUSE+gRPC, de -1.5%. O ganho e degradação máxima de desempenho do RSafeFS, relativamente ao sistema FUSE+gRPC, ocorrem no ambiente com disco SSD e atingem os valores +6.5% e -16.4%, nas cargas *seq-write-1th-1f-1024k* e *rand-write-1th-1f-1024k*, respetivamente.

■ Observação 4. O NFS tem sempre melhor desempenho que os restantes sistemas nas cargas que simulam leituras sequenciais, e esta diferença de desempenho é superior no ambiente com disco NVMe. Por exemplo, na carga *seq-read-1th-1f-4k*, o RSafeFS apresenta uma perda de desempenho de 25.0% no ambiente com disco SSD, enquanto que com o disco NVMe a degradação é de 56.1%. Verifica-se ainda que, para este tipo de cargas, não existem diferenças significativas no desempenho dos sistemas quando as aplicações utilizam diferentes tamanhos de bloco.

■ Observação 5. A diferença de desempenho entre NFS e RSafeFS nas leituras sequenciais em *16th-16f* é menor do que em *1th-1f*. Por exemplo, o RSafeFS no ambiente com disco SSD tem perdas desempenho de 25.0% em ambas as cargas com 1 *thread*, enquanto que com 16 *threads*, com os mesmos tamanhos de bloco, as perdas são de 2.2% e 0.8%.

■ Observação 6. A diferença de desempenho entre os sistemas NFS e RSafeFS nas escritas que utilizam 1 *thread* para realizarem operações com tamanho de bloco de 4 KiB (p.e., *seq-write-1th-1f-4k*, *rand-write-1th-1f-4k*) é mínima. A diferença de desempenho máxima observada neste tipo de cargas foi de -11.7%. Contudo, uma vez que o valor absoluto do NFS corresponde a 0.6 MiB/s, e o *filebench* apenas reporta resultados com uma casa decimal, esta diferença de desempenho é quase negligenciável.

■ Observação 7. Para escritas com tamanhos de bloco de 1024 KiB, o RSafeFS e os restantes sistemas baseados em FUSE apresentam uma degradação de desempenho, relativamente ao NFS, sempre superior a 42.1%.

■ Observação 8. O RSafeFS tem melhor desempenho que o NFS para as escritas sequenciais e aleatórias quando executadas com 16 *threads* e com um tamanho de bloco de 4 KiB. Por exemplo, na carga *seq-write-16th-16f-4k*, o RSafeFS tem ganhos de desempenho de 59.4% no ambiente com disco SSD, e de 73.8% com o disco NVMe.

■ Observação 9. O tamanho de bloco utilizado pelas aplicações nas operações de escrita influencia significativamente o desempenho dos sistemas. Os resultados revelam que os sistemas têm pior desempenho quando são utilizados tamanhos de bloco mais pequenos. Por exemplo, o NFS atinge um débito de 59.6 MiB/s na carga *seq-write-1th-1f-1024k* no ambiente com disco SSD, já quando são feitas operações com tamanho de bloco de 4 KiB o débito é de 0.5 MiB/s.

■ Observação 10. Em relação à utilização de CPU, para a maioria das cargas verifica-se um consumo mais elevado, tanto no cliente como no servidor, pelos sistemas baseados em FUSE com o cliente e servidor *multi-threaded*. Por exemplo, em média, o cliente e o servidor RSafeFS consomem, respetivamente, 2.3× e 0.2× mais CPU que o cliente e servidor NFS. Contudo existem algumas exceções onde o NFS consome mais CPU, por exemplo, nas escritas sequenciais e aleatórias *1th-1f-1024k* tanto o cliente como o servidor de todos os sistemas baseados em FUSE utilizam menos CPU que o NFS.

■ Observação 11. No que diz respeito à utilização de memória, o consumo de RSafeFS é superior ao NFS, tanto no servidor como o cliente. O RSafeFS consome em média mais 0.1× de memória no cliente e 0.2× no servidor.

Tabela 6: Resultados do débito dos clientes assíncronos para micro testes orientados a dados.

		SSD				NVMe			
		NFS MiB/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**NFS** MiB/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %
seq-read	1th-1f-4k	520.1	−12.9	−7.3	−7.3	1069.4	−58.4	−58.6	−59.0
	1th-1f-1024k	520.1	−7.4	−4.5	−4.5	1092.2	−57.1	−57.1	−57.1
	16th-16f-4k	500.1	−0.0	−0.7	+0.3	1069.4	−17.4	−7.2	−8.9
	16th-16f-1024k	501.7	−0.6	−1.0	+0.5	1092.2	−22.4	−7.2	−8.3
rand-read	1th-1f-4k	20.0	−18.1	−34.0	−34.8	15.1	+8.7	+16.7	+14.3
	1th-1f-1024k	254.4	+16.0	+30.1	+28.6	427.7	−8.8	+0.3	−1.5
	16th-16f-4k	192.5	−72.0	−22.0	−22.6	217.3	−75.7	−22.9	−25.0
	16th-16f-1024k	543.2	−2.6	+10.5	+10.3	1192.3	−19.2	+0.1	−2.7
seq-write	1th-1f-4k	116.3	−77.2	+13.7	+16.2	189.8	−84.7	+34.5	+32.8
	1th-1f-1024k	142.9	−80.8	+3.9	+3.5	179.9	−84.8	+148.4	+154.2
	16th-16f-4k	121.2	−37.7	+17.2	+16.4	415.9	−80.0	−19.9	−21.2
	16th-16f-1024k	143.9	−48.6	+7.5	+9.8	414.5	−80.5	+18.6	+18.8
rand-write	1th-1f-4k	64.3	−52.3	+85.3	+96.5	89.0	−74.1	+154.9	+153.0
	1th-1f-1024k	99.3	−71.0	+21.6	+27.1	165.0	−85.7	+104.7	+112.4
	16th-16f-4k	89.4	−3.2	+20.8	+18.5	134.0	−37.2	+109.6	+107.9
	16th-16f-1024k	92.0	−14.9	+11.7	+9.5	212.0	−61.6	+46.1	+47.6

A seguir, são enumeradas as principais observações dos resultados da Tabela 6, bem como das Tabelas 16, 17, 18, que demonstram o consumo de recursos computacionais nas máquinas durante a execução das cargas de trabalho nos cenários de teste com cliente assíncrono.

■ Observação 1. Analogamente aos resultados obtidos com os clientes síncronos, o desempenho varia entre cenários de teste, cargas de trabalho e ambientais experimentais. Globalmente, o NFS obtém melhor desempenho que os restantes sistemas nas cargas de leituras. Neste tipo de cargas, observa-se uma diferença de desempenho média entre o NFS e o RSafeFS de -11.1%. Nas cargas de escrita, o RSafeFS obtém, na maioria dos cenários, melhor desempenho que os sistemas que utilizam o protocolo NFS. Apresentando um ganho médio de desempenho, relativamente ao NFS, de 50.2%.

■ Observação 2. O RSafeFS e a integração de FUSE com gRPC têm desempenhos equiparáveis para todos os cenários de teste. A diferença de desempenho média do RSafeFS relativamente à integração de FUSE com gRPC é de -0.04%. Ainda, o ganho máximo de desempenho observado é de 6.0% na carga *rand-write-1th-1f-4k*, e a degradação máxima de 2.8% na carga *read-16th-16f-1024k*.

■ Observação 3. O FUSE-NFS apresenta um desempenho similar aos restantes sistemas baseados em FUSE no que diz respeito a leituras sequenciais, atingindo, contudo, pior desempenho nas restantes cargas de trabalho, em ambos os discos.

■ Observação 4. Os sistemas baseados em FUSE, nos quais se incluem o FUSE-NFS e o RSafeFS, têm pior desempenho que o NFS nas leitura sequenciais, sendo esta degradação particularmente acentuada no ambiente com disco NVMe, para cargas do tipo *1th-1f*.

■ Observação 5. Globalmente, os sistemas baseados em FUSE, onde se incluem o RSafeFS e o FUSE-NFS, têm um consumo superior CPU relativamente ao NFS. Em média, o RSafeFS consome 4.4× mais CPU no cliente e 0.4× mais no servidor, o FUSE-NFS consome 4.6× mais CPU no cliente que faz com que o consumo de CPU, no servidor NFS, aumente 0.5×.

■ Observação 6. Em relação ao consumo de memória, em todas as cargas de trabalho, os sistemas RSafeFS e a integração de FUSE com gRPC apresentam um maior consumo de memória que o NFS, sobretudo na máquina cliente. Em relação ao NFS, no RSafeFS observa-se um consumo médio de memória 13× superior na máquina cliente, e 0.7× na máquina servidor. No entanto, importa referir que o consumo de memória deste cliente pode ser ajustado aos recursos da máquina. Neste cenário de teste, o cliente RSafeFS foi configurado para manter em memória dados até um máximo de 1 GiB, antes de serem agrupados dados contíguos de pedidos consecutivos, e estes começarem a serem enviados para o servidor. Caso seja necessário, este cliente pode ser configurado com um valor inferior, permitindo assim um menor consumo de memória.

Metadados

As Tabelas 7 e 8 apresentam os débitos de cada cenário de teste com clientes síncrono e assíncrono, respetivamente, sobre os micro testes orientados as metadados, nos ambientes com as máquinas servidores equipadas com discos SSD e NVMe.

Como referido anteriormente as cargas de trabalho *read-Xth* foram executadas durante um período máximo de 30 minutos, de forma a atingir estabilidade entre diferentes execuções. Contudo, mesmo assim, durante todos os cenários, obtivemos um desvio padrão elevado em relação à média, impossibilitando assim retirar observações válidas para este tipo de carga de trabalho.

Tabela 7: Resultados do débito dos clientes síncronos para micro testes orientados a metadados.

	SSD								NVMe							
	NFS ops/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** ops/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
create-1th	211.9	−7.5	−11.0	−23.9	−17.8	−18.4	−12.0	−11.3	218.4	−15.6	−16.1	−16.8	−16.5	−16.4	−16.3	−16.4
create-16th	233.6	−11.5	−8.7	−22.0	−16.8	−16.1	−9.4	−9.1	243.4	−22.3	−10.7	−11.3	−11.7	−11.9	−11.7	−11.8
delete-1th	135.8	−13.3	−6.7	−6.9	−6.8	−7.0	−7.2	−7.9	143.6	−6.5	−5.6	−7.2	−5.1	−6.1	−8.6	−7.3
delete-16th	139.4	−18.1	−7.8	−7.8	−7.8	−7.8	−8.0	−8.2	156.0	−12.9	−11.3	−10.7	−11.2	−11.3	−14.3	−13.9
read-1th	6183.9	−89.2	−90.3	−90.7	−90.1	−90.1	−90.4	−90.4	6062.4	−88.3	−87.5	−88.0	−88.5	−88.5	−88.5	−88.6
read-16th	57050.2	−96.2	−91.2	−92.6	−90.2	−90.3	−91.4	−91.4	59644.0	−97.3	−89.1	−89.3	−89.7	−89.6	−89.5	−89.5
stat-1th	52203.5	−87.6	−89.0	−89.0	−89.1	−89.6	−89.4	−89.8	31932.1	−83.8	−84.2	−86.4	−87.0	−84.5	−85.4	−86.1
stat-16th	430451.1	−98.4	−89.9	−89.9	−91.1	−91.1	−71.6	−70.9	421026.8	−98.6	−89.7	−89.8	−91.6	−90.9	−79.4	−73.8

A seguir, são enumeradas as principais observações dos resultados da Tabela 7, bem como das Tabelas 19, 20, 21, que demonstram o consumo de recursos computacionais, nomeadamente CPU, memória utilizada, e largura de banda de rede e disco, pelas máquinas que continham o cliente e servidor, durante a execução das diferentes cargas de trabalho nos cenários de teste com cliente síncrono.

■ Observação 1. Para todas as cargas de trabalho orientadas a metadados, o NFS apresenta o melhor desempenho. Esta diferença de desempenho é mais significativa nas cargas do tipo *read* e *stat*, observando-se perdas de desempenho superiores a 87.5% na carga *read*, e superiores a 70.9% na carga *stat*. Já nas cargas do tipo *create* ou *delete*, embora continue a haver perdas de desempenho, a diferença já não é tão acentuada, apresentando valores compreendidos entre -7.5% e -23.9% no *create* (em SSD e NVMe) e -5.1% e -18.1% no *delete* em SSD e NVMe.

■ Observação 2. A camada de *caching* de metadados, utilizada no RSafeFS, não introduz melhorias no desempenho nas cargas *create* e *delete*. Contudo, para a carga *stat-16th*, quando

a camada de *caching* é configurada com 300 MiB de capacidade máxima e 60 segundos como período máximo de validade dos metadados, o RSafeFS reporta ganhos médios de 1.8× relativamente ao RSafeFS sem camadas de processamento adicionais.

■ Observação 3. No que diz respeito à utilização de CPU, com exceção das cargas de trabalho *read-16th* e *stat-16th*, todos os sistemas apresentam maior consumo comparativamente ao NFS. Em concreto, considerando os resultados obtidos nos ambientes com disco SSD e NVMe, o RSafeFS sem camadas de processamento adicionais, consome em média mais 1.0× CPU que o NFS no cliente, e 0.4× no servidor.

■ Observação 4. No que diz respeito à utilização de memória, o sistema NFS consume menos memória para todas as cargas de trabalho. Sendo esta diferença particularmente acentuada nas cargas *read* e *stat*, relativamente aos sistemas que utilizam a *framework* gRPC para garantir a comunicação entre cliente e servidor.

■ Observação 5. No que diz respeito à utilização de rede, à excepção da carga de trabalho *create*, o sistema NFS utiliza mais rede que os restantes cenários.

Tabela 8: Resultados do débito dos clientes assíncronos para micro testes orientados a metadados.

	SSD								NVMe							
	NFS ops/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** ops/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
create-1th	208.3	−17.3	−23.2	−23.0	−19.1	−19.0	−13.2	−13.2	220.2	−18.5	−18.0	−18.9	−17.8	−18.1	−18.0	−18.1
create-16th	227.2	−22.1	−19.6	−19.0	−15.0	−15.1	−6.9	−7.7	245.5	−24.8	−12.1	−12.9	−11.8	−11.8	−11.6	−11.8
delete-1th	129.1	−14.2	−18.6	−18.1	−13.5	−13.7	−15.1	−15.0	145.8	−15.5	−8.1	−5.9	−12.2	−8.1	−9.4	−9.5
delete-16th	124.0	−9.9	−13.8	−13.4	−8.9	−8.9	−9.6	−9.9	151.0	−15.3	−9.3	−8.2	−11.0	−10.5	−11.3	−9.7
read-1th	6020.2	−90.9	−89.9	−89.8	−89.9	−89.9	−89.9	−89.9	6037.0	−89.7	−88.8	−88.4	−88.7	−88.6	−88.8	−88.4
read-16th	57075.8	−93.7	−90.1	−90.4	−90.6	−90.5	−90.5	−90.5	60188.8	−94.6	−89.6	−89.6	−90.0	−90.2	−89.9	−89.8
stat-1th	49740.8	−94.8	−88.3	−88.4	−88.7	−89.6	−88.8	−88.7	34308.6	−93.3	−84.7	−84.8	−85.7	−85.8	−85.9	−85.9
stat-16th	426640.9	−98.6	−90.6	−90.4	−91.7	−91.7	−71.3	−69.9	415545.3	−98.6	−90.0	−90.1	−92.8	−92.5	−79.7	−81.1

A seguir, são enumeradas as principais observações dos resultados da Tabela 8, bem como das Tabelas 22, 23, 24, que demonstram o consumo de recursos computacionais, nomeadamente CPU, memória utilizada, e largura de banda de rede e disco, pelas máquinas que continham o cliente e servidor, durante a execução das diferentes cargas de trabalho nos cenários de teste com cliente assíncrono.

■ Observação 1. Para todos os cenários avaliados, o sistema NFS apresenta o melhor desempenho. Em particular, as cargas do tipo *read* e *stat* apresentam a maior diferença de

desempenho, sendo esta até 94.6% na carga *read* e 98.6% na carga *stat*. Relativamente às cargas do tipo *create* e *delete*, esta diferença de desempenho máxima é de -24.8% e -18.6%, respetivamente.

■ Observação 2. O desempenho nos sistemas baseados em FUSE, nos quais se incluem o RSafeFS e o FUSE-NFS, é similar entre as diferentes configurações. Para as cargas *create* e *delete*, todos estes sistemas apresentam uma degradação de desempenho, relativamente ao NFS, compreendida entre 5% e 25%. Nas cargas *read* e *stat*, observa-se a uma degradação superior a 50%.

■ Observação 3. A camada de *caching* utilizada no RSafeFS apenas melhora o desempenho, relativamente aos sistemas baseados em FUSE, em *stat-16th* quando o período de validade dos metadados é de 60 segundos, e a capacidade máxima de armazenamento é 300 MiB. Ainda, as diferentes configurações da camada de *caching* ao nível das políticas de substituição não apresentam diferenças de desempenho significativas.

■ Observação 4. Para estes micro testes orientados a metadados, os desempenhos obtidos, e os recursos computacionais utilizados por cada um dos sistemas, é similar com clientes síncronos e assíncronos.

5.4.2 *Macro testes*

As Tabelas 9 e 10 apresentam o desempenho (débito) atingido pelos sistemas NFS, baseados em FUSE, e baseados RSafeFS, quando submetidos a macro testes com clientes síncrono e assíncrono.

Tabela 9: Resultados do débito dos clientes síncronos para macro testes.

	SSD								NVMe							
	NFS ops/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-md-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** ops/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-md-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
file-server	826.5	−62.7	−63.8	−65.2	−58.4	−57.2	−57.6	−55.4	1088.4	−73.6	−51.9	−66.0	−60.4	−51.5	−69.2	−68.5
mail-server	668.2	−47.2	−43.3	−44.7	−38.4	−38.9	−38.0	−38.2	827.6	−57.1	−39.6	−46.4	−44.7	−38.6	−48.8	−48.1
web-server	4370.9	−37.2	−37.7	−39.2	−38.1	−38.2	−29.9	−29.7	5034.6	−60.5	−43.9	−45.0	−46.8	−46.8	−30.2	−31.0

A seguir, são enumeradas as principais observações dos resultados da Tabela 9, bem como das Tabelas 25, 26, 27, que demonstram o consumo de recursos computacionais,

nomeadamente CPU, memória utilizada, e largura de banda de rede e disco, pelas máquinas que continham o cliente e servidor, durante a execução das diferentes cargas de trabalho nos cenários com cliente síncrono.

■ Observação 1. Para todos os cenários de teste, o sistema NFS apresenta o melhor desempenho. Sobre a carga de trabalho *file-server*, todos os sistemas compreendem uma degradação no desempenho superior a 50%, quer na configuração com disco SSD como com disco NVMe. Relativamente às cargas *mail-server* e *web-server*, à exceção do sistema FUSE+Capnp sobre a configuração com disco NVMe, cujo desempenho para todas as cargas de trabalho sofre uma degradação superior a 50%, todos os restantes sistemas compreendem uma degradação no desempenho entre 25% e 50%, tanto para SSD como para NVMe.

■ Observação 2. A camada de *caching* de dados utilizada no RSafeFS não melhora significativamente o desempenho deste sistema. Já a utilização da camada de *caching* de metadados melhora o desempenho do RSafeFS, na carga *web-server*, relativamente ao RSafeFS sem camadas de processamento adicionais, entre 15.3% e 15.6% no ambiente com disco SSD, e 25.4% e 27.0% sobre a configuração com disco NVMe. Ainda, as diferentes políticas de substituição utilizadas pelas camadas de *caching* no sistema RSafeFS, nomeadamente de substituição aleatória e de menos utilizado recentemente, não acarretam nenhuma variação significativa no desempenho.

■ Observação 3. No que diz respeito à utilização de CPU, os sistemas baseados em FUSE têm maior consumo que o NFS em ambas as máquinas. Considerando todas as cargas de trabalho do tipo macro, a diferença média na utilização CPU entre os sistemas RSafeFS e NFS, nas máquinas cliente e servidor é de 2.5× e 1.4×, respetivamente.

■ Observação 4. À excepção do servidor FUSE+Capnp, todos os outros sistemas têm maior utilização de memória que o NFS.

■ Observação 5. Os sistemas RSafeFS que incluem camadas de *caching* de dados (nomeadamente RSafeFS-dc-lru-30s-1GiB e RSafeFS-dc-rnd-30s-1GiB), têm uma maior utilização de memória que o NFS para todas as cargas de trabalho. Em média, relativamente ao NFS, o RSafeFS com estas configurações da camada de *caching* de dados, consome no cliente, mais 2.1× de memória na carga *file-server*, 8.3× na carga *mail-server*, e 1.6× na carga *web-server*.

■ Observação 6. Todos os sistemas avaliados têm uma maior utilização de rede que o sistema NFS, embora o seu desempenho seja inferior. Isto deve-se a *caching* eficiente de dados e metadados realizado pelo cliente NFS, que permite responder localmente aos pedidos das aplicações e comunicar menos vezes com o servidor, traduzindo-se assim numa menor utilização da rede.

Tabela 10: Resultados do débito dos clientes assíncronos para macro testes.

	SSD								NVMe							
	NFS ops/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** ops/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
file-server	803.6	−56.3	−64.4	−64.6	−56.3	−56.8	−62.1	−58.7	922.4	−61.7	−59.9	−60.4	−60.7	−60.2	−60.3	−60.5
mail-server	663.3	−39.3	−44.5	−43.4	−38.4	−31.9	−42.0	−39.4	795.1	−44.5	−44.0	−43.6	−44.4	−44.4	−44.2	−37.5
web-server	148956.7	−98.0	−94.0	−94.0	−95.3	−95.2	−94.0	−93.9	160927.8	−98.6	−93.4	−93.5	−94.9	−94.9	−93.7	−93.7

A seguir, são enumeradas as principais observações dos resultados da Tabela 10, bem como das Tabelas 28, 29, 30, que demonstram o consumo de recursos computacionais, nomeadamente CPU, memória utilizada, e largura de banda de rede e disco, pelas máquinas que continham o cliente e servidor, durante a execução das diferentes cargas de trabalho nos cenários com cliente assíncrono.

■ Observação 1. Comparativamente ao NFS, a degradação de desempenho de todos os sistemas avaliados encontra-se na mesma ordem de grandeza/magnitude. Adicionalmente, o sistema FUSE-NFS apresenta um desempenho similar aos restantes sistemas baseados em FUSE. Isto é, quer para o ambiente com SSD como com NVMe, nas cargas *file-server* e *web-server* a diferença de desempenho é superior a -50%, e na carga *mail-server* a degradação do desempenho está compreendida entre 25% e 50%.

■ Observação 2. Na carga de trabalho *web-server*, comparativamente ao cliente síncrono, o cliente assíncrono do NFS apresenta uma melhoria no desempenho de 34×. Contudo, este aumento significativo no desempenho não é acompanhado pelos restantes sistemas, atingindo por isso uma diferença relativa nunca inferior a 90%.

■ Observação 3. Em relação à utilização de CPU, exceptuando na carga *web-server*, os restantes sistemas baseados em FUSE têm um maior consumo que o NFS, tanto no cliente como no servidor.

■ Observação 4. Nos sistemas RSafeFS que incluem camadas de caching de dados (nomeadamente RSafeFS-dc-lru-30s-1GiB e RSafeFS-dc-rnd-30s-1GiB), têm uma maior utilização de memória que o NFS para todas as cargas de trabalho. Relativamente ao NFS, estas configurações do RafeFS, com uma camada de *caching* de dados no cliente, consomem em média, no cliente, 2.3× mais memória na carga *file-server*, 8.0× na carga *mail-server*, e 1.7× na carga *mail-server*.

■ Observação 5. Tal como observado com os clientes síncronos, os sistemas baseados em FUSE, onde se incluem o RSafeFS e o FUSE-NFS, embora tenham o desempenho inferior ao NFS, têm uma maior utilização da rede.

5.5 ANÁLISE

Nesta secção é feita uma análise detalhada sobre os resultados apresentados na secção anterior, sendo identificadas as razões das diferenças de desempenho entre os sistemas NFS, baseados em FUSE, e baseados em RSafeFS, para as diferentes cargas de trabalho. Esta análise segue a mesma ordem que os resultados apresentados anteriormente. Inicialmente é feita a análise para os micro testes, primeiramente para as cargas orientadas a dados e depois para as cargas orientadas a metadados, e finalmente, para os macro testes.

5.5.1 *Micro testes: dados*

Nesta subsecção discutimos os resultados apresentados nas Tabelas 5 e 6, sendo feita primeiramente uma análise sobre as operações de leitura, e depois sobre as operações de escrita.

Leitura

No que diz respeito aos cenários de teste orientados às leituras, o comportamento dos sistemas com clientes síncronos e assíncronos é idêntico, pois estes clientes apenas diferem na implementação das operações de escrita. Desta forma, a análise para este tipo de cargas não distingue os dois tipos de clientes.

Leitura sequencial. Neste tipo de teste, *N threads* lêem sequencialmente *N* ficheiros. Quando são utilizadas várias *threads*, para se garantir que não são realizadas leituras concorrentes sobre o mesmo ficheiro, cada uma das *threads* lê um ficheiro distinto.

Os resultados anteriormente apresentados demonstram que o NFS obtém sempre melhor desempenho que os restantes sistemas baseados em FUSE, e que, em todos os sistemas, a diferença de desempenho entre cargas que apenas diferem no tamanho de bloco não é significativa. A análise sobre o comportamento dos vários sistemas revelou que isto acontece devido ao mecanismo de *read-ahead*, que através de *prefetching* de dados normaliza os tamanhos dos vários pedidos, independentemente do tamanho de bloco utilizado pelas aplicações. Relativamente à diferença de desempenho entre o NFS e os restantes sistemas baseados em FUSE, esta deve-se ao tamanho de *read-ahead* utilizado por cada um dos sistemas.

O NFS utiliza um *read-ahead* máximo de 1024 KiB, assim, leituras sequenciais geradas pelas aplicações com tamanho de bloco inferior a este são normalizadas para este valor. Permitindo, por exemplo, que cargas do tipo *seq-read-Xth-Xf-4k* tenham desempenhos semelhantes às do tipo *seq-read-Xth-Xf-1024k*.

Os sistemas baseados em FUSE, nos quais se incluem os baseados em RSafeFS, têm o tamanho de *read-ahead* máximo limitado a 128 KiB pelo FUSE [74]. Ainda, esta limitação no tamanho de bloco dos pedidos do FUSE, faz com que pedidos gerados com tamanhos de bloco superiores a 128 KiB (p.e., *seq-read-Xth-Xf-1024k*), sejam também limitados pelo tamanho de bloco máximo utilizado pelo FUSE. Desta forma, nos sistemas baseados em FUSE, qualquer pedido de leitura sequencial é normalizado para 128 KiB, independentemente do tamanho pedido pelas aplicações.

Esta diferença de tamanho em que os pedidos de leitura são normalizados, traduz-se, na prática, num maior número de operações efetuadas pelos sistemas baseados em FUSE. Tomando como exemplo o tamanho do ficheiro lido no teste com 1 *thread*, enquanto o NFS realiza um total de 32,768 operações de 1024 KiB para ler os 32 GiB de dados, os sistemas baseados em FUSE utilizam 262,144 operações de 128 KiB, transcrevendo-se em 8× mais operações.

Quando as cargas utilizam 16 *threads* para serem lidos 16 ficheiros, a diferença de desempenho entre o NFS e os sistemas baseados em FUSE, que utilizam a *framework* gRPC, diminui. Por exemplo, no ambiente com disco SSD, enquanto que a diferença de desempenho entre clientes síncronos RSafeFS e NFS em *seq-read-1th-1f-4k* é de -25.0%, em *seq-read-16th-16f-4k* é de -2.2%. Isto acontece porque, no NFS, o volume de dados a transferir entre o cliente e servidor é superior ao limite máximo da rede, limitando assim o desempenho do sistema. Já nos restante sistemas, o uso de mais *threads* vai permitir utilizar mais recursos, aproximando-se assim do desempenho do NFS. Na integração de FUSE com o Cap'n Proto RPC não se verificam estes ganhos de desempenho entre cargas porque a implementação do cliente é *single-thread*, só conseguindo assim realizar sequencialmente as operações geradas pelas 16 *threads*.

Leitura aleatória. Neste tipo de teste, *N threads* lêem aleatoriamente dados de *N* ficheiros. Como nas leituras sequenciais, para se garantir que não são realizadas leituras concorrentes sobre o mesmo ficheiro, cada *thread* efetua leituras num ficheiro distinto. Como neste tipo de teste os padrões de acesso a ficheiros são aleatórios, os sistemas não realizam *read-ahead*. Desta forma, os sistemas apenas lêem o volume de dados pedido pelas aplicações. Assim, como se observa nos resultados apresentados anteriormente, a diferença de desempenho entre o NFS e os restantes sistemas baseados em FUSE diminui, e em algumas cargas foi até possível observar uma melhoria no desempenho relativamente ao NFS.

Escrita

No que diz respeito às cargas de trabalho orientadas às escritas de dados, como se pode verificar pelos resultados apresentados, há diferenças bastante significativas no desempenho dos sistemas quando estes são utilizados com clientes síncronos ou assíncronos. Como exemplo desta diferença de desempenho, no cenário de testes com o RSafeFS, é possível verificar ganhos de desempenho de 5× entre clientes assíncronos e síncronos para a carga *seq-write-1th-1f-4k*.

Esta radical diferença de desempenho é justificada pelas garantias de coerência de dados dos sistemas de ficheiros remotos. Em detalhe, nos sistemas com o cliente síncrono, o servidor garante que os dados modificados pelas operações de escrita são persistidos antes de responder ao cliente, para garantir que não há perda de dados (no caso de falha temporária do servidor). Com o cliente assíncrono, o servidor só garante que os dados são persistidos quando explicitamente clientes pedem, por exemplo, através da operação *commit* no NFS, ou das operações *fsync* ou *flush* no caso RSafeFS.

As garantias oferecias pelo clientes síncronos permitem maior coerência entre múltiplos clientes, comparativamente aos cliente assíncronos, mas em contra partida acrescentam um custo significativo no desempenho do sistema.

Escrita sequencial. Neste tipo de teste, *N threads* escrevem sequencialmente em *N* ficheiros. Analogamente às operações de leitura, para se garantir que não são realizadas escritas concorrentes sobre o mesmo ficheiro, cada uma das *threads* escreve num ficheiro distinto.

Como referido anteriormente, nos cenários de teste com cliente síncrono qualquer escrita feita pelas aplicações gera uma operação de escrita a ser enviada para o servidor, e este só responde depois de garantir que os dados envolvidos na operação foram persistidos. Para garantir que os dados são persistidos, no servidor, cada operação de escrita é seguida de uma operação que persiste os dados, por exemplo, através da chamada de sistema *fsync*.

Os resultados obtidos com os clientes síncronos revelam que o NFS e os sistemas baseados em FUSE têm desempenhos semelhantes, quando as aplicações submetem operações de escritas com tamanhos de 4 KiB. Contudo, quando são submetidas operações de escrita com tamanhos de 1024 KiB, os sistemas baseados em FUSE apresentam uma degradação de desempenho superior a 40%, quando comparados com o NFS.

Esta diferença de desempenho, nas operações de escrita com tamanhos de bloco de 1024 KiB, deve-se à limitação do tamanho de bloco imposta pelo FUSE. O cliente NFS opera com blocos de dimensões até 1024 KiB, permitindo assim que sejam enviados pedidos de escrita com estas dimensões para o servidor. O cliente RSafeFS, e os restantes clientes dos sistemas baseados em FUSE, devido ao limite imposto pelo FUSE, para o tamanho de bloco, só permite que sejam enviados pedidos de escrita até 128 KiB para o servidor. Fazendo assim com que os sistemas baseados neste gerem um maior volume de operações

a serem enviadas pela rede, bem como mais operações para persistir os dados realizadas pelo servidor. Para escritas com tamanho de bloco até 128 KiB o desempenho entre o NFS e os restantes sistemas baseados em FUSE é semelhante. Por exemplo, no ambiente com disco SSD, em *seq-write-1th-1f-4k* e *rand-write-1th-1f-4k* o RSafeFS e o NFS obtêm o mesmo desempenho, com um débito de 0.5 MiB/s e 0.6 MiB/s, respetivamente.

Nas escritas do tipo *16th-16f-4k*, o RSafeFS e a integração de FUSE com gRPC obtiveram melhores desempenhos que o NFS. Isto deve-se ao facto dos servidores destes sistemas estarem configurados para atender os pedidos com um maior número de *threads* de que um servidor NFS comum. Como na integração de FUSE com Cap'n Proto RPC o cliente é *single-thread*, não se verificam ganhos de desempenho relativamente ao NFS neste tipo de cargas. Em vez disso, observa-se uma degradação no desempenho.

Nos clientes assíncronos RSafeFS e NFS, como explicado em capítulos anteriores, nem todas as escritas efetuadas pelas aplicações geram operações de escrita para o servidor. Assim, com este tipo de cliente além do sistema responder às aplicações, depois de garantir que os dados das escritas foram armazenados em memória no cliente, em vez de persistidos no disco do servidor, permite que o cliente agrupe dados contíguos de escritas consecutivas, reduzindo depois o número de pedidos a serem enviados para o servidor.

Os resultados revelam que o cliente assíncrono do RSafeFS tem melhor desempenho que o cliente assíncrono do NFS. Esta diferença de desempenho deve-se aos diferentes modelos de coerência de cada um dos sistemas. Enquanto que o cliente assíncrono do NFS envia periodicamente para o servidor pedidos que forcem a persistência de dados através da operação *commit*, o cliente assíncrono do RSafeFS apenas realizam esses pedidos quando as aplicações o indicam explicitamente (através de operações *fsync*), ou quando os ficheiros são fechados. Como estas cargas de trabalho não geram operações que forcem a persistência de dados no servidor, o NFS acarreta uma penalização ao forçá-las implicitamente.

Apesar do FUSE-NFS também seguir o modelo de coerência implementado no RSafeFS, o desempenho é inferior. Isto porque, enquanto que o cliente assíncrono do RSafeFS tenta agrupar vários pedidos de escrita consecutivos de forma a reduzir o número de pedidos a enviar para o servidor, o FUSE-NFS envia sempre pedidos de escrita de tamanho 4 KiB, independentemente do tamanho escrito pelas aplicações. Tomando como exemplo as dimensões da carga de trabalho que escreve um ficheiro de 32 GiB, enquanto o RSafeFS, por agrupar agrupa escritas até 1 MiB, envia 32,768 operações para o servidor, o FUSE-NFS envia 8,388,608 operações de 4 KiB, traduzindo-se assim em 256$\times$ mais operações enviadas para o servidor.

Escrita aleatória. Neste tipo de teste, *N threads* escrevem aleatoriamente em *N* ficheiros, previamente alocados. Analogamente às escritas sequenciais, para se garantir que não ocorrem escritas concorrentes sobre o mesmo ficheiro, cada *thread* escreve dados num ficheiro distinto. Como os resultados anteriormente apresentados demonstram, o desempenho

dos sistemas, bem como a diferença de desempenho entre sistemas, neste tipo de teste é semelhante às escritas sequenciais. Desta forma, as razões apresentadas nos testes com escritas sequenciais justificam também as diferenças de desempenho nestas cargas de trabalho.

5.5.2 *Micro testes: metadados*

Nesta subsecção são analisados os resultados obtidos nos micro testes orientados a metadados, apresentados anteriormente nas Tabelas 7 e 8. Uma vez que o fluxo das operações de metadados segue o mesmo formato para clientes síncronos e assíncronos, estes apresentam resultados semelhantes. Desta forma, a análise destes resultados não distingue os dois tipos de clientes.

Criação de ficheiros (create). Nesta carga de trabalho, *N threads* criam múltiplos ficheiros de pequenas dimensões (4 KiB), distribuídos por várias diretorias previamente criadas. Como observado nos resultados obtidos, todos os sistemas baseados em FUSE apresentam uma degradação de desempenho, relativamente ao NFS, entre 5% a 25%.

Esta diferença de desempenho entre o NFS e os restantes sistemas baseados em FUSE, onde se inclui o RSafeFS, deve-se à diferença no número de operações realizadas entre cliente e servidor.

No sistema NFS, por cada diretoria onde são criados os ficheiros, são realizadas duas operações: *lookup* para validar se a diretoria existe e *access* para verificar as permissões de acesso. Após estes passos, para criar cada ficheiro são realizadas três operações: *lookup* para verificar se já existe (nesta carga de trabalho a resposta é sempre negativa), *create* para criar o ficheiro, e por fim a operação *write* para escrever o conteúdo do ficheiro.

No que diz respeito aos sistemas de ficheiros baseados em FUSE e RSafeFS, estes compreendem um maior número de operações comparativamente ao sistema NFS, porque necessitam de responder aos pedidos submetidos pelo módulo FUSE. Assim, nestes sistemas são realizadas seis operações para criar e escrever o ficheiro. Inicialmente, é realizada a operação *getattr* para verificar que o ficheiro ainda não existe. Só depois são realizadas as seguintes operações para criar e escrever no ficheiro: *create* para criar o ficheiro, *getattr* para gestão de *caches* internas do FUSE, *write* para escrever dados no ficheiro, *flush* para fechar o ficheiro, e *release* para sinalizar que não há mais referências para o ficheiro aberto.

Os sistemas baseados em FUSE também realizam várias operações *getattr* para obter os atributos das diretorias onde os ficheiros são criados. No NFS, estas operações não são transmitidas para o servidor porque são respondidas localmente pela *cache* de atributos utilizada pelo cliente. Como referido anteriormente, esta *cache* do cliente NFS é atualizada por operações que incluem atributos nos resultados [39]. Em concreto, a resposta à operação

create contém os atributos da diretoria onde o ficheiro é criado, evitando assim que o cliente precise de enviar um pedido *getattr* para o servidor para obter esses atributos.

A camada de *caching* de metadados utilizada no RSafeFS não influencia positivamente o desempenho do sistema, uma vez que os pedidos nunca conseguem ser respondidos pela camada. Em detalhe, nos sistemas baseados em FUSE são realizadas duas operações *getattr* durante a criação do ficheiro: a primeira, para verificar se um ficheiro existe, nunca é respondida pela *cache*, porque este ficheiro ainda não existe; a segunda, não consegue ser respondida pela *cache* porque é o primeiro pedido após o ficheiro ser criado. Os restantes pedidos *getattr*, por exemplo, sobre as diretorias que contêm os ficheiros, também não conseguem ser respondidos pela *cache*, porque entretanto esses atributos foram invalidados por outras operações. Neste caso através da operação *create*, pois na eventualidade desta operação ter sucesso no servidor os atributos da diretoria onde o ficheiro é criado são modificados.

Ao contrário do que acontece no NFS, a camada de *caching* de metadados utilizada no RSafeFS não é atualizada por atributos enviados nas respostas a outras operações (p.e., *create*, *mkdir*, *remove*), porque estes atributos não são enviados nas respostas destas operações. Este tipo de otimizações não conseguem ser desenvolvidas no RSafeFS, porque este sistema é construído sobre a premissa de camadas independentes, onde a comunicação entre camadas é realizada através da API do FUSE. Como na API do FUSE as respostas às operações apenas devolvem um valor inteiro para indicar sucesso/insucesso das operações, não é possível devolver atributos de ficheiros e/ou diretorias nestas respostas.

Remoção de ficheiros (delete). Nesta carga de trabalho, *N threads* vão continuamente removendo ficheiros previamente alocados e distribuídos por várias diretorias. Os resultados obtidos revelam que, em relação ao NFS, os sistemas baseados em FUSE obtêm uma diferença de desempenho compreendida entre -5% e -25%,

O sistema NFS, para cada remoção de um ficheiro, realiza duas operações: *lookup* para verificar se o ficheiro existe, e *remove* para remover o ficheiro. Adicionalmente, cada vez que um pedido acede a uma determinada diretoria pela primeira vez, são também realizadas as operações *lookup* e *access*, para verificar, respetivamente, a existência diretoria no sistema de ficheiros e as permissões a acessibilidade.

Nos sistemas baseados em FUSE, nos quais se inclui o RSafeFS, são também utilizadas duas operações para remover um ficheiro: *getattr* para verificar que o ficheiro existe, seguida da operação *unlink* para remover o ficheiro. Ao contrário do NFS, em vez da operação *access* por cada primeiro acesso à diretoria, é feita a operação *getattr*.

O volume de operações realizadas entre o NFS e os sistemas baseados em FUSE é semelhante. Contudo, analisando os resultados em detalhe, é possível observar que o sistema FUSE-NFS também apresenta uma degradação de desempenho face ao NFS idêntica à dos restantes cenários construídos em FUSE. Uma vez que este sistema utiliza o mesmo

protocolo para comunicar com o servidor NFS, este gargalo no desempenho está diretamente relacionado com custo de introduzir o FUSE no sistema [74].

As diferentes configurações da camada de *caching* de metadados utilizada no RSafeFS também não trazem ganhos de desempenho nesta carga de trabalho. Seria expectável que a utilização desta *cache* melhorasse o desempenho do RSafeFS, uma vez que os metadados utilizados durante a pré-alocação dos ficheiros poderiam ser mantidos em *cache* e utilizados durante a remoção dos ficheiros. De facto, eles são mantidos na *cache*, mas o facto da carga ser sequencial - os primeiros ficheiros a serem criados vão ser os primeiros a serem removidos - e o volume da carga ser considerável, faz com que desde a criação de um ficheiro até a sua remoção passem vários minutos, ultrapassando assim os períodos utilizados (10 e 60 segundos) nas configurações da camada de *caching* para validade dos metadados.

Leitura de ficheiros (read). Nesta carga de trabalho, *N threads* vão continuamente ler o conteúdo de vários ficheiros previamente distribuídos por várias diretorias. Embora sejam realizadas leituras sobre dados, esta carga de trabalho é considerada orientada a metadados porque visa abrir e fechar milhares de ficheiros de pequenas dimensões (4 KiB) ao longo de toda a execução. Como anteriormente observado, todos os sistemas baseados em FUSE obtêm uma degradação de desempenho superior a 50%, quando comparados com o NFS.

No sistema NFS, como não há operações para abrir e fechar ficheiros, devido a natureza *stateless* do servidor, a leitura de um ficheiro origina três operações a serem enviadas para o servidor: *lookup* para verificar se o ficheiro existe, *access* para verificar as suas permissões de acesso, e uma operação *read*, que dada a dimensão dos ficheiros consegue ler todo o seu conteúdo.

No RSafeFS, e nos restantes sistemas baseados em FUSE, cada leitura de um ficheiro é antecedidas das operações para abri-lo, nomeadamente *getattr* para verificar que o ficheiro existe, e *open* para abri-lo; e sucedida das operações para fechá-lo, nomeadamente *flush* para fechá-lo, e *release* para sinalizar que não há mais referências para o ficheiro aberto. Depois do ficheiro ser aberto, e antes de ser fechado, para ler o conteúdo do ficheiro, são realizadas a seguintes operações: *read*, *read* e *getattr*. Apesar da primeira operação *read* ler todo o conteúdo do ficheiro (4 KiB), como a aplicação (*filebench*) tenta ler depois do final do ficheiro, o FUSE gera mais uma operação leitura, que por sua vez gera uma operação *getattr* para confirmar o tamanho do ficheiro [74]. Isto penaliza severamente o desempenho dos sistemas baseados em FUSE, pois como esta carga realiza leituras sobre ficheiros de pequenas dimensões, uma operação de leitura seria suficiente para ler a totalidade do ficheiro.

Relativamente ao desempenho obtido com a camada de *caching* de metadados utilizada no RSafeFS e respetivas configurações, não é possível retirar conclusões válidas, uma vez que os resultados apresentam um desvio padrão elevado.

Então, no sentido de analisar ganhos de desempenho trazidos pela *cache* de atributos utilizada por padrão no NFS, para esta carga de trabalho avaliou-se também o desempenho

do NFS sem ela. Introduzindo assim no sistema o custo do cliente ter de enviar para o servidor os pedidos *getattr*, que anteriormente eram respondidos pela *cache* no cliente. Os resultados, apresentados na Tabela 11, revelam que o NFS sem esta *cache* de atributos tem uma degradação média de desempenho de 80.8%, relativamente à configuração padrão do NFS. Ainda assim, esta configuração obtém melhor desempenho que os sistemas baseados em FUSE, incluindo o FUSE-NFS, que obtém uma degradação média de 92.2%, relativamente ao NFS padrão nesta carga de trabalho.

Desta forma, uma vez que comparativamente ao NFS, os sistemas baseados em FUSE invocam mais operações para cada leitura, a diferença de desempenho está diretamente relacionada com o volume de operações realizadas, bem como com a sobrecarga introduzida pelo FUSE para todas as operações POSIX [74].

Tabela 11: Resultados do débito do NFS com o cliente a não fazer *caching* de atributos.

	SSD				NVMe			
	NFS ops/s	**NFS-noac** rel. diff. %	**FUSE-NFS** rel. diff. %	**RSafeFS** rel. diff. %	**NFS** ops/s	**NFS-noac** rel. diff. %	**FUSE-NFS** rel. diff. %	**RSafeFS** rel. diff. %
read-1th	6020.2	-78.4	-90.9	-89.8	6037.0	-83.5	-89.7	-88.4
read-16th	57075.8	-80.2	-93.7	-90.4	60188.8	-81.2	-94.6	-89.6

Verificação de estado de ficheiros (stat). Nesta carga de trabalho *N threads* verificam o estado de ficheiros, previamente distribuídos por várias diretorias. Mais concretamente, durante a execução desta carga, são realizadas, no máximo, 32 milhões de operações de verificação de estado de 500 mil ficheiros, levando assim que o estado dos ficheiros seja verificado várias vezes durante a execução da carga. Os resultados obtidos neste tipo de teste demonstram que todos os sistemas baseados em FUSE obtêm uma degradação de desempenho superior a 50%, relativamente o NFS.

Por cada verificação de estado de ficheiros, o volume de operações enviadas pelos clientes para os servidores dos sistemas NFS e baseados em FUSE, nos quais se inclui o RSafeFS, é semelhante. No sistema NFS, na resposta à operação *lookup* - para obter o identificador do ficheiro e verificar que o ficheiro existe - são também enviados os atributos do ficheiro. Sendo depois estes lidos da *cache* do cliente NFS para responder ao pedido da aplicação. Nos sistemas baseados em FUSE, são enviados pedidos *getattr* para os servidores, no mesmo número de pedidos *lookup* enviados pelo cliente ao servidor no sistema NFS.

Visto que, para verificar o estado de um ficheiro, o NFS e os sistemas baseados em FUSE enviam o mesmo número de operações para os servidores, a diferença de desempenho entre

estes sistemas, deve-se à sobrecarga introduzida pelo FUSE neste tipo de teste. Os resultados obtidos pelo FUSE-NFS comprovam esta sobrecarga introduzida pelo FUSE. Apesar deste sistema utilizar o protocolo NFS para comunicar com o servidor, o FUSE-NFS apresenta uma degradação de desempenho superior a 90%, em relação ao NFS.

Relativamente à camada de *caching* de metadados utilizada no RSafeFS, esta apenas melhora o desempenho do sistema quando são utilizadas 16 *threads* para gerar as operações de verificação de ficheiros (*stat-16th*), e quando é configurada com um período de validade dos metadados de 60 segundos. Analisando os resultados obtidos com RSafeFS, sem esta camada de *caching*, no ambiente com disco SSD, durante os 1200 segundos da execução de *stat-1th* são realizadas em média 6.8 milhões de operações de verificação de estado de ficheiros. Como esta carga é sequencial, ou seja, o estado de um ficheiro só é novamente verificado após o estado dos restantes ficheiros terem sido verificados, este volume de operações traduz-se em 13.6 operações de verificação por cada ficheiro. Desta forma, durante a execução do teste são necessários 88.2 segundos para o estado de um ficheiro ser novamente verificado. Como nestes cenários de avaliação, a camada de *caching* de metadados do RSafeFS foi configurada com períodos de validade de 10 e 60 segundos, sempre que é feita uma verificação do estado de um ficheiro, os metadados mantidos em *cache* já são considerados inválidos, pois o período de validade já foi ultrapassado.

Quando são utilizadas 16 *threads* para gerar a carga, o RSafeFS sem camadas de processamento, realiza 32 milhões de operações de verificação de estado de ficheiros em 735 segundos. Este volume de operações traduz-se assim em 11.5 segundos entre operações de verificação de estado sobre o mesmo ficheiro. Desta forma, a configuração da camada de *caching* utilizada no RSafeFS, quando configurada com o período de validade de 60 segundos permite melhorar o desempenho do sistema, em média, 1.5×, relativamente ao RSafeFS sem camadas de processamento.

Conclui-se assim, que a camada de *caching* de metadados utilizada no RSafeFS é eficiente para este tipo de cargas (onde os ficheiros são raramente ou nunca modificados), quando configurada corretamente.

5.5.3 *Macro testes*

Nesta secção, discutimos o desempenho dos sistemas de ficheiros NFS, baseados em FUSE, e RSafeFS para os macro testes, cujas cargas de trabalho visam simular ambientes mais próximos da realidade, nomeadamente um servidor de ficheiros (*file-server*), um servidor de e-mails (*mail-server*), e um servidor web (*web-server*). Estes resultados estão apresentados nas Tabelas 9 e 10.

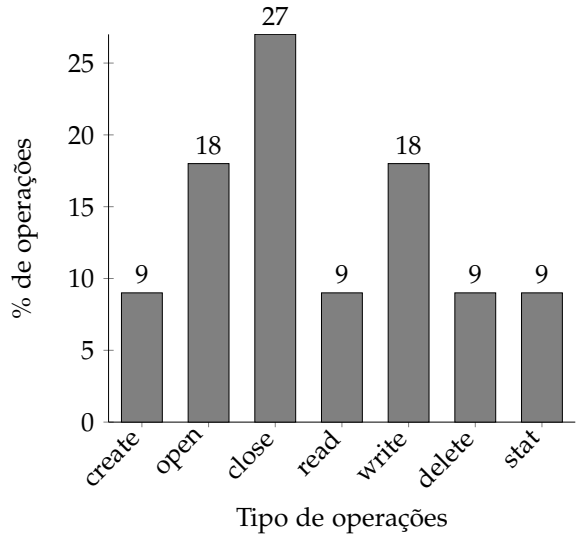


Figura 8: Distribuição do tipo de operações geradas pelo *filebench* durante a execução da carga de trabalho *file-server*.

Servidor de ficheiros (file-server). Esta carga de trabalho é constituída por 50 *threads*, em que cada uma está repetidamente a realizar o seguinte conjunto de tarefas: criar um ficheiro de pequenas dimensões (128 KiB), escrever aleatoriamente 16 KiB de dados noutro ficheiro, ler todo o conteúdo de um ficheiro, remover um ficheiro, e verificar o estado de um ficheiro. A Figura 8 apresenta a distribuição do tipo de operações geradas durante a execução desta carga de trabalho: 54% estão relacionadas com a abertura, criação e fecho de ficheiros; 18% com a escrita de dados; 9% com a leitura de dados; e os restantes 18% repartidos de forma semelhante entre a verificação de estado de ficheiros e remoção de ficheiros. De notar que o somatório destas percentagens é de 99%, pois na figura são apresentados valores aproximados para não serem apresentados com casas decimais. Importa ainda referir que estas percentagens, relativas ao tipo de operações realizadas, são da perspetiva da aplicação. Os sistemas avaliados podem variar o número de operações que realizam para responderem às operações geradas pela aplicação. Por exemplo, numa leitura com um tamanho de bloco de 1024 KiB, enquanto que o NFS necessita de uma operação para efetuar a leitura dos dados, os sistemas baseados em FUSE, como referido anteriormente, necessitam de oito operações.

Como observado nos resultados obtidos, a degradação de desempenho dos sistemas baseados em FUSE e RSafeFS comparativamente ao NFS é significativa, com os sistemas baseados em FUSE a apresentarem uma degradação de desempenho superior a 50% relativamente ao NFS. Analisando o tipo de operação realizadas, verifica-se que, esta carga de trabalho conjuga as operações realizadas nos micro testes orientados a metadados anteriormente analisados. Assim sendo, é possível concluir que esta diferença de desempenho está relacio-

nada com: (1) o volume de operações realizadas; (2) a sobrecarga introduzida pelo FUSE no processamento das operações; (3) *caching* de dados e metadados realizado pelo NFS.

Em maior detalhe, como referido na análise dos micro testes orientados a metadados, nos sistemas baseados em FUSE as operações de leitura e a escrita de dados são antecedidas e sucedidas de várias operações relacionadas com a abertura e fecho de ficheiros (p.e, *open*, *flush*, *release*), enquanto que no sistema NFS - porque o servidor é *stateless* - apenas são realizadas as operações *read* e *write* na leitura e escrita de dados em ficheiros. Também como observado no micro teste *read*, os sistemas baseados em FUSE realizam mais operações na leitura do conteúdo dos ficheiros, uma vez que o FUSE gera essas operações quando a aplicação tenta ler dados depois do fim do ficheiro. Tal como no micro teste *read*, isto penaliza severamente o desempenho destes sistemas, pois como são manipulados vários milhares de ficheiros de pequenas dimensões, muita das vezes, uma operação de leitura é suficiente para ler todo o conteúdo de dos ficheiros.

Nos sistemas baseados em FUSE também são enviadas mais pedidos *getattr* para o servidor comparativamente com o NFS. Estes pedidos não são enviados no NFS, porque conseguem ser respondidos pela *cache* de atributos/metadados do cliente do NFS.

Como observado, a camada de *caching* de metadados utilizada pelo RSafeFS não melhora o desempenho do sistema. A razão para não haver melhoria no desempenho com a sua utilização são as várias operações que invalidam os metadados armazenados. Considerando as operações geradas pela aplicação durante esta carga de trabalho, as que implicam invalidar metadados da *cache* de metadados do RSafeFS, para manter a coerência, são: criação de ficheiros (*create*), pois caso tenham sucesso, os atributos das diretorias onde os ficheiros são criados são modificados no servidor; fechar ficheiros (*close*), porque previamente poderiam ser realizadas operações que modificam o tamanho do ficheiro (p.e, escritas de dados); e remoção de ficheiros (*delete*), uma vez que se tiver sucesso, os atributos das diretorias que continham os ficheiros são modificados.

De salientar novamente que, no NFS estas operações, que implicam invalidar metadados da *cache*, recebem também como resultado os atribuídos dos ficheiros/diretorias modificados no servidor, permitindo assim eficientemente atualizar a *cache* de atributos/metadados do cliente.

Relativamente à camada de *caching* de dados utilizada no RSafeFS, uma análise detalhada das razões para não apresentar uma melhoria de desempenho, será realizada conjuntamente com os restantes testes macro, na análise do último macro teste (*web-server*).

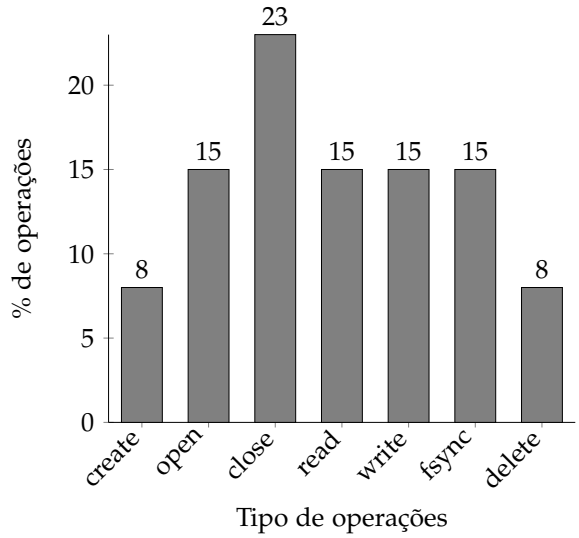


Figura 9: Distribuição do tipo de operações geradas pelo *filebench* durante a execução da carga de trabalho *mail-server*.

Servidor de e-mails (mail-server). Esta carga de trabalho utiliza 16 *threads* para gerar vários tipos de operações, que combinadas procuram simular um servidor de e-mails. À semelhança da carga *file-server*, as operações geradas estão relacionas com a criação, escrita, leitura e remoção de ficheiros de pequenas dimensões. Ainda, no *mail-server* sempre que são escritos dados num ficheiro, a aplicação realiza uma operação para garantir explicitamente a persistência destes. A Figura 9 apresenta a distribuição do tipo de operações geradas durante a execução deste macro teste: 46% estão relacionadas com a criação, abertura e fecho de ficheiros; 15% com a leitura de dados; 15% com a escrita de dados; 15% com operações persistem dados; e 8% com a remoção de ficheiros.

Exceptuando a integração de FUSE com o Cap'n Proto RPC cuja degradação de desempenho é superior a 50%, devido à implementação *single-thread* do cliente, os restantes sistemas baseados em FUSE apresentam uma diferença de desempenho compreendida entre -25% e -50%, quando comparados com o NFS.

Novamente, a diferença de desempenho entre o NFS e os sistemas baseados em FUSE, onde se inclui o RSafeFS, está relacionada com o número de operações transmitidas entre cliente e servidor, com sobrecarga introduzida pelo FUSE, e com *caching* eficiente de dados e metadados realizado no NFS. Ainda, relativamente à diferença no volume de operações transmitidas entre clientes e servidores, as operações geradas explicitamente pela aplicação para garantir a persistência dos dados, não são realizadas pelo cliente NFS. Importa referir que esta otimização não causa nenhum problema na coerência dos dados, podendo por isso também ser implementada nos sistemas baseados em FUSE (quando utilizados com cliente síncrono). Nesta carga de trabalho, o cliente assíncrono do NFS tem o mesmo comportamento que o cliente síncrono, ou seja, as operações *write* são definidas com a opção

file_sync, para que o servidor garanta que os dados dessa operação são persistidos antes de responder ao cliente. Assim sendo, como o cliente tem sempre a garantia que os dados estão persistidos em disco, quando a aplicação explicitamente pede para os dados serem persistidos o cliente ignora o pedido, não enviando assim a operação *commit* para o servidor.

Relativamente à camada de *caching* de metadados utilizada no RSafeFS, o motivo para não haver melhorias significativas no desempenho é o mesmo que identificado no macro teste anterior. Isto é, durante a execução da carga de trabalho são realizadas várias operações (p.e, *create*, *remove*, *fsync*) que implicam invalidar metadados armazenados na *cache* do cliente para manter a coerência.

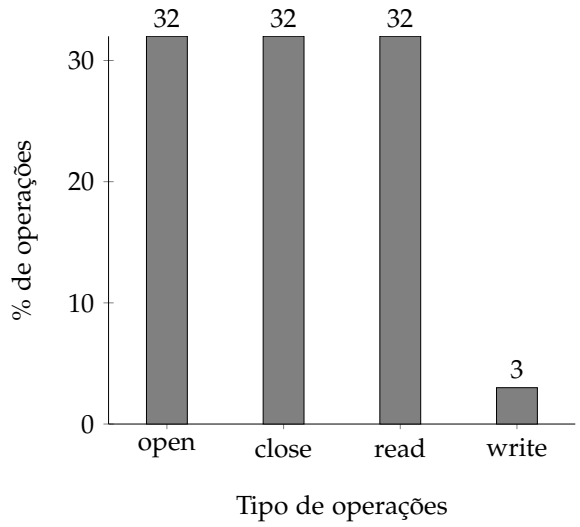


Figura 10: Distribuição do tipo de operações geradas pelo *filebench* durante a execução da carga de trabalho *web-server*.

Servidor web (web-server). A distribuição do tipo de operações geradas durante a execução desta carga de trabalho é apresentada na Figura 10. Contrariamente aos macro testes anteriormente analisados, este apenas compreende quatro tipo de operações. As operações relacionadas com a abertura, leitura e fecho de ficheiros são responsáveis, cada uma, por 32% das operações realizadas, já as operações relacionadas com escritas são responsáveis por apenas 3% das operações geradas. Cada uma das 100 *threads* que simulam esta carga lê vários ficheiros pré-alocados de 16 KiB (nunca modificados durante a execução), e periodicamente escreve até 16 KiB de dados para um ficheiro partilhado por todas as *threads*, na forma de *append*, simulando um registo de eventos.

Os resultados revelam que nos sistemas com clientes síncronos, os baseados em FUSE apresentam uma degradação de desempenho é compreendida entre 25% e 50%, relativamente ao NFS, já com os clientes assíncronos a degradação é superior a 50%.

A diferença de desempenho entre o NFS e os restantes sistemas baseados em FUSE, onde se inclui o RSafeFS, é uma vez mais explicada pela diferença no número de operações realizadas entre o cliente e o servidor, e pela sobrecarga introduzida pelo FUSE no processamento das operações.

Durante a leitura de ficheiros, os sistemas baseados em FUSE e RSafeFS são penalizados pelos mesmos problemas identificados anteriormente, ou seja, são realizadas várias operações, nomeadamente para abrir e fechar ficheiros, e no caso da aplicação tentar ler depois do fim do ficheiro (como é o caso) são feitas mais duas operações (*read* e *getattr*).

Em relação à camada de *caching* de metadados utilizada pelo RSafeFS, verifica-se, tal como em *stat-16th*, que esta camada permite melhorar o desempenho do sistema para este tipo de cargas em que os ficheiros são raramente ou nunca modificados, e que as operações realizadas sobre eles não implicam invalidar metadados armazenados em *cache*. Assim, por exemplo, no cliente síncrono RSafeFS com a camada de *caching* configurada com 300 MiB de capacidade máxima e 60 segundos como período de validade máxima para os metadados, observam-se ganhos médios de desempenho de 15% no ambiente com disco SSD, e de 26% no ambiente com disco NVMe, relativamente o RSafeFS sem estas camadas de processamento.

Nos macro testes, analisando os pedidos transmitidos entre cliente e servidor NFS, verifica-se que o cliente procura diminuir o número de operações de leitura enviadas para o servidor utilizando para isso dados que tem em memória para responder às aplicações. Isto é facilmente observado no servidor *web*, pois como os ficheiros nunca são modificados durante a execução da carga, passado algum tempo, o cliente deixa de enviar pedidos de leitura para o servidor porque este já contém os dados necessários para responder à aplicação. Em vez disso, o cliente envia um pedido *getattr* - quando a aplicação abre o ficheiro - para obter os tempos de modificação do ficheiro no servidor, e verificar se os dados que tem em memória ainda são válidos.

Como isto não é realizado pelo FUSE, os sistemas baseados neste, apresentam perdas desempenhos significativas, quando comparados com o NFS. Neste sentido, para o melhorar o desempenho do RSafeFS foi desenvolvida uma camada de realiza *caching* de dados, para que, quando utilizada no cliente, possa permitir diminuir o número de operações enviadas para o servidor. Ainda, esta camada foi desenvolvida para oferecer as mesmas garantias de coerência que os mecanismos de *caching* de dados utilizados pelo NFS. Isto é, os dados armazenados em memória são revalidados quando os ficheiros são abertos, através da verificação dos atributos do ficheiro [52]. Contudo, como se observa nas Tabelas 9 e 10 os resultados obtidos com esta camada não melhoram o desempenho do sistema.

Na tentativa de se perceber quais os motivos para não serem observados ganhos de desempenho, testaram-se diferentes configurações de *caching* para a esta camada. Como anteriormente referido, esta camada é configurada, para além da política de substituição

dos blocos, através de três parâmetros: o limite máximo de memória utilizada, o tamanho dos blocos armazenados, e o período de validade dos blocos. Nos resultados anteriormente apresentados, a camada de *caching* de dados utilizada no RSafeFS foi configurada com 1 GiB como limite máximo de memória utilizada, os blocos tinham como dimensão 128 KiB, e o tempo de validade dos blocos foi de 30 segundos. O tamanho dos blocos foi configurado com dimensão de 128 KiB, para permitir operar com blocos de igual dimensão aos utilizados (no máximo) pelo FUSE, possibilitando assim, que sejam realizadas menos operações de cópia de dados para responder aos pedidos submetidos pelo FUSE.

Contudo esta dimensão do bloco revelou não ser apropriada para estes macro testes, devido a dimensão dos ficheiros utilizados durante as execuções das cargas. Nestas cargas, os ficheiros têm pequenas dimensões (p.e, 16 KiB), assim, quando utilizados blocos com 128 KiB, o tamanho dos blocos é, na maioria das vezes, superior ao tamanho dos ficheiros. Como os blocos são alocados na totalidade, apenas uma pequena parte do bloco é ocupada com dados dos ficheiros, ficando a restante parte desaproveitada, e sem poder ser utilizada por outros ficheiros. Por sua vez, isto faz com que rapidamente a quantidade de memória máxima alocada pela *cache* seja atingida, e que os blocos estejam constantemente a ser removidos da *cache*, para serem colocados novos blocos. Assim sendo, configurar a camada de *caching* de dados utilizada no RSafeFS com tamanhos de blocos mais pequenos poderia permitir uma gestão mais eficiente da memória utilizada, permitido assim que mais dados sejam armazenados em *cache*. Neste sentido, a camada de *caching* de dados do RSafeFS foi configurada para utilizar 16 KiB como tamanho de bloco.

Tabela 12: Resultados do débito dos clientes síncronos RSafeFS, com diferentes configurações da camada de *caching* da dados.

	SSD						NVMe					
	NFS ops/s	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-16KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-16KiB** rel. diff. %	**NFS** ops/s	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-16KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-16KiB** rel. diff. %
file-server	826.5	-65.2	-58.4	-57.2	-26.2	-25.8	1088.4	-66.0	-60.4	-51.5	-19.3	-18.5
mail-server	668.2	-44.7	-38.4	-38.9	-25.0	-24.1	827.6	-46.4	-44.7	-38.6	-22.3	-23.6
web-server	4370.9	-39.2	-38.1	-38.2	-16.7	-16.5	5034.6	-45.0	-46.8	-46.8	-17.0	-17.1

A Tabela 12 apresenta o desempenho (débito) atingido pelo sistema RSafeFS, com o cliente síncrono, e com as configurações da camada de *caching* de dados com blocos de 16 KiB. Estas novas configurações foram designadas por RSafeFS-dc-lru-30s-1GiB-16KiB e RSafeFS-dc-rnd-30s-1GiB-16KiB. Os resultados obtidos demonstram que, quando a camada é configurada com estas definições, a diferença de desempenho relativamente ao NFS diminui, em média, para -22.5% em *file-server*, -23.8% *mail-server* e -16.8% em *web-server*. Esta diminuição da diferença de desempenho traduz-se, relativamente o RSafeFS sem camadas de processamento, numa melhoria de desempenho de 1.26×, 0.40× e 0.44×, nas cargas *file-server*, *mail-server* e *web-server*, respetivamente.

Como se observa, a configuração com os blocos com tamanho de 128 KiB, revela-se uma má configuração, quando comparada com a de blocos de 16 KiB. Mesmo na carga *file-server*, onde os ficheiros têm tamanhos superiores a 128 KiB, a configuração da camada de *caching* com tamanho de blocos de 16 KiB demonstrou obter melhor desempenho. Isto permite concluir que mesmo nas cargas, onde os ficheiros têm dimensões superiores ao blocos da *cache*, tamanhos de blocos mais pequenos permitem uma gestão mais eficiente da *cache*. Isto porque, como os dados são armazenados em blocos com menores dimensões, muito provavelmente, quando for necessário retirar blocos da *cache*, nem todos serão do mesmo ficheiro.

5.6 sumário

Nesta secção realiza-se um sumário das observações mais relevantes, retiradas durante o desenvolvimento do sistema e através da avaliação experimental deste. As observações que se seguem identificam os benefícios trazidos pelo RSafeFS, quais os principais gargalos de desempenho encontrados no sistema, e de que forma poderiam ser ultrapassados.

■ Observação 1. A modularidade e flexibilidade trazida pelo RSafeFS permite facilmente adicionar novas funcionalidades a sistemas de ficheiros remotos. Como prova disso, as camada de *caching* desenvolvidas para o RSafeFS permitiram melhorar significativamente o desempenho do sistema, para certas cargas de trabalho, quando corretamente configuradas. Por exemplo, nos micro testes orientados a metadados, nomeadamente na verificação de estado de ficheiros, a utilização da camada de *caching* de metadados, permitiu melhorar o desempenho do sistemas em 1.5×; nos macro testes, a camada de *caching* de dados permitiu melhorar o desempenho entre 0.4× e 1.26×. Estes ganhos de desempenho demonstram que os sistemas de ficheiros remotos desenvolvidos através do RSafeFS conseguem facilmente ser ajustados aos requisitos de diferentes aplicações e tipos de cargas de trabalho.

■ Observação 2. Em contrapartida, a simplicidade de desenvolvimento e integração de novas camadas no RSafeFS, devido à API do FUSE, torna difícil o desenvolvimento de

otimizações. Por exemplo, no NFS, a *cache* de atributos/metadados é atualizada através das respostas a outras operações. Por exemplo, na resposta à operação que cria um ficheiro, são também enviados os atributos do ficheiro criado e da diretoria onde o ficheiro é criado. Isto permite tornar o sistema mais eficiente, porque fazendo *caching* desta informação evita que o cliente tenha de enviar um pedido para obter essa informação. No RSafeFS não é possível desenvolver esta otimização, porque as camadas cumprem a API do FUSE, onde as respostas às operações apenas devolvem apenas um valor inteiro. Como forma de solucionar este problema, seria interessante estender a API do FUSE ou a API das camadas do SafeFS, para permitir enviar mais informação nas respostas às operações. Contudo, importa referir que, no caso de extensão da API das camadas do RSafeFS, perderia-se a compatibilidade de integração de sistemas de ficheiros baseados em FUSE como camadas independentes.

■ Observação 3. O tamanho de bloco máximo utilizado pelo FUSE (128 KiB) nas operações de escrita e leitura é a principal razão pela perda de desempenho do RSafeFS nos micro testes orientados a dados. Quando o NFS e RSafeFS utilizam o mesmo tamanho de bloco nas operações enviadas para o servidor (p.e., escritas com tamanho de bloco inferior a 128 KiB, nos sistemas com clientes síncronos) os desempenhos do sistemas são comparáveis. Por isso, a utilização de blocos com dimensões superiores a 128 KiB, por exemplo até 1024 KiB, nas operações de escrita e leitura de dados por parte do FUSE permitiria melhorar o desempenho do RSafeFS.

■ Observação 4. Relativamente ao NFS, o RSafeFS é penalizado nas cargas de trabalho que manipulam vários ficheiros dimensões, por realizar as várias operações relacionadas com a abertura e fecho de ficheiros (p.e., *open*, *flush* e *release*). Na leitura de ficheiros, quando as aplicações realizam leituras após o fim dos ficheiros, o RSafeFS, por ter responder ao pedidos submetidos pelo FUSE, é penalizado por realizar mais duas operações, nomeadamente a operação de leitura, e uma operação *getattr*, para confirmar o tamanho do ficheiro. Isto revela-se muito penalizador em ficheiros de pequenas dimensões, podendo, no pior dos casos, aumentar para o triplo o número de operações realizadas durante a leitura do ficheiro. Ainda, foram observadas sobrecargas consideráveis introduzidas pelo FUSE em algumas operações, por exemplo, remoção e verificação de estado de ficheiros.

6

CONCLUSÃO

No âmbito desta dissertação foi desenvolvido o sistema RSafeFS, uma plataforma que permite desenvolver de forma modular e flexível diferentes sistemas de ficheiros remotos. Neste sentido, a análise do estado da arte incidiu nos sistemas de ficheiros modulares e nos sistemas de ficheiros remotos.

Analisando ambos os tipos de sistemas de ficheiros (modulares e remotos) surgem dois problemas. O SafeFS apresenta-se como referência nos sistemas modulares, contemplando uma arquitetura modular e extensível, porém esta modularidade e flexibilidade não é trazida para cenários de armazenamento remoto. Por outro o NFS - estado de arte dos sistemas de ficheiros remotos -, uma vez que está desenvolvido em *kernel*, é difícil estendê-lo com novas funcionalidades de forma a torná-lo adaptável para diferentes casos de uso.

Assim, esta dissertação teve como objetivo desenvolver a plataforma RSafeFS com uma arquitetura modular e flexível, para permitir simplificar o desenvolvimento de sistemas de ficheiros remotos. Avançando o estado de arte, apresentado um sistema de ficheiros que conjuga as primitivas de modularidade e flexibilidade do SafeFS com as capacidades de armazenamento remoto do NFS. Para isto foi necessário desenvolver uma camada que permitisse uma instância RSafeFS operar como um servidor do sistema, isto é, que seja capaz de receber pedidos de clientes, e que consiga enviar-lhes as respostas, depois de efetuar o processamento necessário. Ainda, foi também necessário desenvolver uma camada de comunicação, baseada em protocolos remotos, para permitir a interoperabilidade entre instâncias cliente e servidor do sistema.

No sentido de melhorar o desempenho do protótipo, e de demonstrar a facilidade de integração de novas funcionalidades foram também desenvolvidas duas camadas de processamento, nomeadamente *caches* de dados e metadados. Estas oferecem garantias semelhantes aos mecanismos de *caching* encontrados no NFS, e permitem melhorar significativamente o desempenho do sistema, quando configuradas corretamente. A utilização da camada de *caching* de metadados permitiu melhorar o desempenho do RSafeFS até 1.5× num dos micro testes orientados a metadados, já a utilização da camada de *caching* de dados permitiu melhorar o desempenho do sistema até 1.26× nos macro testes.

Para avaliar o protótipo RSafeFS, foi também testado o NFS e um outro sistema baseado neste, em que o cliente é implementado através da plataforma FUSE. Com esta avaliação foi possível perceber o impacto do FUSE e das primitivas do RSafeFS na construção de sistemas de armazenamento remotos. Esta comparação revelou que o desempenho do protótipo implementado é penalizado por algumas das características do FUSE, nomeadamente o tamanho máximo de bloco limitado a 128 KiB, elevado número de operações para operar cargas de trabalho que manipulam vários ficheiros de pequenas dimensões, e pela sobrecarga introduzida pelo FUSE nas operações. Relativamente às características herdadas do SafeFS, conclui-se, como esperado, que o esforço de implementação de novas camadas de processamento que permitem dotar o sistema com novas funcionalidades é reduzido. Contudo, torna-se difícil desenvolver otimizações, uma vez que as camadas têm de cumprir a API (rígida) do FUSE.

Em suma, esta dissertação apresenta uma plataforma que permite a utilizadores e programadores desenvolverem sistemas ficheiros remotos modulares e flexíveis, através de diferentes combinações de camadas de processamento independentes.

6.1 TRABALHO FUTURO

O protótipo apresentado nesta dissertação permite desenvolver sistemas de ficheiros remotos. Ainda, a arquitetura modular, flexível e extensível do protótipo permite que os sistemas de ficheiros remotos desenvolvidos através desta plataforma facilmente sejam dotados com novas funcionalidades tanto no cliente como no servidor, colmatando assim as limitações dos sistemas de sistemas de ficheiros remotos atuais. No sentido de melhorar a plataforma desenvolvida, o RSafeFS pode ser estendido com a implementação de novas camadas de funcionalidade, orientadas ao conteúdo (p.e, compressão, deduplicação, cifragem), que permitam dotar os sistemas de ficheiros remotos com funcionalidades pretendidas pelos utilizadores e aplicações para este tipo de sistemas.

Como trabalho futuro, numa outra perspetiva, seria interessante estender esta solução para operar sobre uma infraestrutura distribuída, onde cada nó de armazenamento apresenta uma estrutura modular que permitirá escolher quais as melhores funcionalidades (p.e, redução de espaço, *caching*) tendo e conta os requisitos das diferentes aplicações e recursos disponíveis nesse nó. Ainda, desenvolver um orquestrador/controlador que permita instanciar facilmente estes nós, bem como as suas configurações, numa infraestrutura distribuída. Desta forma, será possível obter um sistema de ficheiros distribuído modular e flexível, adaptável aos recursos e/ou necessidades de cada servidor, com a separação do fluxo de dados e metadados da orquestração de nós de armazenamento. Com isto, será possível trazer os benefícios desta solução (RSafeFS) para um cenário de armazenamento mais complexo que contempla *hardware* heterogéneo, e que dará suporte a várias aplicações com diferentes requisitos.

BIBLIOGRAFIA

[1] Amazon S3 web page. https://aws.amazon.com/s3. Visitado em: 08-01-2020.

[2] Boost web page. https://www.boost.org. Visitado em: 08-01-2020.

[3] Cap'n Proto web page. https://capnproto.org. Visitado em: 08-01-2020.

[4] Catfs. https://github.com/kahing/catfs. Visitado em: 08-01-2020.

[5] Cryfs. https://www.cryfs.org. Visitado em: 08-01-2020.

[6] dbxfs. https://github.com/rianhunter/dbxfs. Visitado em: 08-01-2020.

[7] Dropbox web page. https://www.dropbox.com. Visitado em: 08-01-2020.

[8] dstat - versatile tool for generating system resource statistics. https://linux.die.net/man/1/dstat. Visitado em: 08-01-2020.

[9] Encfs. https://vgough.github.io/encfs. Visitado em: 08-01-2020.

[10] exports - nfs server export table. https://linux.die.net/man/5/exports. Visitado em: 08-01-2020.

[11] A fuse module for nfsv3/4. https://github.com/sahlberg/fuse-nfs. Visitado em: 08-01-2020.

[12] gcc web page. https://gcc.gnu.org. Visitado em: 08-01-2020.

[13] Gcsf. https://github.com/harababurel/gcsf. Visitado em: 08-01-2020.

[14] Google Drive web page. https://www.google.com/drive. Visitado em: 08-01-2020.

[15] gRPC web page. https://grpc.io. Visitado em: 08-01-2020.

[16] iperf3: A tcp, udp, and sctp network bandwidth measurement tool. https://github.com/esnet/iperf. Visitado em: 08-01-2020.

[17] Lessfs. https://github.com/rootfs/lessfs. Visitado em: 08-01-2020.

[18] libfuse web page. https://libfuse.github.io. Visitado em: 05-11-2020.

[19] Lzo. http://www.oberhumer.com/opensource/lzo/. Visitado em: 08-01-2020.

[20] nfs - fstab format and options for the nfs file system. https://linux.die.net/man/5/nfs. Visitado em: 08-01-2020.

[21] ping - send icmp echo_request to network hosts. https://linux.die.net/man/8/ping. Visitado em: 08-01-2020.

[22] Protocol Buffers web page. https://developers.google.com/protocol-buffers. Visitado em: 08-01-2020.

[23] Quicklz. http://www.quicklz.com. Visitado em: 08-01-2020.

[24] s3fs. https://github.com/s3fs-fuse/s3fs-fuse. Visitado em: 08-01-2020.

[25] Samba web page. https://www.samba.org. Visitado em: 05-11-2020.

[26] Server message block (smb) protocol versions 2 and 3. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-smb2. Visitado em: 08-01-2020.

[27] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *International Workshop on Information Hiding*, pages 73–82. Springer, 1998.

[28] M. Belshe, R. Peon, and M. Thomson. Hypertext transfer protocol version 2 (http/2), 2015.

[29] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[30] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.

[31] P. J. Braam and R. Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[32] B. Callaghan, B. Pawlowski, and P. Staubach. Rfc1813: Nfs version 3 protocol specification, 1995.

[33] R. Card. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux, 1995*, 1995.

[34] B. Carpentieri. Efficient compression and encryption for digital data transmission. *Security and Communication Networks*, 2018, 2018.

[35] G. Carter, J. Ts, and R. Eckstein. *Using Samba: A File & Print Server for Linux, Unix & Mac OS X*. "O'Reilly Media, Inc.", 2007.

[36] M. Chen, D. Hildebrand, G. Kuenning, S. Shankaranarayana, B. Singh, and E. Zadok. Newer is sometimes better: An evaluation of nfsv4. 1. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 165–176, 2015.

[37] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[38] M. Eisler. Xdr: External data representation standard. *RFC 4506*, 2006.

[39] M. Eisler, R. Labiaga, and H. Stern. *Managing NFS and NIS: Help for Unix System Administrators*. "O'Reilly Media, Inc.", 2001.

[40] T. Esteves, R. Macedo, A. Faria, B. Portela, J. Paulo, J. Pereira, and D. Harnik. Trustfs: An sgx-enabled stackable file system framework. In *2019 38th International Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pages 25–30. IEEE, 2019.

[41] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems (TOCS)*, 12(1):58–89, 1994.

[42] C. R. Hertel. *Implementing CIFS: The Common Internet File System*. Prentice Hall Professional, 2004.

[43] C. R. Hertel. Server message block in the age of microsoft glasnost. *login Usenix Mag.*, 37(1), 2012.

[44] J. H. Howard et al. *An overview of the andrew file system*, volume 17. Carnegie Mellon University, Information Technology Center, 1988.

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

[46] S. R. Kleiman et al. Vnodes: An architecture for multiple file system types in sun unix. In *USENIX Summer*, volume 86, pages 238–247, 1986.

[47] R. Macedo, J. Paulo, J. Pereira, and A. Bessani. A survey and classification of software-defined storage systems. *ACM Computing Surveys (CSUR)*, 53(3):1–38, 2020.

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

[49] A. D. McDonald and M. G. Kuhn. Stegfs: A steganographic file system for linux. In *International Workshop on Information Hiding*, pages 463–477. Springer, 1999.

[50] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications magazine*, 32(9):33–38, 1994.

[51] B. Nowicki. Rfc 1094,"nfs: Network file system protocol specification," mar. 1989, 27 pages, sun microsystems. *Inc., Santa Clara, CA*.

[52] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. Nfs version 3: Design and implementation. In *USENIX Summer*, pages 137–152. Boston, MA, 1994.

[53] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow. The nfs version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*, 2000.

[54] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. Mercier, and R. Oliveira. Safefs: a modular architecture for secure user-space file systems: one fuse to rule them all. In *Proceedings of the 10th ACM International Systems and Storage Conference*, page 9. ACM, 2017.

[55] J. Postel et al. User datagram protocol. 1980.

[56] J. Postel et al. Transmission control protocol. 1981.

[57] J. Postel and J. Reynolds. File transfer protocol. 1985.

[58] D. Reinsel, J. Gantz, and J. Rydning. The digitization of the world from edge to core. *IDC White Paper*, 2018.

[59] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[60] D. S. Rosenthal. Evolving the vnode interface. In *USENIX Summer*, volume 99, pages 107–118. Citeseer, 1990.

[61] R. Sandberg, D. Goldberg, and S. Kleiman. Design and implementation of the sun network filesystem.

[62] B. Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *International Workshop on Fast Software Encryption*, pages 191–204. Springer, 1993.

[63] R. Sharpe. Just what is smb? *Oct*, 8:9, 2002.

[64] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Rfc3010: Nfs version 4 protocol, 2000.

[65] K. Shvachko, H. Kuang, S. Radia, R. Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.

[66] J. Sipek, Y. Pericleous, and E. Zadok. Kernel support for stackable file systems. In *Proc. of the 2007 Ottawa Linux Symposium*, volume 2, pages 223–227. Citeseer, 2007.

[67] B. Sosinsky. *Networking bible*, volume 567. John Wiley & Sons, 2009.

[68] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

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

[70] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking.

[71] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196, 2013.

[72] R. Thurlow. Rpc: Remote procedure call protocol specification version 2. Technical report, RFC 5531, May, 2009.

[73] S. C. Tweedie et al. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo*. Durham, North Carolina, 1998.

[74] B. K. R. Vangoor, V. Tarasov, and E. Zadok. 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, Santa Clara, CA, Feb. 2017. USENIX Association.

[75] S. R. Walli. The posix family of standards. *StandardView*, 3(1):11–17, 1995.

[76] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. 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, 2006.

[77] A. Westerlund and J. Danielsson. Arla: a free afs client. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 32–32, 1998.

[78] C. P. Wright, M. C. Martino, and E. Zadok. Ncryptfs: A secure and convenient cryptographic file system. In *USENIX Annual Technical Conference, General Track*, pages 197–210, 2003.

[79] E. Zadok. *Fist: a system for stackable file-system code generation*. Columbia University, 2001.

[80] E. Zadok and I. Badulescu. A stackable file system interface for linux. In *LinuxExpo Conference Proceedings*, volume 94, page 10, 1999.

[81] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical report, Technical Report CUCS-021-98, Computer Science Department, Columbia University, 1998.

[82] E. Zadok, I. Badulescu, and A. Shender. Extending file systems using stackable templates. In *USENIX Annual Technical Conference, General Track*, pages 57–70, 1999.

[83] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright. On incremental file system development. *ACM Transactions on Storage (TOS)*, 2(2):161–196, 2006.

[84] E. Zadok and J. Nieh. Fist: A language for stackable file systems. 2000.

A

CONSUMO DE RECURSOS COMPUTACIONAIS

CONSUMO DE RECURSOS COMPUTACIONAIS DURANTE A EXECUÇÃO DOS MICRO TESTES ORIENTADOS A DADOS PARA OS CENÁRIOS DE TESTE COM CLIENTE SÍNCRONO

Tabela 13: Resultados da utilização de CPU nos micro testes orientados a dados para os cenários de teste com cliente síncrono.

			SSD				NVMe			
			NFS % CPU	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**NFS** % CPU	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %
seq-read	1th-1f-4k	client	16.08	+15.4	+35.8	+34.3	25.11	−32.4	+12.6	+13.5
		server	4.30	+57.9	+109.8	+102.8	18.16	−46.3	−50.2	−50.3
	1th-1f-1024k	client	13.84	+25.5	+44.7	+53.7	21.87	−32.0	+23.5	+26.9
		server	3.88	+71.9	+128.4	+73.2	18.49	−43.3	−47.9	−48.9
	16th-16f-4k	client	16.90	+14.4	+74.7	+70.8	26.68	−31.9	+128.3	+119.9
		server	4.46	+87.7	+188.1	+196.6	18.82	−47.2	+138.0	+119.8
	16th-16f-1024k	client	13.94	+25.9	+95.1	+92.8	23.07	−28.3	+163.3	+159.3
		server	4.11	+114.8	+199.5	+212.4	19.46	−48.9	+121.4	+127.0
rand-read	1th-1f-4k	client	6.55	+28.4	+54.0	+55.1	6.58	+20.2	+75.8	+77.4
		server	4.82	−12.0	+14.1	+13.9	7.83	+13.2	−61.2	−63.7
	1th-1f-1024k	client	9.93	+19.0	+34.4	+33.8	13.02	+11.1	+91.8	+91.6
		server	4.20	−22.4	+13.3	+6.0	10.04	−6.3	−21.7	−37.5
	16th-16f-4k	client	19.08	−52.4	+227.4	+229.8	21.20	−60.5	+222.9	+221.2
		server	32.23	−87.6	+61.6	+61.8	38.48	−73.4	+6.9	+7.4
	16th-16f-1024k	client	14.93	−7.0	+76.8	+82.4	28.34	−44.4	+113.7	+113.8
		server	4.26	−12.2	+193.2	+177.9	21.78	−54.0	+100.7	+107.3
seq-write	1th-1f-4k	client	0.85	+27.1	+14.1	+12.9	0.62	+129.0	+108.1	+114.5
		server	0.71	+22.5	+9.9	+35.2	1.73	+23.1	+26.0	+29.5
	1th-1f-1024k	client	4.13	−46.2	−63.0	−61.0	4.28	−39.7	−58.4	−58.4
		server	3.01	−68.1	−69.8	−67.8	6.44	−53.0	−54.3	−54.3
	16th-16f-4k	client	0.32	+300.0	+1943.8	+1753.1	0.65	+135.4	+1189.2	+1189.2
		server	1.46	−41.8	+15.8	+8.9	5.50	−59.6	+50.9	+51.5
	16th-16f-1024k	client	6.96	−68.1	−11.6	−16.5	8.58	−65.4	+36.4	+24.2
		server	3.91	−78.0	−46.5	−50.4	30.09	−89.9	−47.0	−54.7
rand-write	1th-1f-4k	client	0.90	+21.1	+16.7	+17.8	0.82	+102.4	+97.6	+79.3
		server	0.67	+34.3	+20.9	+23.9	1.80	+15.0	+22.2	+19.4
	1th-1f-1024k	client	3.01	−27.9	−38.5	−48.2	3.58	−38.5	−53.9	−52.5
		server	2.08	−58.2	−49.0	−56.7	3.84	−40.9	−37.8	−37.5
	16th-16f-4k	client	0.46	+215.2	+1558.7	+1543.5	0.71	+131.0	+1207.0	+1185.9
		server	1.38	−36.2	+18.1	+10.9	4.93	−60.0	+63.3	+60.2
	16th-16f-1024k	client	5.08	−54.9	+27.8	+26.6	6.62	−65.1	+37.5	+32.8
		server	2.53	−70.0	−18.6	−19.8	8.93	−75.9	+1.0	−3.6

Tabela 14: Resultados da utilização de memória nos micro testes orientados a dados para os cenários de teste com cliente síncrono.

			SSD				NVMe			
			NFS MiB	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**NFS** MiB	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %
seq-read	1th-1f-4k	client	97.31	−12.7	−7.7	−7.7	87.70	−3.2	+4.1	+4.4
		server	113.81	−6.7	+6.3	−0.1	156.78	−2.8	+7.7	+3.2
	1th-1f-1024k	client	98.65	−12.4	−6.8	−5.3	89.15	−3.3	+4.0	+3.9
		server	113.50	−6.4	+6.8	−3.9	154.57	−1.2	+9.2	+4.8
	16th-16f-4k	client	103.09	−19.9	+27.0	+21.5	100.73	−17.5	+83.6	+84.1
		server	115.36	−7.8	+22.9	+14.3	161.26	−5.3	+73.1	+54.1
	16th-16f-1024k	client	119.40	−17.6	+29.7	+25.7	117.50	−16.0	+95.4	+86.2
		server	114.71	−7.6	+23.6	+14.5	156.87	−2.7	+77.9	+55.8
rand-read	1th-1f-4k	client	88.16	−5.8	−0.6	+0.1	82.56	+2.3	+6.0	+6.2
		server	116.39	−8.9	+4.1	−2.2	163.24	−5.6	+3.7	−0.3
	1th-1f-1024k	client	82.60	−0.8	+12.0	+12.9	82.09	+0.8	+13.1	+12.6
		server	115.37	−8.7	+4.9	−1.4	162.05	−4.6	+5.1	+1.2
	16th-16f-4k	client	80.76	+2.6	+24.0	+23.2	80.77	+2.1	+23.7	+25.7
		server	116.84	−9.0	+19.3	+12.3	164.47	−6.2	+69.4	+56.0
	16th-16f-1024k	client	98.65	−1.1	+48.3	+49.7	97.91	+0.1	+109.4	+115.3
		server	114.13	−7.0	+22.0	+14.9	160.27	−3.5	+74.2	+61.6
seq-write	1th-1f-4k	client	96.43	−12.3	−11.6	−9.2	103.71	−13.8	−16.4	−12.6
		server	119.08	−10.5	+0.6	−8.4	163.25	−7.9	+2.4	−2.0
	1th-1f-1024k	client	95.17	−9.6	−8.5	−6.9	104.91	−16.7	−17.8	−16.6
		server	117.84	−10.5	+3.1	−3.5	160.69	−6.3	+4.1	−0.2
	16th-16f-4k	client	85.02	−1.1	+13.2	+14.7	104.24	−13.2	−8.7	−6.3
		server	126.70	−16.3	−2.0	−8.0	175.22	−13.9	+57.0	+47.4
	16th-16f-1024k	client	100.63	−0.5	+18.4	+18.4	120.09	−12.1	−6.1	−3.2
		server	120.99	−12.2	+21.7	+14.0	166.20	−9.2	+76.0	+66.7
rand-write	1th-1f-4k	client	93.45	−8.3	−6.1	−4.8	101.36	−10.4	−6.5	−3.1
		server	120.20	−11.8	+2.5	−3.8	163.86	−8.2	+66.6	+56.4
	1th-1f-1024k	client	94.51	−6.9	−6.9	−5.9	102.47	−11.7	−7.6	−4.7
		server	117.46	−10.0	+4.2	−1.8	160.33	−6.0	+70.4	+60.0
	16th-16f-4k	client	87.06	+1.7	+24.0	+22.8	102.72	−10.3	−6.0	−3.3
		server	130.51	−18.3	+15.5	+6.9	176.61	−14.6	+67.7	+54.7
	16th-16f-1024k	client	102.96	+1.4	+21.7	+20.9	118.01	−9.6	−2.3	+0.7
		server	121.25	−12.7	+23.7	+16.6	163.61	−8.0	+81.3	+72.6

Tabela 15: Resultados da utilização da rede nos micro testes orientados a dados para os cenários de teste com cliente síncrono.

			SSD				NVMe			
			NFS MiB/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**NFS** MiB/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %
seq-read	1th-1f-4k	client	527.23	−28.2	−22.6	−24.4	1070.84	−70.9	−54.7	−54.5
		server	0.59	+306.8	+78.0	+74.6	1.12	+62.5	+2.7	+1.8
	1th-1f-1024k	client	527.19	−28.2	−22.6	−24.4	1070.84	−74.0	−51.5	−51.5
		server	0.61	+291.8	+82.0	+83.6	1.11	+38.7	+16.2	+17.1
	16th-16f-4k	client	535.44	−35.9	−3.0	−2.5	1071.55	−71.8	−5.9	−9.4
		server	0.64	+235.9	+93.8	+95.3	1.13	+55.8	+60.2	+58.4
	16th-16f-1024k	client	535.43	−34.6	−1.5	−1.5	1105.38	−72.5	−11.3	−10.5
		server	0.65	+240.0	+95.4	+93.8	1.16	+52.6	+50.0	+54.3
rand-read	1th-1f-4k	client	18.54	−7.4	−33.1	−33.4	19.03	−22.0	−10.9	−14.4
		server	1.11	+55.9	−46.8	−46.8	1.13	+32.7	−29.2	−31.9
	1th-1f-1024k	client	230.35	−18.4	−13.0	−14.9	372.02	−28.4	+8.4	+11.8
		server	0.24	+283.3	+95.8	+91.7	0.45	+215.6	+131.1	+144.4
	16th-16f-4k	client	190.08	−90.5	−24.2	−24.3	210.24	−93.2	−24.2	−25.3
		server	8.22	−77.6	−39.5	−39.8	8.83	−83.4	−41.3	−42.4
	16th-16f-1024k	client	489.59	−55.2	+2.1	+3.4	1078.19	−75.6	−13.8	−13.2
		server	0.57	+103.5	+131.6	+133.3	1.13	+25.7	+45.1	+44.2
seq-write	1th-1f-4k	client	0.03	+33.3	−33.3	−33.3	0.04	+0.0	−50.0	−50.0
		server	0.56	+3.6	−7.1	−3.6	0.63	−1.6	−9.5	−7.9
	1th-1f-1024k	client	0.08	−50.0	−75.0	−75.0	0.02	+50.0	+0.0	+0.0
		server	62.12	−81.1	−81.8	−80.7	67.06	−78.4	−79.2	−79.3
	16th-16f-4k	client	0.09	−66.7	−22.2	−22.2	0.14	−71.4	−42.9	−42.9
		server	1.96	−73.0	+73.0	+57.1	2.64	−76.5	+68.2	+67.4
	16th-16f-1024k	client	0.08	−50.0	+12.5	+0.0	0.10	−70.0	−20.0	−30.0
		server	101.41	−88.8	−63.2	−65.3	246.89	−93.9	−64.8	−69.8
rand-write	1th-1f-4k	client	0.04	+0.0	−25.0	−25.0	0.04	+0.0	−25.0	−25.0
		server	0.61	+1.6	−3.3	−1.6	0.69	−7.2	−11.6	−13.0
	1th-1f-1024k	client	0.05	−20.0	−60.0	−60.0	0.02	+50.0	+0.0	+0.0
		server	45.79	−75.1	−68.0	−73.4	47.45	−73.8	−72.6	−73.1
	16th-16f-4k	client	0.11	−63.6	−27.3	−27.3	0.15	−73.3	−40.0	−40.0
		server	2.51	−75.7	+66.5	+61.4	2.79	−77.1	+82.1	+79.6
	16th-16f-1024k	client	0.05	−20.0	+80.0	+80.0	0.04	−50.0	+25.0	+25.0
		server	76.61	−85.3	−45.5	−45.6	102.93	−88.6	−42.0	−44.2

CONSUMO DE RECURSOS COMPUTACIONAIS DURANTE A EXECUÇÃO DOS MICRO TESTES ORIENTADOS A DADOS PARA OS CENÁRIOS DE TESTE COM CLIENTE ASSÍNCRONO

Tabela 16: Resultados da utilização de CPU nos micro testes orientados a dados para os cenários de teste com cliente assíncrono.

			SSD				NVMe			
			NFS % CPU	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**NFS** % CPU	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %
seq-read	1th-1f-4k	client	16.38	+34.3	+79.6	+76.3	25.08	−9.4	+18.0	+19.0
		server	4.32	+66.9	+88.9	+94.2	19.00	−23.0	−50.9	−52.3
	1th-1f-1024k	client	14.17	+35.5	+88.8	+87.7	21.34	+0.1	+36.4	+37.6
		server	4.16	+63.9	+106.5	+102.9	19.95	−31.8	−55.3	−54.4
	16th-16f-4k	client	16.44	+32.8	+80.8	+81.8	26.28	+54.0	+145.7	+146.5
		server	4.30	+68.1	+181.4	+192.1	20.23	+22.3	+117.6	+129.8
	16th-16f-1024k	client	13.68	+48.6	+101.0	+98.6	22.64	+60.3	+183.8	+186.6
		server	4.04	+75.2	+185.4	+192.8	18.80	+67.8	+144.6	+153.4
rand-read	1th-1f-4k	client	6.18	+50.0	+59.9	+61.2	7.67	+21.9	+51.6	+51.8
		server	4.90	−25.5	+16.9	+14.1	6.75	−5.3	−54.4	−54.4
	1th-1f-1024k	client	9.98	+25.9	+94.1	+98.4	12.37	+33.9	+112.0	+110.4
		server	3.98	+4.3	+66.1	+65.6	10.54	+25.9	−22.1	−22.6
	16th-16f-4k	client	17.75	−34.0	+268.2	+271.1	21.23	−47.5	+236.7	+239.7
		server	31.06	−72.0	+60.8	+60.7	38.44	−53.3	+2.3	+2.9
	16th-16f-1024k	client	14.76	+29.1	+89.5	+94.5	25.24	+45.7	+158.4	+160.6
		server	4.17	+52.5	+183.2	+187.8	21.34	+27.6	+109.0	+111.0
seq-write	1th-1f-4k	client	1.26	+505.6	+1292.1	+1325.4	2.85	+178.9	+1114.7	+1116.5
		server	5.87	+68.5	−31.3	−32.0	24.27	−64.4	−61.1	−61.9
	1th-1f-1024k	client	1.48	+376.4	+303.4	+280.4	2.06	+263.6	+787.9	+815.5
		server	7.27	+44.7	−37.6	−39.3	22.68	−53.1	−3.5	−3.4
	16th-16f-4k	client	1.03	+2531.1	+2726.2	+2693.2	3.94	+807.9	+1515.0	+1508.4
		server	4.86	+315.6	+151.0	+151.2	32.44	−46.7	+0.9	+1.2
	16th-16f-1024k	client	1.21	+2276.9	+573.6	+547.1	3.65	+825.2	+529.6	+540.5
		server	5.82	+224.7	−13.9	−14.9	32.72	−48.1	−36.2	−35.1
rand-write	1th-1f-4k	client	4.46	+90.8	+387.0	+418.8	6.33	+18.6	+593.0	+602.4
		server	14.81	−36.7	−22.8	−15.5	25.06	−49.1	−7.5	−10.7
	1th-1f-1024k	client	1.99	+280.4	+147.2	+140.7	3.32	+110.8	+305.4	+328.9
		server	4.02	+138.6	−24.4	−23.9	8.76	+47.5	+55.8	+59.4
	16th-16f-4k	client	5.37	+579.1	+303.7	+296.3	8.00	+350.4	+562.6	+567.2
		server	12.03	+72.9	−18.8	−18.6	24.50	−27.1	+20.7	+24.0
	16th-16f-1024k	client	0.83	+3822.9	+581.9	+502.4	2.29	+1380.3	+547.2	+557.2
		server	2.82	+560.3	−17.7	−16.3	8.32	+94.7	+33.9	+35.3

Tabela 17: Resultados da utilização de memória nos micro testes orientados a dados para os cenários de teste com cliente assíncrono.

			SSD				NVMe			
			NFS MiB	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**NFS** MiB	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %
seq-read	1th-1f-4k	client	99.75	−11.4	+2233.6	+2081.4	94.30	−4.6	+1294.8	+1234.3
		server	110.35	+8.5	+10.7	+15.6	148.80	+7.8	+10.8	+8.2
	1th-1f-1024k	client	99.81	−10.1	+2228.3	+2081.0	95.10	−4.0	+1274.4	+1212.9
		server	109.47	+0.7	+11.4	+13.0	142.87	+3.0	+12.7	+9.1
	16th-16f-4k	client	103.65	−10.8	+2032.7	+1892.8	101.79	−8.2	+1246.5	+1213.3
		server	116.21	+7.3	+120.9	+120.8	163.67	+13.3	+54.2	+70.9
	16th-16f-1024k	client	119.26	−8.9	+1715.5	+1571.9	116.94	−6.4	+1080.5	+1044.6
		server	115.58	−0.0	+121.9	+122.1	160.38	+1.6	+51.8	+73.0
rand-read	1th-1f-4k	client	87.32	−1.4	+857.4	+1749.7	83.16	+2.9	+999.0	+1869.9
		server	116.32	+0.9	+99.6	+90.0	165.51	+0.9	+68.4	+68.0
	1th-1f-1024k	client	82.70	+2.9	+921.8	+1656.7	82.95	+1.7	+764.3	+1806.8
		server	115.60	+0.2	+101.0	+91.1	162.38	+0.1	+69.5	+70.9
	16th-16f-4k	client	80.79	+9.2	+1471.7	+1923.8	80.73	+9.1	+908.5	+2207.1
		server	116.68	+0.0	+99.2	+90.5	165.88	+0.1	+67.1	+59.4
	16th-16f-1024k	client	98.46	+8.0	+1078.6	+1322.7	98.20	+7.5	+713.1	+1667.8
		server	113.84	+1.5	+103.1	+95.2	160.46	+1.4	+70.4	+61.7
seq-write	1th-1f-4k	client	129.80	−36.4	+1010.8	+847.9	130.28	−36.5	+354.7	+331.9
		server	123.70	+8.9	+85.6	+91.0	178.57	+6.3	+60.8	+57.3
	1th-1f-1024k	client	130.69	−35.4	+1161.5	+905.9	129.20	−35.0	+615.2	+620.9
		server	124.36	+8.4	+93.3	+99.9	177.40	+6.9	+64.9	+64.4
	16th-16f-4k	client	123.85	−32.0	+1753.8	+1646.1	122.09	−31.3	+1134.3	+1049.8
		server	123.92	+8.9	+112.1	+108.4	172.90	+10.1	+76.2	+73.2
	16th-16f-1024k	client	139.02	−27.8	+1646.5	+1493.3	136.66	−26.8	+1093.1	+1040.7
		server	124.43	+8.6	+113.0	+104.0	175.07	+9.0	+79.9	+75.2
rand-write	1th-1f-4k	client	135.72	−38.1	+1116.0	+1046.7	144.01	−42.1	+455.9	+402.2
		server	138.70	−2.6	+79.5	+87.8	196.12	−2.9	+53.5	+49.2
	1th-1f-1024k	client	116.83	−27.1	+1312.6	+1259.2	112.38	−25.1	+844.1	+777.4
		server	123.85	+9.1	+108.1	+105.9	174.01	+9.2	+66.7	+67.7
	16th-16f-4k	client	249.57	−65.9	+851.8	+795.9	229.53	−62.9	+601.5	+558.2
		server	142.58	−4.8	+82.7	+81.7	198.61	−3.8	+58.3	+50.8
	16th-16f-1024k	client	140.58	−27.9	+1658.2	+1531.5	138.15	−26.5	+1096.6	+1066.5
		server	126.76	+6.8	+106.1	+102.5	181.35	+5.6	+72.6	+68.3

Tabela 18: Resultados da utilização da rede nos micro testes orientados a dados para os cenários de teste com cliente assíncrono.

			SSD				NVMe			
			NFS MiB/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**NFS** MiB/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %
seq-read	1th-1f-4k	client	527.24	−12.5	−7.1	−7.1	1049.84	−56.8	−57.0	−57.4
		server	0.57	+121.1	+100.0	+100.0	1.10	+10.0	−8.2	−9.1
	1th-1f-1024k	client	527.21	−7.1	−4.4	−4.4	1070.83	−55.5	−55.5	−55.5
		server	0.56	+144.6	+112.5	+112.5	1.12	+14.3	−3.6	−3.6
	16th-16f-4k	client	511.48	−0.4	−1.4	−0.4	1049.85	−16.2	−6.7	−8.4
		server	0.61	+101.6	+101.6	+103.3	1.10	+83.6	+60.9	+58.2
	16th-16f-1024k	client	511.46	−0.4	−1.4	+0.1	1070.83	−21.1	−6.7	−7.6
		server	0.61	+101.6	+104.9	+104.9	1.12	+75.0	+65.2	+61.6
rand-read	1th-1f-4k	client	19.80	−19.8	−36.0	−36.5	14.93	+6.4	+12.8	+11.1
		server	1.18	−27.1	−48.3	−48.3	0.90	−4.4	−10.0	−11.1
	1th-1f-1024k	client	230.30	+18.3	+35.4	+36.3	383.29	−6.6	+4.4	+5.3
		server	0.24	+183.3	+258.3	+258.3	0.48	+97.9	+108.3	+108.3
	16th-16f-4k	client	189.04	−73.3	−25.9	−26.1	213.18	−76.8	−27.2	−28.1
		server	8.16	−74.0	−40.2	−40.2	8.86	−76.4	−42.0	−42.1
	16th-16f-1024k	client	490.25	−9.7	+4.5	+5.0	1041.54	−24.2	−7.3	−7.3
		server	0.57	+87.7	+136.8	+140.4	1.07	+71.0	+65.4	+65.4
seq-write	1th-1f-4k	client	0.11	+1500.0	+54.5	+54.5	0.25	+672.0	+24.0	+24.0
		server	120.77	−76.3	+13.6	+16.1	196.46	−84.0	+33.8	+32.1
	1th-1f-1024k	client	0.14	+1200.0	+28.6	+28.6	0.19	+852.6	+142.1	+152.6
		server	148.15	−80.0	+3.9	+3.4	186.21	−84.2	+144.3	+149.8
	16th-16f-4k	client	0.12	+3616.7	+83.3	+83.3	0.27	+1744.4	+100.0	+100.0
		server	122.15	−33.9	+21.2	+20.4	414.32	−78.3	−17.5	−18.5
	16th-16f-1024k	client	0.14	+3035.7	+14.3	+14.3	0.27	+1703.7	+59.3	+59.3
		server	144.81	−45.2	+7.0	+7.4	408.74	−78.7	+12.8	+14.4
rand-write	1th-1f-4k	client	2.81	−27.8	−94.3	−94.0	3.78	−59.5	−91.5	−91.3
		server	69.02	−52.1	+82.0	+92.9	95.41	−73.9	+149.2	+147.4
	1th-1f-1024k	client	0.12	+1491.7	+25.0	+33.3	0.19	+726.3	+84.2	+94.7
		server	102.40	−69.7	+22.4	+27.9	169.37	−85.0	+104.3	+111.8
	16th-16f-4k	client	3.38	+50.9	−95.3	−95.6	4.78	+5.2	−90.2	−90.2
		server	85.26	+8.4	+28.6	+26.9	122.99	−26.1	+131.5	+130.8
	16th-16f-1024k	client	0.07	+6542.9	+71.4	+71.4	0.19	+2489.5	+47.4	+47.4
		server	91.23	−8.7	+10.6	+9.0	208.47	−57.8	+43.5	+43.9

CONSUMO DE RECURSOS COMPUTACIONAIS DURANTE A EXECUÇÃO DOS MICRO TESTES ORIENTADOS A METADADOS PARA OS CENÁRIOS DE TESTE COM CLIENTE SÍNCRONO

Tabela 19: Resultados da utilização de CPU nos micro testes orientados a metadados para os cenários de teste com cliente síncrono.

		SSD								NVMe							
		NFS % CPU	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** % CPU	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
create-1th	client	1.06	+78.3	+240.6	+236.8	+231.1	+248.1	+259.4	+260.4	1.13	+160.2	+258.4	+280.5	+257.5	+271.7	+233.6	+269.0
	server	0.80	+38.8	+71.2	+53.7	+60.0	+55.0	+71.2	+68.8	1.76	+145.5	+153.4	+159.7	+147.2	+147.2	+149.4	+148.3
create-16th	client	0.91	+169.2	+412.1	+357.1	+407.7	+404.4	+428.6	+430.8	1.48	+121.6	+243.9	+252.7	+255.4	+235.8	+257.4	+248.0
	server	0.63	+81.0	+111.1	+123.8	+88.9	+84.1	+109.5	+111.1	2.17	+100.9	+114.7	+119.8	+83.9	+83.9	+79.7	+85.7
delete-1th	client	1.13	+15.9	+132.7	+131.0	+138.1	+137.2	+138.1	+138.1	1.04	+121.2	+158.7	+166.3	+128.8	+137.5	+156.7	+167.3
	server	0.91	+14.3	+39.6	+31.9	+47.3	+68.1	+36.3	+44.0	1.87	+77.5	+78.6	+70.6	+77.0	+71.7	+80.2	+79.1
delete-16th	client	1.49	+49.0	+99.3	+106.7	+109.4	+110.7	+113.4	+104.7	1.66	+81.3	+84.9	+71.1	+79.5	+72.9	+95.2	+88.6
	server	1.02	+1.0	+23.5	+32.4	+24.5	+27.5	+26.5	+23.5	2.03	+68.0	+59.6	+64.0	+64.5	+58.6	+55.7	+66.0
read-1th	client	8.13	−39.9	+2.2	+4.9	+5.7	+5.0	+6.8	+7.0	8.40	−33.0	+26.9	+26.4	+15.5	+18.3	+17.1	+13.5
	server	4.31	−54.1	−43.2	−37.8	−46.4	−42.9	−46.2	−43.6	6.18	+24.3	+40.3	+27.7	+39.5	+37.7	+35.3	+41.6
read-16th	client	40.11	−71.9	−34.0	−40.2	−21.4	−23.5	−30.7	−31.5	41.41	−75.7	−15.8	−15.4	−12.7	−12.9	−12.5	−14.2
	server	33.45	−84.7	−59.6	−62.3	−54.2	−50.7	−59.8	−59.6	39.77	−71.7	−51.9	−52.1	−53.2	−54.5	−53.2	−53.7
stat-1th	client	8.58	+26.1	+25.9	+27.7	+32.6	+30.7	+25.5	+30.5	7.65	+42.9	+29.0	+24.3	+30.5	+34.1	+39.7	+34.4
	server	6.56	+26.4	+26.7	+24.4	+23.9	+26.2	+25.0	+10.8	9.20	+10.1	+7.1	+10.1	+17.8	+6.0	+11.5	+14.6
stat-16th	client	73.45	−84.4	−7.3	−6.2	−2.7	−3.4	−12.7	−17.0	72.93	−85.2	−8.4	−8.0	−9.1	−2.7	−34.2	−25.8
	server	13.91	−38.2	+91.1	+91.9	+79.8	+78.5	−52.4	−49.3	15.66	−28.9	+37.2	+35.8	+27.3	+42.0	−46.4	−43.2

Tabela 20: Resultados da utilização de memória nos micro testes orientados a metadados para os cenários de teste com cliente síncrono.

		SSD								NVMe							
		NFS MiB	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-md-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-md-60s-300MiB** rel. diff. %	**NFS** MiB	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-md-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-md-60s-300MiB** rel. diff. %
create-1th	client	93.42	+12.5	+9.8	+65.5	+8.1	+8.9	+6.3	+6.0	84.55	+14.9	+75.9	+77.7	+14.4	+17.9	+11.2	+21.2
	server	116.74	−14.4	−7.2	−1.2	−15.4	−6.0	−13.7	−5.1	182.14	−8.2	+6.6	+5.8	−3.7	−3.7	−3.7	−3.9
create-16th	client	94.94	+10.3	+11.6	+64.5	+10.3	+10.7	+8.4	+8.0	86.20	+13.8	+75.2	+77.0	+16.7	+20.6	+13.9	+24.1
	server	119.77	−16.1	−8.6	−3.4	−16.1	−8.8	−14.3	−7.1	183.30	−8.6	+6.1	+5.5	−4.0	−3.9	−4.1	−4.1
delete-1th	client	97.83	−8.1	+8.2	+9.3	+9.7	+9.2	+9.4	+9.4	85.88	+6.4	+794.9	+69.0	+19.9	+18.2	+10.9	+17.0
	server	117.64	−13.6	−4.9	−4.9	−4.6	−4.5	−4.8	−4.9	181.44	−8.0	+3178.5	+3453.8	−14.9	−14.4	−2.4	−2.4
delete-16th	client	98.60	−5.5	+8.2	+9.1	+9.6	+9.2	+9.0	+9.9	87.20	+5.7	+783.4	+68.9	+860.2	+1924.6	+10.7	+16.4
	server	117.60	−13.6	−4.8	−4.6	−4.1	−4.2	−4.7	−4.7	182.08	−8.5	+3166.9	+3262.7	+3722.7	+3807.7	−2.8	−2.6
read-1th	client	127.15	+21.3	+2289.7	+2002.2	+3004.1	+2947.0	+2916.2	+2958.4	122.29	+23.1	+2555.3	+2862.4	+3689.6	+3708.5	+3704.0	+3609.3
	server	118.21	−8.4	+3563.2	+3703.6	+3494.2	+3614.1	+3450.0	+3437.3	188.97	+1.5	+2585.3	+2565.0	+2663.3	+2632.2	+2570.8	+2633.6
read-16th	client	128.18	+38.3	+672.4	+1430.4	+3092.2	+2952.0	+3078.7	+3083.6	122.85	+39.9	+441.1	+1786.3	+3727.6	+3680.8	+3383.6	+3895.6
	server	119.14	−3.5	+4785.7	+5211.8	+5045.3	+5010.2	+5001.2	+4871.0	189.71	+4.2	+3868.5	+3874.8	+4046.8	+3971.2	+3786.9	+3958.9
stat-1th	client	92.90	+107.8	+3397.4	+3507.8	+5276.5	+5010.9	+4528.5	+4463.4	86.08	+120.9	+4133.9	+3589.7	+5403.1	+5540.8	+5223.9	+5054.8
	server	113.97	−14.0	+5347.5	+5416.8	+5287.9	+5421.8	+3753.3	+3484.0	182.21	−8.7	+3373.1	+3645.8	+3743.6	+3926.7	+2264.8	+2405.4
stat-16th	client	93.72	+106.2	+1167.9	+1216.9	+5594.4	+5286.9	+5532.2	+5310.1	88.52	+115.5	+1636.8	+1454.8	+5877.6	+5437.4	+5938.1	+5490.4
	server	117.20	−16.2	+5214.2	+5269.6	+4985.2	+5188.2	+5289.9	+5313.2	184.89	−10.2	+3564.0	+3799.3	+3773.8	+3811.7	+3530.7	+3797.4

Tabela 21: Resultados da utilização da rede nos micro testes orientados a metadados para os cenários de teste com cliente síncrono.

		SSD								NVMe							
		NFS MiB/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** MiB/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
create-1th	client	0.05	+200.0	+60.0	+40.0	+40.0	+40.0	+60.0	+60.0	0.06	+116.7	+33.3	+33.3	+33.3	+33.3	+33.3	+33.3
	server	0.33	+33.3	+9.1	−9.1	+0.0	+0.0	+6.1	+6.1	0.34	+20.6	+2.9	+0.0	+0.0	+0.0	+2.9	+2.9
create-16th	client	0.04	+275.0	+75.0	+75.0	+75.0	+75.0	+75.0	+75.0	0.06	+150.0	+33.3	+33.3	+16.7	+16.7	+16.7	+16.7
	server	0.25	+80.0	+60.0	+36.0	+44.0	+48.0	+60.0	+60.0	0.39	+7.7	+5.1	+2.6	+2.6	+2.6	+2.6	+2.6
delete-1th	client	0.07	+0.0	−42.9	−42.9	−42.9	−42.9	−42.9	−42.9	0.07	+0.0	−42.9	−42.9	−42.9	−42.9	−42.9	−42.9
	server	0.06	+33.3	−16.7	−16.7	−16.7	−16.7	−16.7	−33.3	0.06	+50.0	−16.7	−16.7	−16.7	−16.7	−16.7	−16.7
delete-16th	client	0.07	+0.0	−42.9	−42.9	−42.9	−42.9	−42.9	−42.9	0.08	+0.0	−50.0	−50.0	−50.0	−50.0	−50.0	−50.0
	server	0.06	+16.7	−16.7	−16.7	−16.7	−16.7	−16.7	−16.7	0.07	+28.6	−28.6	−28.6	−28.6	−28.6	−28.6	−28.6
read-1th	client	9.68	−76.5	−80.7	−81.4	−80.4	−80.3	−80.9	−80.9	9.49	−74.2	−75.1	−76.1	−77.2	−77.0	−77.1	−77.2
	server	1.45	−55.2	−71.7	−72.4	−71.7	−71.7	−72.4	−72.4	1.42	−50.0	−64.1	−65.5	−66.9	−66.9	−66.9	−66.9
read-16th	client	85.53	−91.3	−82.0	−84.8	−80.1	−80.3	−82.5	−82.4	89.38	−94.0	−77.9	−78.3	−79.0	−78.7	−78.5	−78.6
	server	12.18	−82.7	−76.5	−80.5	−73.9	−74.1	−77.0	−77.0	12.74	−87.9	−70.9	−71.4	−72.2	−72.0	−71.6	−71.8
stat-1th	client	1.67	+38.3	−37.1	−37.7	−37.7	−40.7	−39.5	−41.9	1.55	+13.5	−40.0	−47.7	−50.3	−40.6	−43.9	−46.5
	server	1.81	+16.0	−41.4	−41.4	−41.4	−44.2	−43.1	−45.3	1.72	−2.3	−45.3	−52.3	−54.7	−45.9	−49.4	−51.7
stat-16th	client	6.14	−59.1	+9.0	+9.6	−2.9	−2.6	−80.0	−80.0	6.23	−66.3	+8.7	+7.4	−10.1	−2.4	−81.9	−81.2
	server	6.47	−63.7	−1.7	−1.5	−12.1	−11.9	−80.2	−80.1	6.48	−69.0	−3.1	−4.3	−19.3	−12.2	−82.9	−82.1

CONSUMO DE RECURSOS COMPUTACIONAIS DURANTE A EXECUÇÃO DOS MICRO TESTES ORIENTADOS A METADADOS PARA OS CENÁRIOS DE TESTE COM CLIENTE ASSÍNCRONO

Tabela 22: Resultados da utilização de CPU nos micro testes orientados a metadados para os cenários de teste com cliente assíncrono.

		SSD								NVMe							
		NFS % CPU	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** % CPU	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
create-1th	client	1.02	+141.2	+244.1	+293.1	+331.4	+347.1	+338.2	+352.9	1.04	+174.0	+358.7	+319.2	+356.7	+315.4	+271.2	+303.8
	server	0.78	−5.1	+65.4	+53.8	+57.7	+66.7	+66.7	+44.9	1.80	+54.4	+120.6	+117.8	+103.3	+120.0	+120.6	+120.0
create-16th	client	1.28	+136.7	+255.5	+264.8	+269.5	+255.5	+314.8	+321.1	1.46	+127.4	+252.7	+255.5	+272.6	+282.2	+267.8	+270.5
	server	0.86	+26.7	+48.8	+48.8	+39.5	+39.5	+34.9	+47.7	2.25	+29.8	+105.8	+102.2	+71.1	+70.7	+67.1	+73.3
delete-1th	client	0.80	+147.5	+177.5	+176.2	+202.5	+192.5	+187.5	+178.8	1.07	+159.8	+157.0	+155.1	+134.6	+150.5	+168.2	+159.8
	server	0.79	−12.7	+44.3	+35.4	+48.1	+46.8	+39.2	+43.0	1.99	+49.7	+74.4	+67.8	+61.8	+68.3	+73.4	+56.8
delete-16th	client	1.27	+141.7	+92.9	+101.6	+121.3	+118.1	+117.3	+114.2	1.53	+115.0	+102.6	+99.3	+102.6	+102.6	+103.3	+126.1
	server	0.81	+3.7	+37.0	+46.9	+40.7	+42.0	+46.9	+48.1	2.09	+73.7	+54.5	+58.9	+56.9	+60.3	+54.1	+58.4
read-1th	client	7.85	−25.1	+22.5	+15.0	+17.2	+17.2	+17.1	+15.2	8.12	−15.1	+39.9	+21.4	+29.3	+26.2	+22.0	+24.3
	server	4.42	−59.5	−44.8	−41.9	−46.6	−45.9	−45.0	−45.0	6.45	−23.7	+40.5	+31.2	+36.4	+35.7	+34.9	+33.2
read-16th	client	39.97	−73.2	−20.8	−21.5	−18.0	−19.2	−18.4	−19.3	41.94	−75.3	−13.5	−12.4	−9.7	−13.2	−10.1	−11.0
	server	33.19	−73.1	−51.9	−54.9	−54.4	−54.7	−54.4	−55.0	42.09	−55.3	−53.1	−55.3	−55.1	−52.9	−55.3	−55.6
stat-1th	client	8.38	−17.7	+27.7	+29.2	+30.2	+27.3	+29.1	+32.9	7.65	−13.6	+30.1	+38.0	+39.6	+38.0	+41.2	+38.8
	server	6.54	+11.3	+25.7	+23.1	+23.5	+31.7	+21.3	+21.9	8.57	+15.3	+15.9	+15.5	+19.0	+17.2	+20.5	+14.6
stat-16th	client	73.05	−78.2	−9.5	−7.7	−2.5	−3.8	−10.6	−15.0	72.44	−78.7	−7.7	−7.1	−14.7	−15.3	−34.9	−41.1
	server	13.75	+3.8	+81.3	+83.9	+87.3	+86.2	−47.3	−48.4	15.59	−2.7	+26.6	+25.7	+11.0	+14.1	−44.3	−44.2

Tabela 23: Resultados da utilização de memória nos micro testes orientados a metadados para os cenários de teste com cliente assíncrono.

		SSD								NVMe							
		NFS MiB	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** MiB	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
create-1th	client	93.70	+9.6	+51.8	+51.6	+5.1	+5.8	+1.6	+5.3	84.47	+30.6	+59.4	+60.1	+26.4	+12.1	+11.5	+11.1
	server	111.91	+1.0	−1.6	−1.5	−11.7	−11.4	−10.8	−10.9	182.11	−0.2	+2.3	+1.7	−3.7	−3.7	−3.6	−3.6
create-16th	client	94.18	+9.7	+51.8	+52.8	+8.3	+9.0	+5.2	+8.9	85.50	+30.4	+60.1	+60.9	+29.6	+16.7	+15.3	+15.1
	server	112.76	+0.3	−1.8	−1.9	−11.6	−10.8	−10.1	−10.4	182.97	−0.6	+2.0	+1.5	−4.0	−3.8	−3.9	−4.1
delete-1th	client	93.85	+22.2	+550.9	+511.1	+1891.6	+2457.9	+9.1	+3.5	95.88	+10.5	+188.4	+448.5	+9.4	+9.2	−1.3	−1.5
	server	110.67	+0.1	+5747.3	+5463.7	+5730.6	+5739.3	−0.1	−0.3	191.34	−6.0	+3371.9	+3393.9	−3.2	−3.1	−8.4	−8.4
delete-16th	client	95.26	+21.6	+541.7	+502.8	+1345.8	+2328.3	+6.3	+5.6	96.78	+10.5	+186.7	+403.9	+11.0	+10.5	+0.1	+0.8
	server	110.49	+0.5	+5559.8	+5390.3	+5529.0	+5526.0	+0.0	+0.2	191.58	−5.9	+3367.5	+3389.7	−3.1	−3.2	−8.2	−8.3
read-1th	client	124.46	+19.0	+2102.1	+2298.1	+3172.4	+3124.6	+3226.6	+3145.0	117.32	+22.5	+2807.6	+3027.4	+3900.0	+3888.5	+3890.7	+3880.8
	server	105.74	+11.1	+3994.3	+3875.9	+3931.5	+3846.4	+3725.9	+3941.3	189.62	+6.1	+2406.7	+2539.8	+2818.8	+2668.3	+2757.9	+2677.6
read-16th	client	125.71	+46.9	+947.7	+1429.9	+3580.0	+3282.5	+3333.7	+3229.5	117.93	+52.7	+1287.7	+1945.9	+4336.7	+4457.3	+4569.6	+4208.7
	server	107.71	+18.7	+5496.2	+5324.3	+5439.7	+5616.3	+5394.2	+5428.9	190.77	+10.3	+3604.0	+3893.2	+4052.1	+4005.0	+4076.7	+3937.1
stat-1th	client	95.26	+121.7	+3651.0	+3542.4	+4659.3	+4955.4	+4460.8	+4389.0	86.61	+131.6	+3833.1	+4213.7	+5310.8	+5311.0	+5157.2	+4892.2
	server	111.65	−1.7	+5467.8	+5505.5	+5426.3	+5329.7	+3838.0	+3834.8	182.12	−2.0	+3556.1	+3616.5	+3567.2	+3914.0	+2567.7	+2449.0
stat-16th	client	95.63	+122.4	+1483.7	+1431.9	+5105.5	+4882.9	+5387.6	+5474.4	88.23	+128.9	+1566.6	+1908.7	+5740.6	+5525.5	+6119.3	+5904.1
	server	114.47	−4.3	+5434.2	+5411.2	+5281.7	+5332.1	+5421.5	+5353.3	197.35	−9.8	+3403.7	+3301.2	+3061.8	+3523.2	+3535.1	+3551.8

Tabela 24: Resultados da utilização da rede nos micro testes orientados a metadados para os cenários de teste com cliente assíncrono.

		SSD								NVMe							
		NFS MiB/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** MiB/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-mc-lru-10s-100MiB** rel. diff. %	**RSafeFS-mc-rnd-10s-100MiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
create-1th	client	0.05	+400.0	+20.0	+20.0	+20.0	+20.0	+20.0	+20.0	0.06	+333.3	+16.7	+0.0	+16.7	+16.7	+0.0	+16.7
	server	0.33	+21.2	−12.1	−12.1	−6.1	−6.1	+0.0	+0.0	0.35	+17.1	−5.7	−5.7	−5.7	−5.7	−5.7	−5.7
create-16th	client	0.06	+350.0	+0.0	+16.7	+0.0	+0.0	+16.7	+16.7	0.06	+366.7	+33.3	+16.7	+16.7	+16.7	+16.7	+16.7
	server	0.36	+16.7	−8.3	−5.6	−2.8	−2.8	+5.6	+5.6	0.39	+12.8	+0.0	+0.0	+0.0	+0.0	+0.0	+0.0
delete-1th	client	0.07	+314.3	−57.1	−57.1	−57.1	−57.1	−57.1	−57.1	0.07	+371.4	−42.9	−42.9	−42.9	−42.9	−42.9	−42.9
	server	0.06	+233.3	−33.3	−33.3	−33.3	−33.3	−33.3	−33.3	0.06	+266.7	−16.7	−16.7	−16.7	−16.7	−16.7	−16.7
delete-16th	client	0.06	+466.7	−50.0	−50.0	−50.0	−50.0	−50.0	−50.0	0.08	+387.5	−50.0	−50.0	−50.0	−50.0	−50.0	−50.0
	server	0.05	+360.0	−20.0	−20.0	−20.0	−20.0	−20.0	−20.0	0.06	+333.3	−16.7	−16.7	−16.7	−16.7	−16.7	−16.7
read-1th	client	9.43	−81.3	−80.0	−79.7	−80.0	−79.9	−79.9	−79.9	9.45	−78.8	−77.7	−76.9	−77.7	−77.5	−77.7	−76.9
	server	1.41	−50.4	−70.9	−70.2	−70.9	−70.9	−70.9	−70.9	1.42	−43.7	−67.6	−66.9	−67.6	−67.6	−67.6	−66.9
read-16th	client	85.59	−86.7	−79.9	−80.4	−80.7	−80.7	−80.7	−80.6	90.16	−88.4	−78.8	−78.8	−79.6	−80.1	−79.4	−79.2
	server	12.18	−68.3	−73.5	−74.3	−74.5	−74.5	−74.5	−74.5	12.83	−72.1	−71.7	−71.9	−72.8	−73.4	−72.6	−72.4
stat-1th	client	1.65	+67.3	−35.8	−36.4	−38.2	−42.4	−38.8	−38.2	1.56	+59.0	−37.8	−38.5	−41.7	−42.3	−42.3	−42.3
	server	1.81	+0.6	−40.3	−40.9	−42.5	−46.4	−43.1	−42.5	1.72	−4.7	−43.6	−43.6	−46.5	−47.1	−47.1	−47.7
stat-16th	client	6.10	+2.5	+0.0	+2.0	−10.0	−10.0	−79.5	−80.0	6.25	−3.0	+2.7	+1.9	−24.6	−22.1	−82.4	−84.2
	server	6.41	−42.4	−7.8	−6.2	−16.4	−16.8	−79.6	−79.6	6.46	−44.6	−6.0	−7.0	−30.7	−28.2	−83.0	−84.5

CONSUMO DE RECURSOS COMPUTACIONAIS DURANTE A EXECUÇÃO DOS MACRO TESTES PARA OS CENÁRIOS DE TESTE COM CLIENTE SÍNCRONO

Tabela 25: Resultados da utilização de CPU nos macro testes para os cenários de teste com cliente síncrono.

		SSD								NVMe							
		NFS % CPU	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** % CPU	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
file-server	client	1.42	+193.0	+298.6	+298.6	+385.2	+393.7	+368.3	+385.9	1.98	+127.8	+342.4	+266.2	+322.2	+380.3	+244.9	+244.4
	server	1.46	−3.4	+10.3	+6.8	+23.3	+20.5	+19.2	+25.3	6.25	−18.4	+6.6	−11.0	+1.4	+8.6	−25.4	−25.6
mail-server	client	1.37	+181.8	+211.7	+211.7	+248.2	+246.0	+253.3	+251.8	1.57	+184.1	+317.8	+311.5	+283.4	+323.6	+300.0	+311.5
	server	0.98	+36.7	+63.3	+60.2	+67.3	+66.3	+67.3	+69.4	3.54	+46.9	+61.9	+61.9	+62.7	+80.5	+44.4	+55.9
web-server	client	6.01	+123.3	+241.6	+237.8	+264.2	+274.7	+301.2	+298.7	6.26	+82.1	+204.0	+203.4	+232.1	+239.0	+257.3	+248.4
	server	1.31	+338.9	+390.8	+371.8	+426.7	+418.3	+434.4	+429.0	4.70	+158.9	+351.3	+341.3	+334.5	+336.0	+463.8	+448.5

Tabela 26: Resultados da utilização de memória nos macro testes para os cenários de teste com cliente síncrono.

		SSD								NVMe							
		NFS MiB	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** MiB	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
file-server	client	600.04	+0.6	+9.2	+9.1	+210.8	+206.0	+12.2	+11.8	597.62	+0.8	+9.0	+8.6	+222.0	+216.4	+11.6	+11.0
	server	127.97	−22.4	−4.2	−13.7	−4.0	−6.3	+2.1	+1.9	193.80	−13.2	−1.6	−1.4	−1.3	−0.1	+2.7	+3.7
mail-server	client	256.96	+1.7	+25.2	+24.3	+806.5	+800.9	+32.7	+31.3	251.34	+3.4	+24.9	+24.9	+879.7	+844.6	+32.7	+31.1
	server	121.38	−16.9	+3.6	−6.1	+3.5	+1.4	+10.1	+9.0	186.29	−9.3	+4.1	+2.6	+2.9	+4.5	+8.9	+9.3
web-server	client	1108.66	+0.7	+6.4	+6.2	+136.2	+134.2	+8.2	+7.9	1103.60	+1.0	+6.1	+6.2	+138.2	+135.1	+8.1	+7.1
	server	113.87	−8.5	+13.0	+4.2	+17.3	+15.1	+20.9	+21.3	181.55	−5.9	+9.1	+8.3	+10.2	+11.7	−2.7	−5.3

Tabela 27: Resultados da utilização da rede nos macro testes para os cenários de teste com cliente síncrono.

		SSD								NVMe							
		NFS MiB/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** MiB/s	**FUSE+Capnp** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
file-server	client	3.89	+2.6	−2.8	−6.7	+112.9	+119.3	+13.9	+20.1	4.60	−19.6	+44.3	+1.7	+125.2	+176.1	−8.0	−5.9
	server	10.64	−61.0	−63.1	−64.5	−57.5	−56.2	−56.8	−54.5	14.01	−72.4	−50.8	−65.2	−59.5	−50.4	−68.5	−67.8
mail-server	client	0.84	+46.4	+47.6	+40.5	+898.8	+890.5	+53.6	+56.0	0.86	+40.7	+80.2	+60.5	+983.7	+1103.5	+55.8	+54.7
	server	1.00	−29.0	−36.0	−38.0	−31.0	−31.0	−31.0	−31.0	1.24	−42.7	−31.5	−39.5	−37.9	−30.6	−41.9	−41.1
web-server	client	1.68	+1091.1	+1019.0	+992.9	+6723.2	+6719.6	+1161.3	+1164.9	1.73	+734.7	+1018.5	+997.1	+6449.1	+6454.9	+1292.5	+1275.7
	server	2.03	+79.3	+13.8	+11.3	+10.8	+10.3	+28.1	+28.1	2.32	+12.9	+9.1	+6.5	+1.7	+2.2	+35.3	+34.1

CONSUMO DE RECURSOS COMPUTACIONAIS DURANTE A EXECUÇÃO DOS MACRO TESTES PARA OS CENÁRIOS DE TESTE COM CLIENTE ASSÍNCRONO

Tabela 28: Resultados da utilização de CPU nos macro testes para os cenários de teste com cliente assíncrono.

		SSD								NVMe							
		NFS % CPU	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** % CPU	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
file-server	client	1.44	+445.8	+275.7	+274.3	+369.4	+362.5	+302.1	+336.1	1.81	+344.8	+289.0	+286.2	+319.3	+320.4	+317.1	+302.8
	server	1.42	+92.3	+12.7	+8.5	+22.5	+25.4	+9.2	+0.0	5.61	+96.8	−1.4	−3.4	+1.6	+3.4	−13.0	−0.9
mail-server	client	1.37	+138.0	+207.3	+210.9	+258.4	+292.7	+224.8	+245.3	1.57	+149.0	+286.0	+281.5	+285.4	+291.7	+296.2	+309.6
	server	0.93	+51.6	+54.8	+53.8	+69.9	+83.9	+55.9	+66.7	3.40	+54.4	+62.4	+56.8	+66.5	+67.9	+66.2	+74.7
web-server	client	63.28	−80.4	−12.1	−12.3	−21.1	−15.9	−7.1	−5.2	68.82	−83.5	−4.2	−4.1	−13.9	−11.1	−3.0	−3.0
	server	22.89	−53.7	−7.8	−8.0	−18.0	−16.6	−6.8	−11.5	17.91	+78.0	+21.3	+22.1	+6.0	+7.2	+21.8	+23.2

Tabela 29: Resultados da utilização de memória nos macro testes para os cenários de teste com cliente assíncrono.

		SSD								NVMe							
		NFS MiB	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** MiB	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
file-server	client	600.86	+2.4	+8.4	+8.4	+239.7	+225.4	+11.9	+11.1	593.80	+3.6	+9.1	+9.3	+233.4	+227.6	+14.1	+12.5
	server	114.98	+10.9	−8.2	−7.7	+0.0	+0.1	+10.0	+9.5	188.50	+8.8	−1.8	−2.2	−1.7	−1.2	+3.1	−13.8
mail-server	client	258.97	+3.0	+15.2	+15.2	+811.9	+758.1	+31.2	+31.0	250.09	+7.2	+17.3	+18.7	+825.8	+812.5	+38.5	+37.9
	server	113.93	+2.5	−5.4	−5.7	+0.2	+3.1	+12.6	+13.3	185.71	−0.0	+0.2	−1.1	−0.1	+1.5	+7.8	+5.6
web-server	client	1115.80	+0.7	+4.4	+4.4	+142.5	+131.2	+7.5	+7.2	1106.71	+2.0	+4.7	+4.8	+136.2	+135.2	+8.8	+8.5
	server	107.43	+12.8	+4.6	+4.1	+13.0	+14.2	+23.5	+24.2	179.39	+5.0	+6.0	+5.7	+6.1	+6.9	+14.1	+12.3

Tabela 30: Resultados da utilização da rede nos macro testes para os cenários de teste com cliente assíncrono.

		SSD								NVMe							
		NFS MiB/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %	**NFS** MiB/s	**FUSE-NFS** rel. diff. %	**FUSE+gRPC** rel. diff. %	**RSafeFS** rel. diff. %	**RSafeFS-dc-lru-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-dc-rnd-30s-1GiB-128KiB** rel. diff. %	**RSafeFS-mc-lru-60s-300MiB** rel. diff. %	**RSafeFS-mc-rnd-60s-300MiB** rel. diff. %
file-server	client	3.82	+48.4	−5.5	−6.0	+121.2	+118.6	+0.8	+9.2	4.13	+38.3	+13.1	+11.6	+110.9	+113.8	+11.6	+11.4
	server	10.34	−48.5	−63.6	−63.8	−55.3	−55.9	−61.3	−57.9	11.87	−54.8	−59.1	−59.6	−59.8	−59.2	−59.4	−59.6
mail-server	client	0.84	+70.2	+40.5	+39.3	+889.3	+994.0	+42.9	+54.8	0.86	+83.7	+60.5	+61.6	+946.5	+945.3	+59.3	+80.2
	server	0.99	−14.1	−37.4	−36.4	−31.3	−23.2	−35.4	−32.3	1.19	−21.8	−37.0	−36.1	−37.0	−37.0	−37.0	−29.4
web-server	client	9.60	+147.5	+543.8	+539.5	+2970.6	+3081.2	+540.5	+552.6	10.40	+80.4	+606.7	+599.0	+3246.3	+3256.0	+575.8	+576.0
	server	47.60	−88.3	−83.7	−83.8	−87.2	−86.7	−83.8	−83.6	51.65	−91.3	−82.1	−82.3	−86.0	−86.0	−83.0	−83.0