
**António Miguel Carneirinho Guiomar**

Licenciatura em Engenharia Informática

# T-Stratus - Confiabilidade e Privacidade com Nuvens de Armazenamento de Dados

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Prof. Doutor Henrique João Lopes Domingos, Prof. Auxiliar, Departamento de Informática, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri:

Presidente: Prof. Doutor Fernando Pedro Reino da Silva Birra

Arguente: Prof. Doutora Maria Dulce Pedroso Domingos

Vogal: Prof. Doutor Henrique João Lopes Domingos


FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

**Maio, 2013**

**T-Stratus - Confiabilidade e Privacidade com Nuvens de Armazenamento de Dados**

# Resumo

O objetivo da dissertação visa conceber, implementar e avaliar um sistema para acesso, gestão e pesquisa confiável de dados privados mantidos em nuvens de armazenamento de provedores Internet (ou *Internet Cloud-Storage Solutions*). O sistema (que designaremos por T-Stratus), preserva garantias de autenticidade, privacidade, fiabilidade, disponibilidade dos dados e tolerância a intrusões, sob controlo dos utilizadores e de forma independente dos provedores.

O sistema T-Stratus atua como base de confiança para indexação, encriptação e distribuição de dados em múltiplas nuvens de armazenamento, permitindo o acesso aos mesmos para leitura, escrita e pesquisa em tempo real. O armazenamento distribuído dos dados faz-se por um processo de fragmentação e replicação, sendo os fragmentos cifrados e replicados em múltiplas nuvens de armazenamento.

A arquitetura do sistema T-Stratus, concebida como sistema *middleware*, permite o seu uso como serviço intermediário entre utilizadores individuais e múltiplas nuvens de armazenamento, numa solução de nuvem de nuvens. A solução pretende tirar partido das vantagens da manutenção e replicação de dados privados em múltiplos provedores Internet, com controlo da base de confiança que assegura as propriedades de segurança e confiabilidade. A avaliação experimental do sistema proposto envolve a obtenção de métricas de latência e de taxas de transferência de dados em escrita e leitura, comparando a eficiência da solução com a utilização, considerada não segura e não confiável, das atuais soluções de armazenamento em nuvens na Internet.

**Palavras-chave:** Nuvens de armazenamento de dados; Segurança; Confiabilidade; Privacidade; Gestão e pesquisa segura de dados Privados.

# Abstract

In this thesis, the goal is to design, implement and evaluate a system that is responsible for the management, access and search (in a reliable and secure way) of private data held by some subset of today's available cloud storage providers. This system (which we call T-Stratus) will be able to guarantee several security properties over the stored data, such as authenticity, privacy, reliability, fault and intrusion tolerance and data availability, in a completely independent way, under full control from the users point of view and completely independent from the adopted cloud storage providers.

This system acts like a thrust computer base in order to allow users to index, encrypt and distribute all data through multiple cloud storage providers, providing them with read, write and real time search over the encrypted data. This storage consists in a distribution of the data by a fragmentation and replication process, with the encryption and replication of the fragments in multiple storage clouds.

T-Stratus architecture, designed as a middleware system, can be used as an intermediate/middleman between the user application and the public data storage clouds, in a cloud of clouds solution. This solution's key idea is to take advantage of the data maintenance and replication, provided by the available cloud-storage solutions, while maintaining the thrust base that allow users to keep control over it's security properties and reliability. The system will be evaluated through benchmarking involving latency and throughput metrics in read and write operations, by comparing the efficiency of the presented reliable and secure solution, with the unreliable and insecure use of the available cloud storage solutions.

**Keywords:** Data-Storage Clouds; Security; Dependability; Privacy; Data Management and Secure Data Searching.

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Listagens

# Glossário

**ACL** *Representational State Transfer*, lista que define quem tem permissão de acesso a certos serviços. 13

**AES** *Advanced Encryption Standard*. 36, 74, 75, 92, 93, 108

**API** *Application Programming Interface* é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por aplicativos que não pretendem envolver-se em detalhes da implementação do software, mas apenas usar os seus serviços. 8, 12, 14, 44, 46, 47, 51, 52, 58, 59, 68, 70, 72, 73, 79, 83, 111, 112, 114, 115

**appliance** dispositivo hardware com software integrado, especificamente desenhado para um propósito bem definido. 50

**CBC** *Cipher-Block Chaining*, modo de cifra de blocos, com recurso a um vetor de inicialização. 36, 76

**checksum** código usado para verificar a integridade de dados transmitidos. 11

**ciphertext** resultado de uma encriptação, através de um algoritmo criptográfico. 30, 36, 50, 74, 76

**cloud** conceito de computação em nuvem, que consiste no uso de recursos (software e hardware) fornecidos como serviços na Internet. 1

**cluster** conjunto de computadores, que utiliza um tipo especial de sistema operativo classificado como sistema distribuído. 10, 11, 24, 44, 47, 60, 62–64, 86, 115

**DES** *Data Encryption Standard*, anterior standard, agora considerado inseguro, devido ao tamanho da chave ser bastante pequeno. 92

**digest** nome dado por vezes ao resultado de uma função de *hash*. 25, 34, 53, 76, 77, 81, 91, 92, 94, 108

**hash** sequência de bits geradas por um algoritmo de dispersão, geralmente representada em base hexadecimal. 10, 19, 22, 34, 63, 70, 74, 76, 91

**hash table** estrutura de dados, denominada tabela de dispersão, que associa chaves de pesquisa a valores. 11

**HMAC** *Hash-based Message Authentication Code*, combinação de um algoritmo de *hash* criptográfico com uma chave secreta, de forma a calcular o o código de autenticação de mensagem (MAC). 21, 32, 74

**ISP** *Internet Service Provider* ou fornecedor de acesso à Internet. 3

**MAC** *Message Authentication Code*, pequeno pedaço de informação usado para autenticar uma mensagem, bem como assegurar a sua integridade. 25, 35

**middleware** programa de computador que faz a mediação entre software e demais aplicações. v, 5, 6, 8, 32, 38–40, 43–47, 49, 51, 52, 60, 62, 65, 67, 68, 70, 73, 77, 83, 85–87, 93, 95–104, 108, 109, 111–113, 115

**multicast** difusão de uma mensagem ou informação por um grupo de computadores destino, numa única transmissão origem. 19

**nonce** número arbitrário utilizado apenas uma vez durante uma comunicação criptográfica. 23

**NoSQL** não implementa o modelo dos sistemas de gestão de bases de dados relacionais, não usando o SQL como linguagem para a manipulação dos mesmos. 11, 86

**phishing** fraude eletrónica, caracterizada por tentativas de adquirir dados pessoais de diversos tipos. 29

**plaintext** dados em claro, que não foram alvo de um algoritmo de cifra. 30, 36, 50, 74–76, 103

**proxy** servidor que age como intermediário a pedidos de clientes. 5, 38, 43–45, 47, 49, 60, 64, 83, 113

**RAID** *Redundant Array of Independent Disks*, sub-sistema de armazenamento composto por vários discos individuais, com a finalidade de ganhar segurança e desempenho. 8, 26–28, 115

**replay** ataque criptográfico onde uma transmissão de dados válida é maliciosamente ou fraudulentamente repetida. 23

**REST** *Representational State Transfer*, tipo de arquitetura de software para sistemas distribuídos como a *World Wide Web*. 12, 15, 83, 84

**RSA** *Ron Rivest, Adi Shamir and Leonard Adleman*, algoritmo de criptografia assimétrica. 31

**SQL** *Structured Query Language*. 35

**SSL** *Secure Sockets Layer*, protocolo criptográfico que assegura comunicação segura através da Internet. 7, 13, 44, 64

**stratus** nuvens muito baixas (0 a 1000m) de aspecto estratificado que cobrem largas faixas horizontais do céu. 5

**timestamp** marca temporal, conjunto de caracteres que denotam a hora ou data de ocorrência de um evento. 11, 14, 21–23

**TLS** *Transport Layer Security*, protocolo criptográfico que assegura comunicação segura através da Internet. 7, 44, 64

**Triple DES** *Triple Data Encryption Algorithm*, aplica o algoritmo DES três vezes sobre cada bloco de dados, são apenas conhecidos ataques teóricos. 92, 93

**URL** *Uniform Resource Locator*, corresponde ao endereço de um recurso disponível numa rede. 13, 52

**vetor de inicialização** conjunto de bits, de forma a atribuir um carácter aleatório a uma encriptação, permitindo a geração de diferentes ciphertext para iguais plaintext. 36, 68, 76

**web service** método de comunicação entre dois dispositivos eletrónicos através da *World Wide Web*. 15, 47, 49, 68, 83

# 1

# Introdução

## 1.1 Contexto e Motivação

As soluções de gestão de dados em nuvens de computação e armazenamento de dados na Internet (ou *Internet Cloud Computing and Storage Solutions*) são bastante utilizadas, tal como são disponibilizadas hoje por diferentes fornecedores e em diferentes soluções. Tanto podem ser usadas como repositórios de dados (como a Amazon S3 [Ama]), ou como infraestruturas para suporte e execução de aplicações (como por exemplo a Amazon EC2[1]).

Estas soluções apresentam características interessantes e apelativas [Cho10; CS11; Aba09] quer para utilizadores individuais, quer para empresas ou organizações, com vantagens importantes: baixo custo de armazenamento, escalabilidade e elasticidade face às necessidades, modelo de "custo por uso"(referido como *pay as you go* ou *pay-per-use*); disponibilidade permanente; acesso ubíquo e transparência face às aplicações que podem ser suportadas em diferentes dispositivos computacionais. De um modo geral, a adoção de soluções de computação em nuvem apresentam garantias interessantes de fiabilidade e segurança, podendo em muitos casos ser adequadas às necessidades dos utilizadores.

Existem diferentes tipologias de serviços para nuvens de computação, com flexibilidade para diferentes requisitos. Estas tipologias são diferenciadas quanto ao tipo de recursos, modelos e serviços de computação: *Infrastructure-as-a-Service* (IaaS), onde a *cloud*[2]

[1] `http://aws.amazon.com/ec2/` acedido a 11-03-2013

[2] Ao longo do relatório da dissertação, utilizar-se-á o termo "cloud" correspondente ao equivalente de tradução "nuvem" em língua portuguesa. Esta opção faz-se por questões de simplificação, tendo em conta o uso corrente do termo.

oferece serviços da sua infraestrutura, como CPU, memória e armazenamento; *Platform-as-a-Service* (PaaS), onde é disponibilizado um ambiente de execução para o utilizador (como um SGBD, *DaaS* [Aba09]) e *Software-as-a-Service* (SaaS) onde a *cloud* disponibiliza uma aplicação especifica, geralmente acessível através do *browser*.

A criação de um centro de dados próprio (*cloud* privada) envolve custos bastante elevados, quer a nível material quer a nível de gestão e manutenção. Para além disso, envolve um investimento constante de modo a acompanhar a evolução do sistema e suportar toda a carga de trabalho associada. A rentabilidade a nível económico muito dificilmente se assemelha à de uma *cloud* pública, pois o sistema tem de ser desenvolvido para suportar períodos de elevada e baixa utilização de forma eficiente. Isto envolve que, para uma taxa de utilização bastante baixa, vários recursos estejam a ser desperdiçados [AFGJKKLPRSZ10]. As *clouds* públicas existentes oferecem poder computacional e espaço "infinitos", disponíveis à medida das necessidades do cliente, permitindo acompanhar o crescimento gradual, através de uma escalabilidade dinâmica [CPK10].

No entanto, mesmo considerando as anteriores características e vantagens, a manutenção segura e confiável de dados privados ou dados críticos de diferentes aplicações, quando a base computacional de confiança tem pouco ou nenhum controlo por parte dos utilizadores, pode ser bastante problemática. Tal pode impedir ou inviabilizar a adoção das anteriores soluções.

## 1.2 Segurança em *clouds* de computação e armazenamento

No caso de muitas aplicações que gerem dados críticos ou sensíveis, a opção por serviços de armazenamento de dados mantidos por terceiros só é realista se forem garantidas propriedades de segurança e privacidade, bem como fiabilidade e disponibilidade permanente. Para o efeito, estas propriedades devem ser preservadas sob controlo e auditoria independente por parte dos utilizadores. Só deste modo, os utilizadores poderão adotar esses serviços como sistemas confiáveis.

Existe ainda um conjunto de fatores que levam à fraca adoção destas soluções [Cho10; CPK10; Aba09; JG11; ABCKPR09]. Um destes fatores está associado ao receio de que os dados sejam expostos, tanto por parte de ataques à infraestrutura da *cloud*, como exposição dos mesmos ao próprio fornecedor [BCHHHRV09]. O controlo e auditoria poderia eventualmente ser endereçado por mecanismos que complementassem a possibilidade de intervenção e controlo dos utilizadores ao nível da gestão e auditoria das infraestruturas computacionais dos provedores ou por contratação de níveis de responsabilidade de serviço imputáveis aos provedores que cobrissem condições de disponibilidade, recuperação, confiabilidade, integridade e privacidade dos dados (SLAs [BCHHHRV09]). Tal não é porém possível de adequar para que as soluções existentes se mantenham competitivas e interessantes do ponto de vista dos custos. É pois necessário que as propriedades de segurança e fiabilidade, nomeadamente: autenticidade, privacidade, integridade, controlo de acesso, tolerância a falhas ou intrusões, recuperação e disponibilidade de dados

mantidos em provedores de nuvens de armazenamento na Internet sejam auditadas e controladas pelos próprios utilizadores.

O uso de múltiplas *clouds*, apenas responsáveis pelo armazenamento dos dados cifrados, pode assegurar que as operações são efetuadas de forma confiável [JSBGI11], pois todo o controlo está do lado dos utilizadores, independentemente do nível e garantias que possam ser oferecidas por parte de cada provedor em particular. Em *clouds* de computação, é viável a separação da componente lógica e dos dados por fornecedores distintos pois, em caso de falha, os dados não estão diretamente em risco. Outra abordagem é o particionamento da aplicação em camadas e posterior distribuição das mesmas por *clouds* distintas, evitando revelar ao provedor o processamento lógico efetuado. Finalmente, pode-se considerar um fornecedor $A$, responsável pela monitorização da execução de uma outra *cloud* $B$, que pode enviar resultados intermédios das suas operações a $A$.

## 1.3 Armazenamento e gestão confiável de dados

Para um armazenamento e gestão confiável de dados existem certos aspetos a ter em conta, já mencionados anteriormente, e um pouco mais detalhados de seguida.

### Aspetos de disponibilidade

Esta é uma preocupação constante, pois o sistema tem de estar sempre disponível, mesmo na ocorrência de falhas. Para além da grande cobertura geográfica (com centros de dados em vários pontos do globo), com consequente descentralização do sistema (Sistemas como o Bigtable [CDGHWBCFG08], Cassandra [LM10] ou Dynamo [DHJKLPSVV07]) , muitos fornecedores não conseguem assegurar a disponibilidade do sistema durante 100% do tempo. Há registo de situações onde os sistemas ficaram indisponíveis durante um certo período [AFGJKKLPRSZ10], prejudicando várias empresas que dos mesmo dependem. Verifica-se que a melhor abordagem é a de confiar os dados a um conjunto de *clouds*, em vez de apenas uma. Este caso é comparável ao dos ISPs, que por sua vez recorrem a diversos provedores de modo a não existir um ponto único de falha.

### Aspetos de fiabilidade

Um sistema com elevada fiabilidade é visto como sendo de confiança e, na presença de falhas, consegue manter o seu correto funcionamento. Está livre de erros e os seus resultados são sempre previsíveis. A fiabilidade pode ser assegurada com recurso a algoritmos de tolerância a falhas bizantinas [CL99; AEMGGRW05; CMLRS06; KADCW07; CKLWADR09; LM04], com base na replicação dos dados. A distribuição replicada dos dados por um conjunto de *clouds* públicas oferece elevada fiabilidade, pois para além desta encontra-se também a que é assegurada por cada *cloud* individualmente.

### Aspetos de privacidade

Os dados armazenados devem ser previamente encriptados, de forma a preservar a sua confidencialidade. É necessário assegurar que dados sensíveis nunca venham a ser expostos, mesmo na eventualidade de um atacante ter acesso aos mesmos. Não obstante, a confidencialidade perante o próprio provedor também deve ser mantida e é abordada no tópico seguinte. Sistemas como o CryptDB [PRZB11], DepSky [BCQAS11], EHR [NGSN10], Silverline [PKZ11] ou HP time vault [MHS03] abordam este problema a vários níveis e necessidades, com diferentes aplicabilidades no mundo real.

### Aspetos de independência do fornecedor (*vendor lock-in*)

A auditabilidade e controlo do lado do utilizador garantem condições de independência dos fornecedores, com controlo autónomo da base de confiança de manutenção e gestão permanente dos seus dados. Tal exige soluções de resiliência auditáveis por parte dos utilizadores, que resistam a eventuais quebras de serviço, a intrusões ao nível da infraestrutura ou aplicações dos provedores, ou à possível operação incorreta, maliciosa ou ilícita por parte de pessoal de equipas operativas dos provedores. É igualmente importante que essas soluções impeçam formas de bloqueio ou evitem práticas negociais ilícitas que podem ser exploradas pelos provedores, em relação aos dados de utilizadores. Por outro lado, verifica-se a dificuldade de migração de um fornecedor para outro, bem como a migração completa de um volume grande de dados para a *cloud* [AFGJKKLPRSZ10].

### Outros aspetos relevantes

A escalabilidade é também uma propriedade bastante importante. Um sistema altamente escalável e com um tempo de resposta bastante pequeno face a alterações a este nível, assegura que os recursos estão sempre devidamente alocados. Garante-se que estes não estão a ser usados desnecessariamente, ou que não estão recursos em falta. Em [Aba09] verifica-se que a *cloud*, por si só, não é viável para sistemas de bases de dados transacionais e é abordado o problema de sistemas não transacionais de analise de dados na *cloud* (*data warehouse*).

## 1.4 Coordenação e controlo de dados replicados em *clouds*

Assumindo o uso de múltiplas *clouds* de armazenamento, tem de haver um mecanismo de controlo que permita saber por onde estão repartidos os vários dados. A abordagem passa pelo desenvolvimento de uma *cloud* intermédia que efetua toda a gestão dos dados armazenados, mantendo as chaves secretas que lhes permitem acesso, bem como provas de integridade sobre os mesmos e replicação pelas várias *clouds*. É possível tirar grande partido do uso de múltiplas *clouds*, como mencionado acima, nomeadamente nos aspetos de disponibilidade, confidencialidade e em problemas como *vendor lock-in*. Esta é uma abordagem levada a cabo em sistemas como [BCQAS11; JGMSV08].

## 1.5 Objetivos e contribuições da dissertação

### 1.5.1 Objetivos previstos

O objetivo desta dissertação visa conceber, implementar e avaliar um sistema que permita o acesso, gestão e pesquisa confiável de dados privados, mantidos em nuvens de armazenamento de provedores Internet (ou *Internet Cloud-Storage Solutions*).

O sistema proposto tem em vista a sua utilização como sistema intermediário (agregando um conjunto de componentes e serviços numa arquitetura *middleware*) entre o utilizador e diferentes *clouds* públicas de armazenamento de dados, disponibilizadas por provedores Internet. O sistema suporta operações de leitura, escrita e pesquisa confidencial em tempo real sobre dados mantidos nas diferentes nuvens de armazenamento.

O sistema (que designaremos por T-Stratus[3]), preserva garantias de autenticidade, privacidade, fiabilidade, tolerância a intrusões e disponibilidade dos dados, sob controlo dos utilizadores, de forma independente dos provedores, atuando como uma base confiável de computação controlada por esses mesmos utilizadores.

A solução objetivada suporta encriptação e distribuição de dados por múltiplas nuvens de armazenamento heterogéneas, permitindo a sua utilização de uma forma transparente. O armazenamento distribuído dos dados faz-se por um processo de fragmentação e replicação, sendo os fragmentos cifrados e replicados em múltiplas nuvens de armazenamento, numa arquitetura de nuvem de nuvens.

A solução permite a manutenção e replicação de dados privados em múltiplos provedores Internet, usufruindo das vantagens destes serviços, mas assegurando o controlo da base de confiança associada à preservação das propriedades de segurança e confiabilidade.

### 1.5.2 Contribuições da dissertação

Como contribuições esperadas pretende-se a criação de um sistema de armazenamento de dados seguro, que forneça as seguintes propriedades:

**Confidencialidade dos dados,** de modo a estabelecer condições de garantias de confidencialidade face a falhas, a atos ilícitos ou ataques que podem ser desencadeados, seja ao nível da gestão e administração dos sistemas computacionais por parte de pessoal afeto aos provedores, seja por atacantes externos que atuem por intrusão, com base na exploração de eventuais vulnerabilidades das soluções de software operadas por aqueles provedores;

[3] A designação T-Stratus, resulta da terminologia em língua inglesa "Trust Stratus". A designação tem como inspiração um modelo de sistema baseado numa solução proxy, estruturada com base num ou mais servidores, funcionando como uma nuvem baixa (stratus *cloud*) usada como base de confiança auditável por parte dos utilizadores, para controlo de confiabilidade e privacidade de dados armazenados em nuvens "altas ou médias". Estas últimas correspondem a nuvens heterogéneas de armazenamento de dados de provedores Internet, usadas sem controlo independente de auditabilidade e confiabilidade, por parte daqueles mesmos utilizadores.

**Integridade dos dados armazenados** de modo a estabelecer proteção contra alteração dos mesmos, quer essa alteração seja consequência de corrupção dos dados provocada por erros ou falhas dos sistemas de armazenamento, quer a alteração seja consequência de atos ilícitos ou ataques. Considera-se que estes podem ser desencadeados, quer ao nível da gestão e administração dos sistemas computacionais por pessoal afeto aos provedores, quer por atacantes externos que visem explorar vulnerabilidades das soluções de software operadas por aqueles fornecedores;

**Suporte para múltiplas *clouds* de armazenamento,** tal como são disponibilizadas por provedores Internet, de modo a serem usadas numa arquitetura de *cloud* de múltiplas *clouds*, com a necessária criação de componentes responsáveis pelo mapeamento, indexação, distribuição (com replicação) e acesso aos dados, sendo estes guardados como múltiplos fragmentos mantidos de forma replicada nas diferentes *clouds* que sejam utilizadas.

**Indexação dos dados (ou fragmentos de dados)** inseridos nas várias *clouds* de armazenamento, de forma a poder reconstruir esses dados, através da obtenção dos vários blocos de fragmentos constituintes, que se podem encontrar em várias *clouds*;

**Acesso rápido com recurso a mecanismos de *cache*,** onde os pedidos mais recentes ou mais requisitados serão mantidos para posterior rápido acesso;

**Pesquisa segura (confidencial)** sobre dados privados mantidos cifrados, permitindo efetuar uma pesquisa sobre esses dados cifrados, sem nunca expor à infraestrutura da *cloud* as chaves criptográficas que os protegem;

**Fiabilidade e tolerância a falhas bizantinas,** com recurso a algoritmos de consensos bizantinos de dados escritos ou lidos a partir de fragmentos replicados em diversas *clouds* e que asseguram a consistência dos dados mesmo em caso de falha ou ataques por intrusão em algumas das *clouds* utilizadas;

**Disponibilidade permanente** dos dados, face à eventual indisponibilidade de uma ou mais *clouds* ou por alguma ação maliciosa do tipo "vendor lock-in", desde que a resiliência permitida pelo número de *clouds* usadas permita a reconstituição dos mesmos.

A avaliação experimental do sistema proposto, que agrega as contribuições anteriores, envolve a avaliação de métricas de latência e de taxas de transferência de dados em escrita e leitura, utilizando *clouds* de armazenamento reais de provedores Internet. Esta avaliação permite comparar a eficiência da solução proposta face à utilização das atuais soluções de armazenamento em nuvem, que não oferecem as mesmas garantias de confiabilidade, segurança e privacidade dos dados.

De acordo com o que antes foi referido, as contribuições da dissertação estão focadas num cenário em que os serviços de *middleware* da solução T-Stratus permitem a adoção de múltiplas *clouds* de armazenamento, tal como são vulgarmente disponibilizadas por atuais provedores Internet, tendo em vista garantir propriedades de confiabilidade nas suas diversas facetas, nomeadamente: fiabilidade, disponibilidade e segurança, sob controlo

dos utilizadores. De entre as propriedades de segurança, consideram-se as propriedades de confidencialidade, integridade e privacidade dos dados ou fragmentos de dados mantidos nas múltiplas *clouds* de armazenamento utilizadas. Implicitamente, a utilização de provas de integridade envolvendo técnicas do tipo MAC (ou *Message Authentication Codes*), permitirá também controlo de autenticidade dos fragmentos, o que também poderá ser assegurado por provas de autenticação com métodos de assinatura digital com chaves públicas (utilizando adequadamente processos de criptografia assimétrica ou de chave pública) e certificados de chave pública.

Deve salientar-se desde já que a dissertação toma como foco principal a proteção dos dados (ou fragmentos de dados) mantidos nas infra-estruturas de provedores de *clouds* de armazenamento, tendo em vista a possibilidade de ocorrência de falhas ou intrusões, ao nível dessas infra-estruturas computacionais.

Entende-se que a proteção de propriedades de segurança ao nível das comunicações, no acesso às *clouds* de armazenamento Internet, são protegidas pelas soluções usuais que estão disponíveis, utilizando SSL (ou TLS) com autenticação unilateral das *clouds* (ou autenticação mútua entre o sistema T-Stratus e cada *cloud* utilizada).

Entende-se também que a utilização das *clouds* de armazenamento se faz com base em mecanismos usuais de controlo de acessos de utilizadores (seja com base em pares "userID/password" quer com possibilidade de utilizar outras soluções de autenticação e controlo de acessos, com sistemas diversos baseados em múltiplos fatores de autenticação, nomeadamente: controlo biométrico (do tipo leitores de impressão digital), cartões inteligentes (ou autenticação com *Smartcards*), soluções do tipo OTP (ou "One-Time-Pads"), ou *Tokens* de autenticação com geradores de palavras passe dinâmicas baseadas em dispositivos específicos). Este tipo de mecanismos de autenticação de utilizadores e controlo de acesso são disponibilizados por alguns provedores em soluções mais especializadas de serviços de armazenamento. A dissertação não se foca particularmente sobre este tipo de soluções já existentes que poderão, em todo o caso, ser adotadas para efeitos da solução proposta.

Finalmente, deve destacar-se que a dissertação privilegia a utilização de soluções específicas de *clouds* de armazenamento (e não de *clouds* de computação no sentido estrito). Isto é, não se tem em conta na solução proposta que se execute código ao nível das infra-estruturas das *clouds* utilizadas. Para todos os efeitos, estas apenas serão usadas como repositórios de dados.

## 1.6 Organização do relatório

Seguido deste capítulo introdutório, no Capítulo 2, é feito um levantamento do estado da arte a respeito de sistemas de armazenamento de dados em *clouds* que oferecem, ou não, garantias de confidencialidade dos mesmos. São abordados protocolos de tolerância a falhas bizantinas, baseados na replicação dos dados, bem como a adoção de técnicas como RAID e *erasure codes*, que permitem a recuperação dos dados sem a replicação total

dos mesmos. Por fim, é abordada a problemática da integridade dos dados e, finalmente, a possibilidade de efetuar pesquisas seguras com privacidade dos mesmos.

No Capítulo 3, é feita uma revisão dos objetivos do sistema a implementar e é apresentada a arquitetura do sistema e seus componentes, como arquitetura de *middleware*. Este capítulo é concluído com uma descrição da API fornecida para suporte de aplicações que usarão os serviços *middleware* para integração transparente com diferentes *clouds*, eventualmente heterogéneas.

De seguida, no Capítulo 4 é feita uma descrição mais pormenorizada da implementação levada a cabo, nomeadamente, os aspetos mais relevantes e decisões tomadas.

O Capítulo 5 é dedicado à avaliação experimental do sistema proposto, com base na avaliação das diversas métricas previamente definidas. Estas observações experimentais permitem analisar o funcionamento do sistema e a sua validação, comparando o uso de *clouds* de armazenamento, tal como são hoje disponibilizadas por diversos provedores na Internet

Por fim, o Capítulo 6 apresenta as ilações e contribuições finais com base na avaliação dos resultados. Este capítulo conclui-se com uma subsecção de trabalho futuro a ter em conta como extensão ao sistema desenvolvido.

# 2

# Trabalho relacionado

Neste capítulo apresentam-se e discutem-se diversas referências de trabalho relacionado com os objetivos e contribuições previstas para a dissertação. Na secção 2.1.1 e 2.1.2 são analisados sistemas distribuídos de armazenamento de dados bem conhecidos, que garantem elevado desempenho, disponibilidade, consistência e integridade dos dados. A secção 2.1.3 apresenta algumas referências de soluções para armazenamento de dados na *cloud*. De seguida, na secção 2.2, são analisados vários sistemas e algoritmos de replicação de dados com tolerância a falhas bizantinas. A secção 2.3 apresenta alguns sistemas de referência que fazem uso dos anteriores mecanismos combinados com processos criptográficos para endereçarem o objetivo da gestão confiável de dados em repositórios geridos por terceiros. Finalmente, na secção 2.4, é abordado trabalho relacionado relativamente à problemática das garantias de privacidade de dados mantidos na *cloud*, com ênfase em mecanismos criptográficos que endereçam o suporte a pesquisa de dados privados, cifrados e guardados em repositórios públicos, preservando a proteção de privacidade durante as pesquisas sem expor chaves criptográficas.

## 2.1 Gestão de dados em *clouds*

### 2.1.1 Gestão de dados em *clouds* de armazenamento

Dos sistemas distribuídos estudados para a gestão de dados em nuvens de armazenamento existe uma propriedade importante, comum a todos eles, que é o facto de serem descentralizados (sem um ponto único de falha). Os sistemas Cassandra [LM10] e Dynamo [DHJKLPSVV07] são mais focados num serviço que oferece garantias de elevada disponibilidade. No Dynamo, o principal objetivo é que o sistema se mantenha sempre

disponível, mesmo com a ocorrência de falhas ou outros fatores adversos. Juntamente com o Bigtable [CDGHWBCFG08], estes três sistemas garantem elevada fiabilidade, performance, disponibilidade e são escaláveis para um número elevado de nós na rede.

O Bigtable [CDGHWBCFG08], regra geral, lida com grandes volumes de dados e a sua estrutura interna consiste num mapa multi-dimensional ordenado (que representa a tabela). Cada chave é caracterizada pela linha (*string*), coluna (*string*) e por um identificador temporal (*timestamp*), sendo que cada valor corresponde a um conjunto de *bytes*. O valor de *timestamp* permite ter diferentes versões dos dados e é guardado em ordem decrescente, de modo a obter sempre, numa leitura, a versão mais recente primeiro. As linhas de uma tabela são particionadas em conjuntos denominados *tablets*, permitindo uma melhor distribuição das mesmas pelos vários nós, distribuindo assim a carga computacional. O elevado desempenho deve-se também à ordenação pois, geralmente, o utilizador pretende obter dados dentro de um determinado domínio (algum subconjunto de linhas da tabela) que poderá estar em apenas um servidor. O controlo de acessos é feito através da indexação das colunas e estas estão divididas em famílias de chaves, ou seja, uma família é um conjunto (agrupamento) de chaves de várias colunas. O sistema em si funciona através de um servidor principal e vários servidores de *tablet* que, consoante as necessidades, podem ser dinamicamente atribuídos ou removidos. A elevada escalabilidade permite um aumento do desempenho com a adição de novos nós ao sistema.

À semelhança do Bigtable, o Cassandra e o Dynamo baseiam-se em armazenamento do tipo chave-valor (modelo não relacional) para o armazenamento dos dados. A estrutura interna é também, em ambos, um mapa multi-dimensional indexado por uma chave. Os dados são particionados e replicados pelos vários servidores, sendo as réplicas mantidas de forma consistente através de um algoritmo baseado em quórum. Tanto para operações de leitura como de escrita, o sistema aguarda por um quórum de réplicas de resposta, de forma a confirmar a operação. O particionamento dos dados pelos vários *clusters* (nós) é feito em ambos de forma semelhante, onde são usadas funções de *hash* sobre as chaves e cada nó do sistema tem atribuído a si um dado intervalo desses valores (Figura 2.1), que corresponde às chaves pelas quais é responsável. A carga de trabalho é distribuída de forma uniforme pelos vários nós através da atribuição de conjuntos de valores pertencentes a esses intervalos. Por fim, os dados são replicados por vários nós e também ao nível dos centros de dados, evitando assim que uma falha por catástrofe natural (ou outro fator), que afete todo um centro de dados, não ponha em risco os dados presentes no sistema.

No Cassandra, as colunas são também agrupadas em famílias, sendo a ordenação parametrizável através das mesmas. As operações são atómicas por linha e existem três operações básicas para inserir, remover ou obter o conteúdo de uma dada linha, de uma dada tabela e de um dado conjunto de colunas. No Dynamo, as operações são de *put* (inserção) e *get* (obtenção). A consistência dos ficheiros é mantida através da existência de várias versões do mesmo, sempre que há uma escrita.

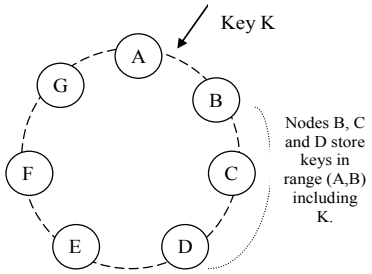



Figura 2.1: Atribuição das chaves pelos vários nós do sistema no Dynamo [DHJKLPSVV07]

### 2.1.2 Riak

O Riak [Ria], baseado no Dynamo [DHJKLPSVV07], consiste numa base de dados NoSQL, que implementa os princípios do mesmo. O armazenamento é, portanto, do tipo chave-valor e podem existir diversos nós (*cluster*), responsáveis por um determinado conjunto de chaves, distribuídas de forma uniforme, tal como especificado no Dynamo. Não possui um ponto único de falha, é dotado de uma elevada capacidade de tolerância a falhas e foi idealizado para correr em ambientes distribuídos (como a *cloud*). Este repositório distribuído chave-valor vem de origem com o mecanismo de armazenamento Bitcask [SSBT10], inicialmente desenvolvido com o propósito de ser usado no Riak, face às necessidades que os autores identificaram. O Bitcask oferece latências baixas para escritas e leituras, bom rendimento com taxas de transferência elevadas, capacidade para lidar com grandes conjuntos de dados, rápida recuperação face a falhas e ausência de perda de dados, sendo os mesmos facilmente recuperáveis e salvaguardados.

O modelo do Bitcask pressupõe que cada instância corresponde a uma diretoria, apenas acessível para escrita por um processo de cada vez. A qualquer momento, apenas um ficheiro dessa diretoria está ativo para escritas por parte do servidor. Quando o mesmo atinge um dado limite, é "fechado", e é criado um novo, tornando-se o antigo imutável. O ficheiro de escrita aceita um formato em que, para cada par chave-valor, recebe um *checksum*, um *timestamp*, o tamanho da chave, o tamanho do valor, a chave e o valor propriamente dito (Figura 2.2). Existe ainda uma estrutura em memória (*keydir*, Figura 2.3)

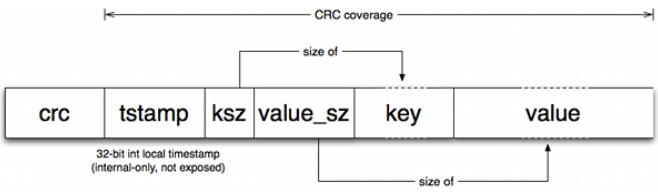



Figura 2.2: Formato de cada par chave-valor adicionado ao ficheiro ativo [SSBT10]

que consiste numa *hash table* onde é feito o mapeamento entre as várias chaves e o ficheiro e posição onde se encontra o valor correspondente. Por fim, existe ainda um processo de

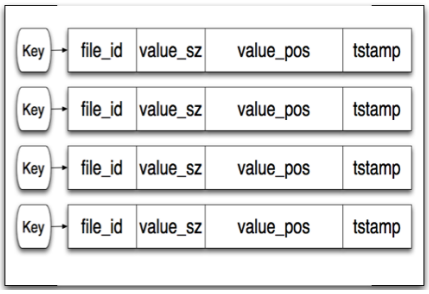



Figura 2.3: Mapeamento entre as várias chaves, respetivos ficheiros e posições dos valores [SSBT10]

compactação que vai iterar sobre todos os ficheiros imutáveis e produz um ou vários ficheiros resultado apenas com as versões mais recentes para cada chave. Neste processo, é também criado um ficheiro (*hint file*), associado a cada um dos anteriores. Este permite

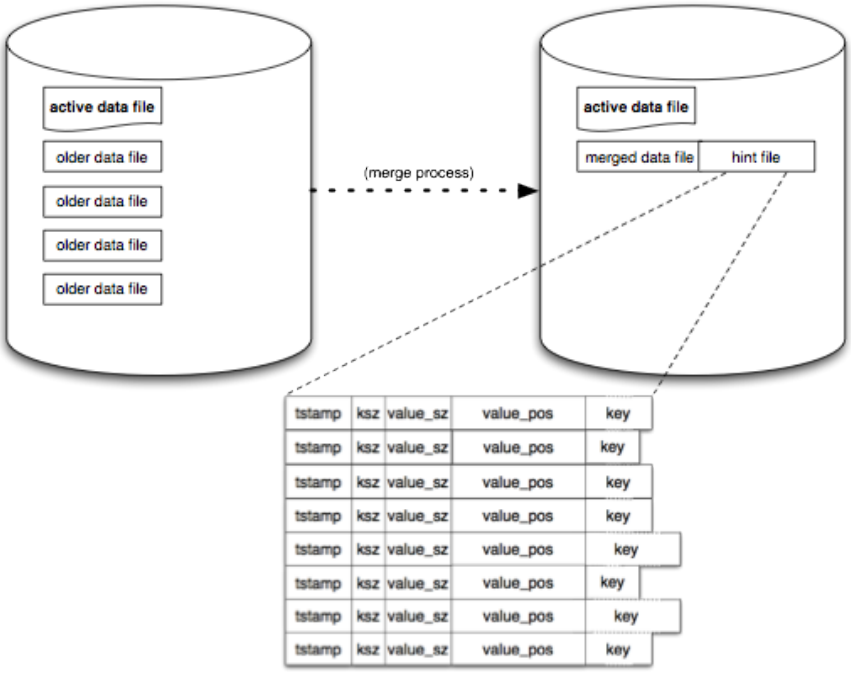



Figura 2.4: *hint file* [SSBT10]

que na eventualidade de um novo processo, este possa construir a estrutura *keydir* de uma forma mais rápida e eficiente, pois as *hint files* possuem, entre outros, a chave e a posição do respetivo valor no ficheiro imutável correspondente.

### 2.1.3 Fornecedores de *clouds*

Nesta secção são apresentados vários serviços de armazenamento de dados na *cloud*. Todos, com exceção do Bitcasa, apresentam uma API REST, para acesso às várias operações e possibilidade de integração noutro sistema, bem como um conjunto de operações standard, nomeadamente, obtenção, inserção e remoção de um objeto.

- A **Google Cloud Storage**[1] permite a partilha de ficheiros e garante a consistência dos dados armazenados. As operações de armazenamento são feitas de forma atómica e os objetos são guardados em contentores, denominados *buckets* (com possível controlo de acessos, através do uso de ACLs). É possível especificar a localização geográfica dos *buckets*, de forma a otimizar a performance.
- O **Windows Azure Storage** [Cal+11] armazena os dados de forma replicada localmente e geograficamente. Disponibiliza armazenamento ao nível de *blobs* (ficheiros), de tabelas (armazenamento estruturado) e de filas. No geral, é um serviço que garante elevada consistência, disponibilidade, particionamento e redundância ao nível do armazenamento dos dados.
- Na **Amazon S3** [Ama] os dados são geridos e replicados pelos diversos servidores. Fornece gestão dos dados ao nível de armazenamento, acesso, backup e recuperação. Podem-se criar *buckets*, à semelhança da Google Cloud Storage, sobre os quais também se pode especificar uma zona geográfica.
- O **Cloud Files**[2] (serviço de armazenamento disponibilizado pelo **Rackspace**) garante replicação dos dados pelos diferentes nós do sistema, geograficamente distribuídos. Os dados são organizados/agrupados por contentores e estes podem ser privados, onde a comunicação é feita por SSL, ou públicos, onde os dados têm associado um URL que permite a sua partilha.
- O **Nirvanix**[3] permite a partilha de ficheiros ou pastas através de links públicos e pode também usar SSL para a criação de uma ligação segura. O sistema está implementado sobre RAID 6 [CLGKP93] e garante segurança, fiabilidade e redundância dos dados armazenados.
- A **Luna Cloud**[4] é relativamente recente e bastante promissora. Apresenta três tipos de serviços: *Cloud Appliance*, onde são disponibilizadas aplicações pré-instaladas de acordo com as necessidades do utilizador; *Cloud Server*, onde o utilizador tem acesso a um servidor (várias distribuições Linux disponíveis ou Windows Server 2008) e paga pelos recursos requeridos (RAM, CPU e disco) e, por fim, o *Cloud Storage*, onde é possível armazenar dados numa estrutura do tipo chave-valor, com a vantagem de que grande parte das operações são compatíveis com a API da Amazon S3 [Ama].
- O **Bitcasa**[5] apresenta um conceito onde todo o sistema de ficheiros se encontra virtualizado, ou seja, na *cloud*. Os dados são mantidos de forma cifrada do lado do cliente. O Bitcasa recorre a algoritmos de duplicação de ficheiros, previsão dos dados que o utilizador vai precisar (que são guardados em disco local) e técnicas de

[1] http://code.google.com/apis/storage/docs/getting-started.html acedido a 28-01-2012

[2] http://www.rackspace.com/cloud/cloud_hosting_products/files/ acedido a 28-01-2012

[3] http://www.nirvanix.com/products-services/cloudcomplete-public-cloud-storage/index.aspx acedido a 28-01-2012

[4] https://www.lunacloud.com/en/cloud-storage acedido a 06-01-2013

[5] http://techcrunch.com/2011/09/12/with-bitcasa-the-entire-cloud/ acedido a 28-01-2012

compressão e encriptação dos mesmos.

- A **Dropbox**[6] é um serviço de armazenamento de ficheiros na *cloud*, bastante conhecido e utilizado hoje em dia. O armazenamento é assegurado pelo uso transparente da Amazon S3, mencionada acima, e para o acesso aos dados são disponibilizados, tanto um cliente local (software instalado no próprio computador), como o acesso via *web*, ou mesmo uma API disponível em várias linguagens de programação.

### 2.1.4 Discussão

Os sistemas estudados inicialmente, nomeadamente, o Bigtable [CDGHWBCFG08], Cassandra [LM10], Dynamo [DHJKLPSVV07] e Riak[Ria] não possuem suporte para privacidade dos dados mantidos em repositórios do tipo "key-value", sendo estes aspetos deixados para aplicações que adotem estes sistemas como soluções de gestão de dados distribuídos. São propostas mais focadas em aspetos de elevada disponibilidade, escalabilidade e desempenho na gestão de grandes volumes de dados. Utilizam replicação para suportarem alta disponibilidade em ambientes de larga escala, garantindo recuperação fiável e permanente dos dados armazenados. São sistemas interessantes no âmbito da presente dissertação, no entanto, não endereçam a questão da confidencialidade dos dados, pois não é seu objetivo. Essa propriedade é deixada para as aplicações de nível acima. Não obstante, e de modo a dotar o sistema de uma capacidade suficientemente modular, é necessário equacionar a adoção de um sistema destes para o armazenamento do índice associado aos ficheiros armazenados na *cloud*. Na verdade, dada a grande semelhança dos vários sistemas a opção torna-se de difícil escolha. Essa escolha recai sobretudo no Cassandra e no Riak, visto terem como fontes de inspiração sistemas como o BigTable e o Dynamo. Visto o objetivo ser apenas o armazenamento de pares chave-valor, sem qualquer tipo de organização aparente no que diz respeito ao tipo de colunas (parte em que o Cassandra é bastante influenciado pelo BigTable, nomeadamente nas famílias de colunas), o Riak apresenta-se como um sistema bastante promissor e mais orientado à gestão de documentos, simplificando toda a gestão dos mesmos. Apresenta um modelo de dados mais simples, ideal para a utilização que se pretende, e consiste numa implementação mais fiel do Dynamo. Apresenta um modelo de consistência com base em *vector clocks*, ao contrário dos *timestamps* no Cassandra, onde uma má sincronização entre os relógios dos vários nós pode dar origem a problemas. Apesar da implementação do Riak ser em Erlang[7] (por sua vez, uma linguagem reconhecida por um bom sistema de concorrência), este apresenta, entre outras, uma API Java para o desenvolvimento de um cliente para acesso às várias operações disponíveis.

Ao longo da secção anterior foram também enumeradas várias *clouds* de armazenamento disponíveis como soluções de fornecedores na Internet. Regra geral, são soluções baseadas em repositórios de dados de grande escala, replicados internamente numa infraestrutura computacional distribuída e abrangente, permitindo o acesso aos dados de

[6] https://www.dropbox.com/ acedido a 12-03-2013
[7] http://www.erlang.org/ acedido a 07-01-2013

forma ubíqua. As soluções destes fornecedores replicam os dados em diferentes zonas geográficas, garantindo otimização de latência de acesso quando os utilizadores se encontram em diferentes zonas mundiais cobertas pela Internet. Estas soluções podem ser facilmente usadas pelas aplicações via interfaces baseadas em serviços WEB (*web services* ou REST) com um conjunto standard de operações para suportar escritas e leituras concorrentes sobre um repositório estruturado, do tipo "key-value store". A definição de operações e tipo de dados pode variar caso a caso. Geralmente, o modelo de concorrência base destas soluções é do tipo "one-writer, n-readers"ou "N writers-N readers and Last-Writer Wins". Algumas (Google, por exemplo), poderão permitir modelos de concorrência com semântica forte, orientada a transações que podem agregar a inserção de diferentes chaves. O único fornecedor que oferece confidencialidade nativa dos dados armazenados é a solução Bitcasa. Este anuncia que todas as escritas são automaticamente cifradas por um processo de cifra de características homomórficas, não existindo porém informação publicada sobre como funciona esse processo ou que algoritmos criptográficos são utilizados. Em qualquer um dos casos, o utilizador final nunca tem controlo autónomo sobre a base de confiança do sistema.

A utilização segura e fiável de *clouds* de armazenamento de dados pode ser endereçada por sistemas capazes de usar simultaneamente múltiplas *clouds* de armazenamento. São exemplos sistemas como o DepSky [BCQAS11] ou o iDataGuard [JGMSV08], mencionados mais à frente.

## 2.2 Mecanismos de tolerância a falhas

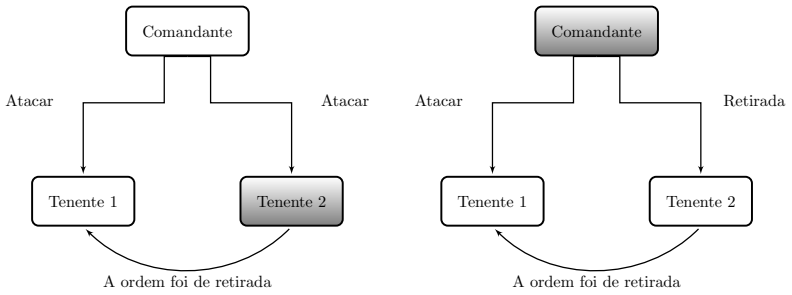
### 2.2.1 Falhas bizantinas

A tolerância a falhas não pode ser vista apenas como a eventualidade de avarias em hardware, a indisponibilidade de algum componente ou serviço, ou o ataque por parte de terceiros. Existem falhas ao nível dos componentes que podem fazer com que estes entrem num estado de erro, ou seja, produzam resultados inesperados, geralmente errados e possivelmente conflituosos. Quando aplicado a sistemas distribuídos, onde é necessário um elevado nível de cooperação entre os vários componentes, torna-se bastante importante prevenir e recuperar o sistema perante este cenário, de forma a que estes não transmitam entre si dados corrompidos.

#### 2.2.1.1 O problema dos generais bizantinos

O conceito de falhas bizantinas tem origem num artigo de 1982, The Byzantine Generals Problem [LSP82], em que esta situação é exposta sob um cenário de guerra, onde um grupo de generais do exército Bizantino pretendem comunicar entre si os planos de batalha, através de mensageiros, em volta de uma cidade inimiga. O objetivo é que os generais leais cheguem a um consenso, mesmo na presença de generais desleais que possam fazer circular informação conflituosa.

Assumindo um cenário onde as mensagens não são assinadas e onde participam três intervenientes: um comandante, emissor da mensagem e dois tenentes (recetores), verifica-se a impossibilidade de recuperar de uma falha ao nível das mensagens trocadas. As duas regras a respeitar são: Todos os tenentes leais obedecem à mesma ordem e, se o comandante for leal, todos os tenentes leais obedecem à ordem que este enviou. Como exemplo, assumem-se apenas duas mensagens possíveis: "atacar"e "retirada". Na Figura 2.5(a) está ilustrado o caso de uma disseminação de uma mensagem onde um dos tenentes é traidor e na Figura 2.5(b) o caso onde o comandante é traidor e envia uma mensagem diferente para cada um dos dois tenentes. As condições não são respeitadas e o tenente 1 não consegue saber quem é o traidor, pois não consegue saber que mensagem foi enviada pelo comandante ao tenente 2. Verificou-se que são necessários $3m + 1$ generais para um

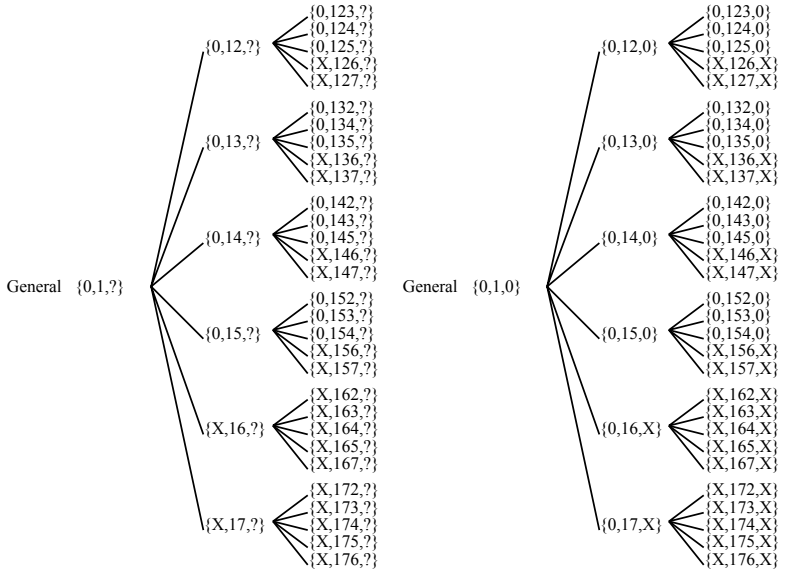



(a) O tenente 2 é traidor. (b) O comandante é traidor

Figura 2.5: O problema dos generais bizantinos com três generais [LSP82]

número de traidores igual a $m$. No entanto, assume-se sempre que as mensagens trocadas satisfazem certas propriedades: são enviadas e recebidas corretamente, sabe-se quem enviou a mensagem e é possível detetar se esta chegou ou não ao destino. O algoritmo descrito funciona em duas fases. Numa primeira fase, o comandante envia a mensagem para todos os tenentes. Ainda nesta fase, durante $m + 1$ rondas, os vários tenentes atuam como comandantes e enviam as mensagens recebidas para todos os seus vizinhos, mantendo um registo do caminho percorrido. Numa segunda fase, os generais tomam uma decisão com base nos valores recebidos, escolhendo a maioria das mensagens recebidas com igual valor. Na Figura 2.6[8] é apresentado um exemplo de troca de mensagens para $n = 7$ generais e $m = 2$. Neste exemplo, o comandante enviou valores certos (0) mas os tenentes 6 e 7 difundiram valores errados (X). Uma mensagem do tipo {0,132,?} significa que o comandante, general com o identificador 1, enviou uma mensagem para o general 3 com o valor 0. De seguida, o general 3 enviou uma mensagem para o general 2 a indicar que o general 1 lhe comunicou o valor 0, e assim sucessivamente para um número $m$ de rondas (neste exemplo, duas rondas). O terceiro valor, assinalado com um ponto de interrogação, consiste no valor de output registado após cada iteração da segunda fase do algoritmo. Cada sub-árvore é vista como uma ronda onde é verificado qual o

[8] http://marknelson.us/2007/07/23/byzantine/ acedido a 30-01-2012




(a) Antes do cálculo do valor de output.

(b) Depois do cálculo do valor de output.

Figura 2.6: O problema dos generais bizantinos com 7 generais, 2 dos quais traidores

valor que aparece na maioria dos casos. Tendo como exemplo o nó {0,12,?}, na Figura 2.6(a), este vai calcular o valor que aparece mais vezes de entre o conjunto de valores de output dos seus filhos {0,0,0,X,X} e vai registar esse valor {0,12,0} (figura 2.6(b)). Assim sucessivamente para cada nó da árvore.

#### 2.2.1.2 Paxos

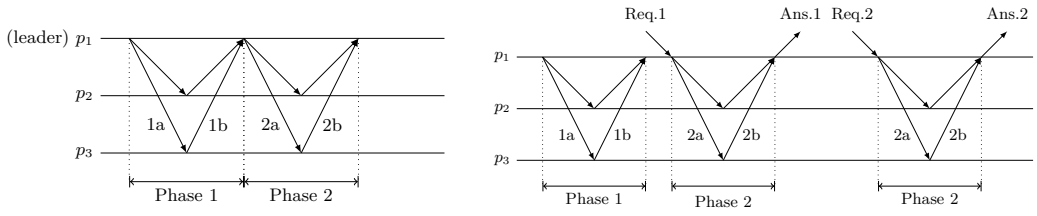
O algoritmo Paxos [LM04] foi descrito pela primeira vez em [Lam98]. Consiste numa família de protocolos e existem, atualmente, várias versões (ou variantes) do mesmo, sendo as mesmas descritas mais abaixo. O Paxos, assume, tal como no PBFT [CL99], o conceito de máquinas de estado replicadas pelos vários nós do sistema, onde cada um deles executa uma dada sequência de operações, derivada de um pedido por parte de um cliente. No final, todos os processos dos vários servidores acordam um valor final para a operação.

De uma forma geral, assume-se a existência de três tipos de agentes no sistema: *proposers*, aqueles que propõem valores; *acceptors*, cooperam entre si para escolher um dos valores propostos e *learners*, que tomam conhecimento dos valores escolhidos. O Paxos assume que inicialmente é escolhido um líder, $l$ (um dos *proposers*), sendo a existência do mesmo crucial no progresso do algoritmo. Os *acceptors* formam conjuntos de quóruns pré definidos, sendo que quaisquer dois quóruns têm obrigatoriamente um *acceptor* em comum. Por fim, assume-se também a existência de um conjunto de números naturais, dos quais cada líder possui um subconjunto distinto dos mesmos.

O algoritmo em si é constituído por duas fases, cada uma com duas sub-fases. As

mensagens trocadas são entre *acceptors* e *learners*, podendo a mesma réplica enviar mensagens para ela própria, visto poder assumir mais do que um papel. Na primeira fase, $l$ escolhe um valor $b$ do seu conjunto de números naturais, acima mencionado, e envia-o a todos os *acceptors*. Um *acceptor*, ao receber a mensagem, verifica se $b$ é maior que qualquer valor recebido até ao momento e, em caso afirmativo, envia uma mensagem de confirmação a $l$. Caso contrário, envia uma mensagem a dizer que vai ignorar o pedido. Neste último caso, se $l$ acredita que ainda é, de facto, líder, envia uma nova mensagem com um número superior a $b$.

Na segunda fase, $l$ tem de definir um valor para a sua proposta. Se existiram propostas previamente aceites por parte dos *acceptors*, estes enviaram os respetivos valores para o *proposer* (líder), que agora tem de atribuir um valor, $v$, à sua proposta, associado ao maior número reportado pelos *acceptors*. Este verifica se recebeu um quórum de mensagens de confirmação de *acceptors* para um dado $b$. Em caso afirmativo, envia uma nova mensagem a todos os *acceptors* com o valor $v$, calculado através das mensagens trocadas na primeira fase ou arbitrariamente escolhido de entre as propostas recebidas. Um *acceptor*, ao receber a mensagem, verifica que $b$ é superior a qualquer outro valor recebido e envia uma mensagem com $v$ e $b$ para todos os *learners*. Se um *learner* receber um quórum de mensagens de *acceptors* para um mesmo $b$ e $v$, toma conhecimento de que o valor $v$ foi escolhido, executa o pedido e envia a resposta ao cliente.




(a) Paxos (b) MultiPaxos

Figura 2.7: Instância do Paxos e MultiPaxos [KSSZWS11].

Este algoritmo tolera até $f$ réplicas falhadas, de um total de $2f + 1$ réplicas. Como mencionado acima, existem várias versões do mesmo, nomeadamente: Paxos estático, onde o conjunto de *acceptors*, bem como os quóruns, são constantes e definidos previamente, e Paxos dinâmico, onde estes são determinados pela própria máquina de estados, como acontece no caso do PBFT [CL99]. O Cheap Paxos [LM04] é outra variação que necessita de apenas $f + 1$ réplicas para o funcionamento do sistema, assumindo $f$ réplicas auxiliares apenas usadas para o tratamento no caso de falha de uma réplica principal. Por fim, existe uma implementação em Java, JPaxos [KSSZWS11], de uma variação denominada MultiPaxos. Assume também o modelo $2f + 1$ e as réplicas chegam a um consenso para uma sequência de valores, em vez de um valor apenas (caso do Paxos). A primeira fase pode ser executada para um conjunto de instâncias, em vez de apenas uma, sendo depois iniciada a segunda fase para cada instância pretendida. A Figura 2.7 representa as mensagens trocadas ao nível dos dois algoritmos, entre três processos, $p1$, $p2$ e $p3$, para

as várias fases (1a significa fase 1, sub-fase a).

Por fim, a família de protocolos Paxos dedica-se apenas à replicação dos dados e ao estabelecimento de um consenso para as operações efetuadas entre os vários processos. São protocolos interessantes no contexto da replicação de dados e na introdução à problemática das falhas bizantinas, isto porque, em rigor, protocolos como os descritos nas subsecções seguintes partem do mesmo principio. Os dados encontram-se replicados pelos vários nós do sistema, sendo que estes mesmos nós chegam a um consenso para a sequência de operações a efetuar, tendo em conta os vários pedidos dos clientes. No entanto são protocolos que, já de si, pretendem resolver o problema da tolerância a falhas bizantinas, tanto a nível de intrusões como falhas nos próprios nós do sistema.

#### 2.2.1.3 Practical byzantine fault tolerance (PBFT)

Neste artigo [CL99] é proposto um algoritmo para a resolução de falhas bizantinas a ser usado num ambiente assíncrono, como é o caso da Internet. É demonstrado o potencial do mesmo quanto à performance, quando comparado com outros anteriormente propostos. É um protocolo de replicação baseado em máquinas de estado, para ser implementado num sistema determinista e apresenta garantias de *safety* e *liveness* (os clientes recebem sempre resposta aos pedidos efetuados) para um total de $3f + 1$ réplicas, onde $f$ são defeituosas. A máquina de estados encontra-se replicada pelos vários nós do sistema e o algoritmo usa autenticação das mensagens com base em criptografia assimétrica, bem como garantias de integridade através de códigos de *hash*.

A configuração das réplicas (denominada *view*) pode alterar ao longo do tempo sendo que, numa dada altura, existe sempre uma réplica primária e várias secundárias. Aquando da ocorrência de uma falha na réplica primária, esta é substituída por outra, outrora secundária, mudando então a configuração do sistema. Garante-se assim a propriedade de *liveness* pois é iniciado o protocolo de troca de *view* assim que existe suspeita de falha da réplica primária. O algoritmo funciona da seguinte forma: 1) O cliente envia um pedido para invocar uma operação à replica primária; 2) A réplica primária envia o pedido às restantes através de *multicast*; 3) As réplicas executam o pedido e enviam a resposta à primária; 4) Do modelo $3f + 1$, o cliente aguarda pela receção de $f + 1$ respostas de réplicas diferentes com o mesmo resultado.

O algoritmo está dividido em três fases: *pre-prepare*, *prepare* e *commit*. As fases de *pre-prepare* e *prepare* servem para ordenar os pedidos enviados numa mesma *view*. As fases de *prepare* e *commit* servem para garantir que os pedidos efetuados com êxito estão totalmente ordenados através das *views*. Todo o processo tem inicio quando a réplica primária, $p$, recebe um pedido, $m$, por parte de um cliente. Na fase de *pre-prepare*, $p$ atribui um número de sequência ($n$) a $m$ e difunde a mensagem para as restantes réplicas. Na fase seguinte, *prepare*, cada réplica secundária, após aceitação da mensagem de *pre-prepare*, envia às restantes uma nova mensagem, com o valor $n$, bem como um identificador da réplica, $i$. Uma réplica encontra-se no estado preparado se contiver no seu

registo (*log*) o pedido $m$, a mensagem de *pre-prepare* de $m$ (com a *view* $v$ e o número $n$) e $2f$ mensagens de prepare de diferentes réplicas que correspondem aos dados da mensagem de *pre-prepare*. Uma vez atingido o estado de preparado por parte de uma réplica, esta envia uma mensagem de *commit* para as restantes. Estas aceitam as mensagens de *commit*, guardando-as no seu *log* e, após a receção de pelo menos $2f + 1$ mensagens de *commit*, referentes ao *pre-prepare* do pedido $m$, considera-se um consenso estabelecido entre as réplicas, sendo a operação executada e o resultado enviado diretamente ao cliente. Na Figura 2.8 está representada a execução do algoritmo para uma réplica primária sem falhas (réplica 0) e uma réplica secundária com falha (réplica 3), sendo $C$ o cliente. O al-

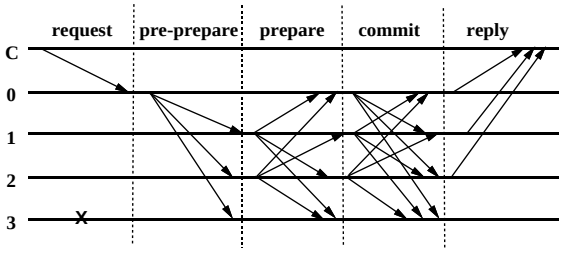



Figura 2.8: Protocolo executado entre quatro réplicas [CL99]

goritmo pode ser otimizado através da seleção de uma réplica, por parte do cliente, para enviar o resultado final enquanto as restantes enviam apenas um *digest* do mesmo, sendo depois efetuada uma comparação dos vários resultados. Outra forma de otimizar o protocolo é retirar a fase de consenso: um cliente difunde a operação para todas as réplicas; cada réplica verifica a validade do pedido, que corresponde efetivamente a uma operação *read-only*; o cliente espera por $2f + 1$ respostas de diferentes réplicas com o mesmo resultado. Caso existam respostas distintas, retransmite o pedido pela via normal, como sendo *read-write*.

#### 2.2.1.4 Fault-scalable Byzantine fault-tolerant services

Um protocolo desenvolvido com vista num sistema escalável a um número elevado de falhas é o Query/Update [AEMGGRW05]. É um protocolo otimista, baseado em quórum e permite uma melhor eficiência com o aumento do número de falhas no sistema, quando comparado com protocolos que recorrem a máquinas de estado replicadas, como o caso anterior. A maior escalabilidade deve-se a apenas quóruns de servidores processarem os pedidos o que proporciona, geralmente, ausência de comunicação entre servidores. Neste protocolo, ao contrário dos estudados anteriormente, são necessários $5b + 1$ servidores para que seja possível tolerar $b$ servidores bizantinos falhados.

Os objetos estão replicados pelos $n$ servidores do sistema. Existem operações de *read-only*, denominadas *queries* e operações de escrita, denominadas *updates*. Os clientes efetuam as várias operações através de pedidos feitos a um quórum de servidores. Um servidor aceita o pedido e executa o método no seu objeto local, sendo que, resultante

de uma operação de *update* está uma nova versão do objeto. É mantido um histórico das várias versões do objeto no servidor e este é enviado aos clientes durante a resposta a um pedido. O cliente guarda todos os históricos recebidos num *object history set* (OHS), que é um *array* de históricos de réplicas, indexado por servidor. Como as respostas só vêm de um subconjunto dos servidores, este OHS representa uma visão parcial do sistema.

Para dar inicio a uma operação, o cliente começa por aceder ao OHS do objeto em questão e, de seguida, envia o pedido com o método a invocar, bem como o OHS do objeto. Os servidores ficam assim com um conhecimento do estado geral do sistema para este objeto. Tanto o cliente como o servidor chegam a um consenso sobre em que versão deve ser feita a operação. Para uma melhor eficiência, cada objeto mantém uma lista de servidores preferidos (quórum preferido), que tentam aceder em primeiro lugar.

Quando um servidor recebe um pedido começa por verificar a integridade da OHS através da comparação de autenticadores (i.e listas de HMACs) das respetivas entradas no histórico da réplica e no OHS. Desta forma garante-se que o histórico global é correto, pois o intermediário da troca de informação entre servidores é o cliente. É feita uma classificação, ou seja, escolha da versão do objeto sob a qual a operação será efetuada. O servidor de seguida envia o OHS no sentido de saber se é de facto a versão mais atual com que se está a trabalhar, para que a operação seja sempre efetuada na versão mais recente do objeto. Isto é feito comparando os candidatos escolhidos a partir do histórico da réplica e do OHS. Caso o servidor possua um candidato superior, devido a uma atualização intermédia, o OHS não é corrente e necessita ser atualizado. Passadas todas as validações, o servidor aceita o pedido e efetua-o no objeto candidato mais recente. Se a operação foi de *update*, resulta uma nova versão do objeto com um novo valor de *timestamp*, calculado de forma a que todos os servidores que efetuem a operação tenham uma versão com o mesmo valor de *timestamp*. O servidor atualiza o histórico correspondente do objeto, indexado pela *timestamp* e envia uma resposta ao cliente indicando o sucesso da operação, o seu resultado, o histórico da réplica e o valor de HMAC. O cliente atualiza o seu histórico de objetos com essa informação e, assim que receber um quórum de respostas corretas, $4f + 1$, dá a operação como executada com sucesso. No caso de operações concorrentes sobre o mesmo objeto, que possam falhar ou dar origem a históricos de réplica diferentes em servidores diferentes, o cliente tem de fazer com que os servidores alcancem um estado consistente, invocando uma operação de *repair*.

#### 2.2.1.5 HQ replication: a hybrid quorum protocol for byzantine fault tolerance

As duas abordagens para tolerância a falhas bizantinas faladas inicialmente nesta secção (PBFT [CL99] e Q/U [AEMGGRW05]) têm alguns aspetos negativos, nomeadamente, no que diz respeito às comunicações entre réplicas, no caso do PBFT, e ao elevado número de réplicas necessárias, no caso do Q/U ($5f + 1$). Outro problema do protocolo Q/U é a contenção verificada numa situação de escritas concorrentes. Neste seguimento foi proposto o protocolo HQ [CMLRS06], que consiste num híbrido entre os dois anteriormente

mencionados. É um protocolo que faz uso das duas aproximações, quórum e máquinas de estado replicadas, seguindo o modelo de $3f + 1$ réplicas de forma a tolerar $f$ falhas, garantindo ótima resiliência.

O protocolo HQ, na ausência de contenção, faz uso de um novo protocolo bizantino baseado em quórum, que é mais leve e onde as leituras necessitam apenas de uma ronda de comunicação e as escritas duas (entre o cliente e as réplicas). Neste protocolo, o quórum é formado por $2f + 1$ réplicas. Quando se está perante um momento de contenção, é usado o algoritmo PBFT, de forma a ordenar eficientemente os pedidos e a resolver conflitos relacionados com operações concorrentes. No entanto, é usada uma nova implementação do PBFT que, ao contrário da anterior, está desenvolvida para escalar de forma eficiente com o aumento do número de falhas, $f$.

Como mencionado acima, as escritas são efetuadas em duas fases. Na primeira fase, *write-1*, o objetivo é obter uma *timestamp* que determina a ordem desta operação em relação às restantes. Uma vez obtido um quórum de mensagens de resposta, denominado certificado, procede-se à fase seguinte, *write-2*, onde o cliente usa o certificado para convencer as réplicas a executar a operação pretendida no *timestamp* estipulado. Uma vez obtidas as respostas de $2f+1$ réplicas, após processamento do pedido *write-2*, a operação de escrita é concluída. O cenário descrito corresponde a uma ausência de contenção, ou seja, de escritas concorrentes, pois existe ainda a possibilidade de o cliente receber uma mensagem de recusa à *timestamp*, por esta já ter sido atribuída a outro cliente, ou uma mensagem a indicar que a operação de escrita já foi executada.

De acordo com o conjunto de respostas recebidas, o cliente pode assumir diversos comportamentos: (1) Se receber um quórum de confirmações para uma mesma *viewstamp* e *timestamp*, gera um certificado de escrita e prossegue para a fase *write-2*; (2) Se receber um quórum de rejeições para uma mesma *viewstamp*, *timestamp* e valor de *hash*, é porque um outro cliente recebeu um quórum de confirmações e, portanto, estará a executar a fase *write-2*. Para facilitar, na presença de clientes lentos ou com falhas, o cliente gera um certificado na mesma e executa a escrita em atraso, como se do outro se tratasse. É possível obter um certificado de escrita válido através dos certificados de permissão de escrita emitidos ao cliente que obteve acesso, pois estes estão incluídos nas mensagens de rejeição. Após esta operação, repete-se a fase *write-1*; (3) Se foram recebidas confirmações com diferentes valores de *timestamp* ou *viewstamp*, poderá significar que alguma réplica não foi notificada de uma operação de escrita ocorrida anteriormente. São geradas mensagens de escrita com o último certificado recebido e enviadas apenas a essas réplicas mais lentas; (4) Se recebeu imediatamente o resultado do pedido que emitiu, significa que outro cliente efetuou a escrita por este, ou seja, efetuou o passo (2). Neste caso, é usado o certificado proveniente na mensagem para avançar logo para a fase *write-2*; (5) Se recebeu um quórum de respostas concedendo operações diferentes para iguais valores de *timestamp* e *viewstamp*, significa que se está perante uma situação de contenção de escritas. É enviado um pedido de *RESOLVE* para todas as réplicas, por forma a resolver o conflito através do protocolo PBFT. Após receber um quórum destas mensagens, inicia-se o PBFT,

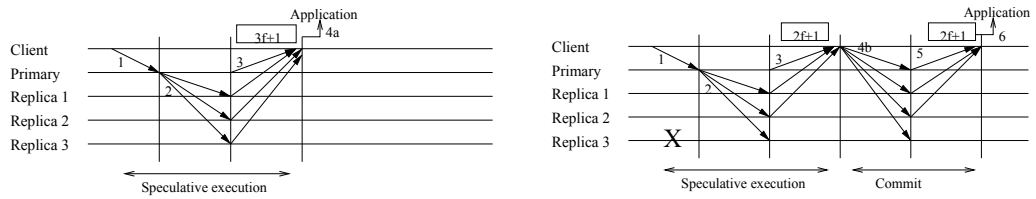
onde todos os pedidos que geraram conflito são executados pela mesma ordem em todas as réplicas corretas. Finalmente, na fase *write-2*, o cliente envia uma mensagem com o certificado obtido na fase *write-1* e aguarda por um quórum de respostas correspondentes válidas. Em caso afirmativo, o resultado é entregue à aplicação, caso contrário, está-se numa situação de contenção e o passo (5) é executado.

No protocolo de leitura o cliente envia um pedido para as réplicas com um *nonce*, para prevenir *replay* da mensagem. Após a receção de um quórum de respostas válidas, retorna o resultado para a aplicação. Se as respostas recebidas traziam diferentes *views-tamps* ou *timestamps*, é enviado um pedido de escrita para as réplicas em questão, seguido do pedido de leitura (caso semelhante a (3) onde poderão haver réplicas mais lentas). O pedido de escrita segue com o último certificado de leitura recebido nas mensagens de resposta.

#### 2.2.1.6 Zyzzyva: speculative byzantine fault tolerance

O protocolo Zyzzyva [KADCW07] usa especulação para reduzir o custo e simplificar o desenho da replicação BFT baseada em máquinas de estado, como o PBFT [CL99]. Ao contrário dos protocolos mencionados, neste, cada réplica executa os pedidos de forma especulativa, sem recorrer a um acordo prévio (como o *three-phase commit protocol*) para definir uma ordem pela qual o pedido deve ser processado. Posto isto, há a possibilidade de diferentes réplicas divergirem nos seus estados e enviarem diferentes respostas ao cliente. No entanto, aplicações do lado do cliente observam abstratamente uma máquina de estados replicada que garante linearização dos pedidos. À semelhança do PBFT, este segue o modelo comum dos protocolos de tolerância a falhas bizantinas, com necessidade de $3f + 1$ réplicas para suportar $f$ falhadas. No entanto, apresenta um menor número de mensagens trocadas entre réplicas para a chegada a um consenso.

O protocolo baseia-se em três sub-protocolos: consenso, alteração de vista e *checkpoint*. No consenso, os pedidos são ordenados para serem executados pelas réplicas. Na alteração de vista, é eleita uma nova réplica primária quando a atual falha e, no *checkpoint* é limitado o estado que tem de ser guardado pelas réplicas (limitando o espaço ocupado pelo histórico das mensagens) e reduzido o custo das alterações de vistas. Na Figura




(a) Execução normal (b) Réplica com falha

Figura 2.9: Execução do protocolo Zyzzyva [KADCW07]

2.9 está representada a execução do sub-protocolo de consenso, tanto para uma execução normal, como para o caso de réplicas falhadas. O cliente envia o pedido para a réplica

primária que por sua vez, após a adição de um número de sequência, envia para as secundárias. Todas executam o pedido e enviam o resultado para o cliente (três primeiros passos da Figura 2.9(a) e 2.9(b)). De seguida, o cliente aguarda (durante um certo período de tempo, estipulado pelo mesmo) por $3f + 1$ respostas mutuamente consistentes. Se recebeu as $3f + 1$ respostas consistentes, dá o pedido como completo, caso contrário, se recebeu entre $2f + 1$ e $3f$, então reúne $2f + 1$ respostas e cria um certificado *commit* para distribuir por todas as réplicas. Estas, ao receberem este certificado (que inclui um histórico das mensagens) podem compara-lo com o seu histórico e verificar se (e onde) ocorreram divergências. Se o cliente voltar a receber pelo menos $2f+1$ respostas de réplicas distintas, considera o pedido como completo e retorna o resultado à aplicação. Existe ainda um último caso onde o cliente pode receber um número de mensagens consistentes inferior a $2f + 1$. Nesse caso, o pedido é enviado novamente a todas as réplicas, que por sua vez vão enviar para a réplica primária. O protocolo segue o seu funcionamento normal, sendo emitido um novo número de sequência para o pedido. Se, ainda assim, o cliente receber respostas com ordenação inconsistente por parte da réplica primária, poderá ser sinal de uma falha associada à mesma, dando-se então inicio ao protocolo de troca de vista, para que seja escolhida outra réplica para assumir o papel de primária.

#### 2.2.1.7 Upright cluster services

No seguimento do estudo das várias alternativas propostas para tolerância a falhas bizantinas surge o UpRight [CKLWADR09], uma biblioteca com o objetivo de fornecer uma alternativa para tolerância a falhas em serviços de *clusters*. Apesar da performance não ser a principal preocupação, verifica-se que esta praticamente não é afetada e ainda assim, no final, obtém-se um sistema mais robusto. O objetivo é possibilitar a fácil adoção de um mecanismo de tolerância a falhas bizantinas a um conjunto de sistemas, garantindo alta disponibilidade e funcionamento correto de todo o sistema. No desenvolvimento desta biblioteca, foi também tido em conta a preservação das propriedades que caracterizam outros sistemas de tolerância a falhas bizantinas já desenvolvidos. Nesse sentido, os autores procuraram manter um bom nível de performance, um baixo custo de replicação e uma elevada robustez. Acima de tudo, contornar os custos de construir todo o sistema de raiz, minimizando a intrusão ao código das aplicações. A arquitetura do sistema encontra-se representada na Figura 2.10. É composta por um cliente, *Application Client*, um servidor, *Application Server* e a biblioteca UpRight. Cada cliente/servidor possui a aplicação localmente, sendo que os acessos são efetuados de forma local através do UpRight *shim* e do componente *glue*. O UpRight *shim* trata das comunicações com os restantes componentes do sistema, enquanto que o *glue* permite estabelecer uma ponte entre o código da aplicação e a interface do UpRight *shim*. Para a execução do pedido de um cliente são necessários os módulos de *request quorum* (RQ), ordem (*order*) e execução, implementados em cada servidor. Nesta biblioteca, do lado dos servidores, é garantido que os mesmos vêm a mesma sequência de pedidos, mantendo o estado consistente e, do

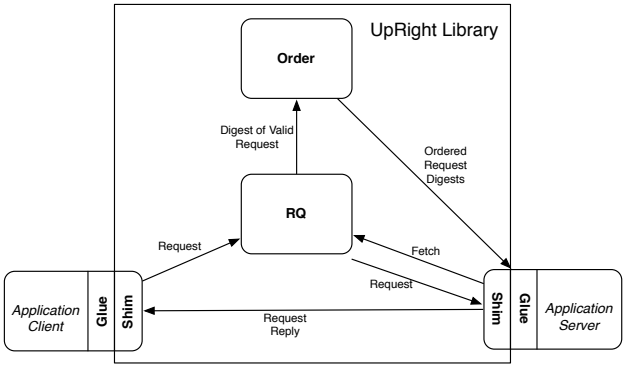



Figura 2.10: Arquitetura UpRight [CKLWADR09]

lado dos clientes, as respostas são vistas pela mesma ordem e no mesmo estado.

Os pedidos efetuados pelos clientes são enviados para o RQ, ou seja, é enviada uma cópia para todos os nós que constituem o quórum. O módulo RQ, após verificação do pedido, nomeadamente, integridade e autenticidade (MAC), encaminha um *digest* do mesmo para o módulo de ordenação, mantendo o pedido completo em sua posse e enviando-o também ao módulo de execução. Ao transformar o pedido do cliente num quórum de pedidos, garante-se que o módulo de ordenação chega a conclusões consistentes. Neste, é produzida uma sequência ordenada dos vários pedidos, através do *digest* dos mesmos. Cada nó do módulo de ordenação irá receber tantas mensagens quanto o número de elementos do quórum. Verifica a consistência do valor obtido e, de forma individual e determinista, propõe uma ordenação para os vários pedidos. Após selecionado o pedido a efetuar, resultante da ordenação, é enviado um *digest* do mesmo para o módulo de execução. Após a receção do *digest*, notifica o quórum que o processo de ordenação foi concluído. O pedido completo referente ao *digest* foi enviado por RQ para o módulo de execução e é entregue à aplicação para ser executado. A resposta é posteriormente enviada diretamente ao cliente. Por fim, quando um nó de execução (módulo de execução) receber uma lista de *digest* ordenados vai processar os pedidos pela ordem estipulada.

De forma a garantir que as réplicas se encontram sempre no mesmo estado, é utilizado, à semelhança do Zyzzyva [KADCW07], o conceito de *checkpoint*. O componente UpRight *shim* contacta periodicamente a aplicação do servidor, para que esta faça um armazenamento persistente do *checkpoint* correspondente ao seu estado. É calculado um *hash* que identifica o *checkpoint* e que pode ser usado para comparar os estados das várias réplicas. Se uma réplica não estiver em concordância, o componente *shim* da mesma comunica com os *shim* das restantes de modo a obter o *checkpoint* mais recente. Reinicia a aplicação do servidor nesse estado e executa a sequencia de operações nele descrito, após o *checkpoint*, de maneira a que a réplica fique no estado corrente.

#### 2.2.1.8 BFT-SMaRt

A biblioteca BFT-SMaRt[Bft] desenvolvida, em parte, por um dos autores do DepSky [BCQAS11], encontra-se disponível em Java e implementa um protocolo de replicação baseado em máquinas de estado, assim como alguns dos protocolos falados anteriormente nesta secção. Foi desenhada de forma a tolerar falhas bizantinas mantendo, ao mesmo tempo, uma elevada eficiência. As propriedades de replicação com base em máquinas de estado são mantidas, assumindo que todas as operações são deterministas, bem como todas as réplicas constituintes do sistema começam no mesmo estado. Esta biblioteca assegura que todas as réplicas executam a mesma sequência de operações. Embora não haja informação disponível de exatamente quais as trocas de mensagens que se verificam entre as várias réplicas e o cliente, existe sempre, à semelhança de outros protocolos do género, a presença de uma réplica coordenadora intitulada de líder.

Acima de tudo, esta biblioteca foi desenhada para oferecer tolerância a falhas bizantinas, ou seja, tolerância face a comportamentos inesperados por parte de alguma réplica, sejam estes *bugs* no próprio software ou até ataques por parte de adversários maliciosos que possam obter controlo sobre as réplicas de forma a alterar o seu comportamento. Em adição, esta biblioteca está desenhada de forma a tolerar ataques do tipo *denial of service*[9] (DoS).

Assume, tal como os protocolos baseados em máquinas de estado mencionados acima, o modelo de $3f + 1$ réplicas, onde $f$ podem ser defeituosas/maliciosas. Por fim, é disponibilizado como uma biblioteca Java, com interface cliente/servidor com métodos para a invocação ordenada, não ordenada e assíncrona de pedidos.

### 2.2.2 RAID e *erasure codes*

Até aqui foram estudadas várias abordagens no que diz respeito ao correto funcionamento do sistema perante a ocorrência de falhas bizantinas. O sistema foi visto como um conjunto de nós servidores que são responsáveis pelo processamento dos pedidos, por parte dos clientes, e os dados são mantidos de forma replicada ao longo dos mesmos. Como se verificou, a problemática consiste em manter a consistência entre os vários objetos replicados, bem como o estado dos mesmos, sem que a performance seja significativamente afetada. Nesta secção pretende-se introduzir o conceito de RAID [CLGKP93], nomeadamente o RAID 5 e RAID 6 e de que forma estes poderiam ser aplicados ao sistema a desenvolver.

O RAID consiste em associar um conjunto de dispositivos de armazenamento (regra geral, discos rígidos) a um único dispositivo lógico, ou seja, é criado um disco lógico através da utilização conjunta de dois ou mais discos, como se de um só se tratasse. Esta forma de armazenamento trás benefícios a nível de segurança dos dados, devido à redundância, e ao nível de performance. Existem dois conceitos importantes, que são o de

[9] http://searchsoftwarequality.techtarget.com/definition/denial-of-service acedido a 16-04-2012

*striping* e *redundancy*. *Striping* consiste em distribuir os dados pelos vários discos, dando a ideia de um único disco bastante rápido e de elevada capacidade. O aumento da performance é notório, principalmente em escritas, pois os discos são acedidos em paralelo e não é introduzida redundância. Já o conceito de *redundancy* introduz redundância a nível dos dados armazenados. A performance poderá ser inferior, pois uma escrita implica a introdução de redundância que tanto pode ser a nível de replicação dos dados por inteiro ou o uso de *erasure codes*[10]. Posto isto, existem várias formas de organizar os discos em RAID, com diferentes combinações que favorecem uma ou outra propriedade (*striping* e *redundancy*).

O RAID 5 faz uso de bits de paridade, introduzindo redundância nos dados, de modo a permitir que os mesmos possam ser recuperados face a uma falha num dos discos. O que distingue o RAID 5 do RAID 4 é o facto de a informação acerca da paridade dos dados se encontrar distribuída ao longo dos vários discos que constituem o *array* (Figura 2.11[11]), em vez de armazenada num disco dedicado. Esta distribuição permite oferecer um maior desempenho relativamente ao RAID 3 e RAID 4 (estes têm um único disco dedicado à paridade), bem como capacidade para tolerância a falhas, não introduzindo assim um ponto único de falha. Os dados de paridade são distribuídos pelos vários discos, permitindo assim que todos participem nas operações de escrita. Comparativamente aos mencionados acima, este modo oferece a melhor performance a nível de leituras e a nível de escritas de grandes volumes de dados.

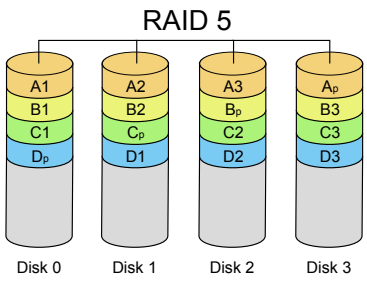



Figura 2.11: *Array* de discos em RAID 5

O RAID 6 (Figura 2.12[12]) é, em muito, semelhante ao RAID 5. A grande diferença está no uso do dobro dos bits de paridade, permitindo assim que dois discos do *array* possam falhar simultaneamente e os dados possam ser recuperados na mesma. Como desvantagens está a necessidade de reservar um maior volume de armazenamento para a paridade, bem como a maior lentidão verificada nas escritas.

Tanto o RAID 5 como o RAID 6 poderiam ser implementados num sistema como o

[10] http://www.networkcomputing.com/deduplication/229500204 acedido a 03-02-2012
[11] http://upload.wikimedia.org/wikipedia/commons/6/64/RAID_5.svg acedido a 06-03-2012
[12] http://upload.wikimedia.org/wikipedia/commons/7/70/RAID_6.svg acedido a 06-03-2012

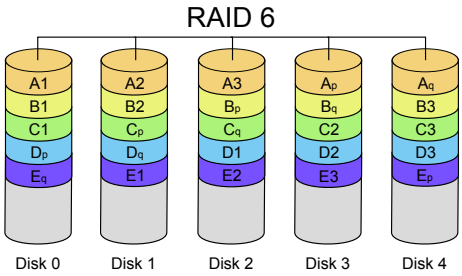



Figura 2.12: *Array* de discos em RAID 6

que se pretende desenvolver, assumindo as *clouds* de armazenamento como sendo discos rígidos. A grande vantagem está no uso de *erasure codes*. O RAID 6, que usa códigos *Reed–Solomon* [Ree], garante a integridade/recuperação dos dados caso dois discos falhem ao mesmo tempo. A utilização de RAID 6 com as técnicas de *erasure codes* associadas pode representar uma mais valia no sentido em que a recuperação dos dados não pressupõe a falha de um disco apenas.

### 2.2.3 Discussão

Para suporte de tolerância a falhas bizantinas em ambientes assíncronos com consequente recuperação do estado normal do sistema, várias abordagens foram revistas. Foram estudadas soluções baseadas em replicação bizantina e soluções de armazenamento com interface RAID (RAID 5) ou baseadas em soluções com *erasure codes* (RAID 6).

No caso de sistemas de replicação bizantina, apresentaram-se soluções orientadas a máquinas de estados, com uma réplica primária coordenadora, como PBFT [CL99], Zyzzyva [KADCW07] e BFT-SMaRt[Bft]. São ainda discutidas soluções baseadas em quórum bizantino, como o Query/Update [AEMGGRW05] e o Upright [CKLWADR09], bem como soluções híbridas como HQ replication [CMLRS06].

O algoritmo Paxos por si só seria insuficiente para lidar com o problema da tolerância a falhas bizantinas. Foi apresentado como uma solução interessante no contexto e introdução a esta problemática tendo em conta que representa, em parte, o protocolo usado nos algoritmos de tolerância a falhas bizantinas, nomeadamente, o consenso e ordenação das operações efetuadas, bem como a replicação dos dados.

A biblioteca BFT-SMaRt apresenta-se como uma solução promissora a integrar no sistema a desenvolver, tendo em conta os objetivos para a qual foi desenhada e desenvolvida. É de fácil integração e permite ultrapassar os obstáculos que o sistema a desenvolver poderia apresentar a este nível. As razões pela qual seria interessante o uso desta biblioteca são em grande parte idênticas às apresentadas pelos autores no próprio artigo DepSky [BCQAS11]. Por um lado, visto estarmos a lidar com *clouds* cuja funcionalidade remete apenas para o armazenamento, o protocolo não está dependente da execução de código por parte dos servidores. Para além disso, muitos fornecedores de *clouds* não oferecem garantias de consistência idênticas às de um disco, o que pode afetar o correto

funcionamento deste tipo de protocolos. Esta biblioteca prevê tais situações. Por fim, estando esta implementada na linguagem Java, tendo sido desenhada com o uso de *clouds* públicas de armazenamento de dados em vista, e tendo sido aplicada em diversos projectos[13] apresenta-se como uma forte candidata a ser aplicada no sistema a desenvolver, de modo a dotar o mesmo da capacidade de tolerância a falhas bizantinas.

No caso de sistemas orientados para armazenamento em RAID (RAID 5 ou RAID 6), tendo como benefício o uso de *erasure codes*, é possível diminuir o espaço de armazenamento necessário para garantir uma recuperação completa dos dados face a falhas. As soluções de armazenamento fiável com soluções RAID ou com uma perspetiva mais genérica de nível *middleware*, baseado em protocolos bizantinos, possuem características diversas. As primeiras permitem disponibilizar uma interface de escrita/leitura de dados com semântica equivalente às interfaces de escrita e leitura em *arrays* de discos. As soluções de *middleware* baseadas em sistemas bizantinos, são mais flexíveis para conceção de uma camada intermédia de acesso a dados armazenados em múltiplas *clouds* de armazenamento (do tipo "key-value store"). No entanto, é possível implementar uma interface de escrita e leitura de blocos com base numa especificação RAID 5 ou RAID 6, no topo de um sistema de *middleware* baseado em acordos bizantinos. Tal permite que diferentes *clouds* de armazenamento sejam vistas como discos de um *array* de discos.

## 2.3 Gestão confiável de dados

### 2.3.1 Controlo sobre os dados mantidos na *cloud*

Em [CGJSSMM09] são descritos vários problemas e preocupações relativamente ao uso de uma *cloud* gerida por terceiros. A falta de controlo sobre os dados do lado do cliente leva ainda muitas empresas e utilizadores individuais a não confiarem os seus dados mais sensíveis a este tipo de serviços. Assim, acaba por não ser aproveitado todo o potencial da *cloud*, pois os mesmos apenas confiam uma parte dos seus dados, menos importantes. As preocupações apresentadas pelos clientes podem ser categorizadas da seguinte forma:

**Segurança tradicional.** Intrusões ao nível da rede e do próprio computador. Vulnerabilidades ao nível do fornecedor da *cloud* que podem facilitar ataques do tipo *cross-site scripting*[14] (XSS), *SQL injection*[15], *phishing*, ou mesmo questões relacionadas com a comunicação segura entre o cliente e a *cloud*, como a autenticação e autorização.

**Disponibilidade.** Problemas como a existência de um único ponto de falha, a percentagem de tempo que o sistema se mantém disponível e a integridade dos ficheiros armazenados, todos eles abordados ao longo deste relatório.

[13] http://code.google.com/p/bft-smart/wiki/UsedInAndBy acedido a 16-04-2012
[14] https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) acedido a 03-02-2012
[15] https://www.owasp.org/index.php/SQL_Injection acedido a 03-02-2012

**Controlo dos dados por terceiros,** com consequente transparência dos mesmos. Garantia de que os dados removidos são de facto removidos da *cloud*; a possibilidade de auditar os dados; obrigações contratuais incertas que negam responsabilidades por parte do fornecedor da *cloud* na perda ou corrupção de dados; espionagem por parte do fornecedor; este poder subcontratar outros serviços sobre os quais temos pouco ou nenhum controlo e a questão do *data lock-in* [CGJSSMM09].

Existem ainda novos problemas que surgem com o aparecimento do *cloud computing*, tais como, invasão de privacidade com o surgimento de bases de dados centralizadas e de grandes dimensões (para efeitos de *data mining*), pois o armazenamento dos dados é cada vez mais fácil e barato. Estas podem conter bastante informação sensível e privada e podem ser alvo de ataques. Outro problema diz respeito à autorização do acesso aos dados, tendo como exemplo o Facebook, que lida com dados de diversos tipos (sensíveis ou não) e que, por sua vez, contém aplicações externas, não verificadas, que lidam com esses mesmos dados.

Por fim, as soluções passam por proteger os dados por completo, e não apenas do exterior. Estes devem ser auto descritivos, ou seja, independentes do ambiente onde se encontram, têm de estar cifrados e possuir uma política de usabilidade associada aos mesmos (de forma a construir um ambiente seguro e de confiança, através de virtualização, para a sua gestão). Relativamente aos dados cifrados, é preciso ter em atenção não limitar o seu uso para operações especificas, exceto se for esse o objetivo. Finalmente, é necessário o uso de uma entidade de confiança do lado do servidor para auditar os dados, garantindo a não violação das propriedades de segurança.

### 2.3.2 EHR

No intuito de armazenar registos de saúde de pacientes num ambiente de *cloud* assegurado por terceiros, em [NGSN10] os autores propõem técnicas que garantem a segurança e privacidade desses dados. O objetivo é obter um sistema que permita que os dados dos pacientes sejam partilhados pelos vários serviços de saúde (hospitais, laboratórios, farmácias, etc), de forma flexível, dinâmica e escalável, mantendo a confidencialidade e privacidade. Desta forma, o fornecedor da *cloud* não consegue inferir nada acerca dos dados armazenados. A estratégia utilizada é através de criptografia com base em atributos, ou seja, um sistema criptográfico baseado em chaves públicas onde cada chave tem associado um conjunto de atributos, sendo que cada *ciphertext* tem associado a si uma política de acesso. A chave secreta, $s$, de um utilizador, pode decifrar um dado *ciphertext*, $c$, apenas se o conjunto de atributos associado a $s$ satisfizer as políticas de acesso de $c$.

Os dados dos pacientes estão guardados sob a forma de ficheiros cifrados e existe uma tabela com entradas correspondentes aos vários ficheiros. Cada entrada contém metadados (cifrados também) associados ao ficheiro (descrição, nome pelo qual foi guardado na *cloud* e chave simétrica para cifrar os dados), políticas de acesso (em *plaintext*) e um índice de pesquisa com várias palavras chave associadas ao ficheiro. Um utilizador

que satisfaça as políticas de acesso a um determinado ficheiro, pode decifrar os metadados associados e encontrar a localização do mesmo, bem como a chave simétrica para o decifrar. Uma pesquisa é sempre feita sobre dados cifrados através de *Secure Channel Free Public-Key Encryption with Keyword Search* (PEKS) [NGSN10]. A gestão das chaves privadas e a autenticação está a cargo de uma entidade de confiança.

### 2.3.3 The HP Time Vault Service

Lidar com informação confidencial que apenas deve ser exposta num dado momento no tempo é o problema abordado pelos autores em [MHS03]. Para esse efeito propõem um sistema baseado em *Identifier-based Encryption* (IBE), onde a informação permanece confidencial durante um certo período de tempo, até à data em que é suposto ser revelada. O ideal seria disponibilizar a informação, ainda confidencial, de forma a que se possa ter acesso à mesma mais tarde. Esta abordagem evita o elevado tráfego que seria gerado na rede se o conteúdo fosse disponibilizado para todos os intervenientes apenas na altura em que é suposto ser revelado. Quando comparado com o uso de RSA verificou-se que, para o efeito, o esquema IBE oferece maior simplicidade e eficiência.

Uma chave criptográfica IBE consiste numa *string*, que representa um qualquer conjunto de caracteres ou *bytes*, e num valor público. Neste sistema, estas chaves são emitidas por uma entidade de confiança e a geração de uma chave de decifra pode ser adiada, ou seja, pode ser gerada muito depois da chave de cifra.

Partindo de um cenário onde *Alice* quer enviar uma mensagem para *Bob*, mas quer que esta esteja disponível apenas a 1 de Janeiro de 2004, às 12:00 horas, os seguintes passos são efetuados: (1) *Alice* cifra o documento com uma chave criada a partir da *string* "GMT200401011200" e do valor público fornecido pela entidade de confiança. O documento cifrado é enviado para *Bob*, juntamente com a chave "GMT200401011200". (2) A entidade de confiança está constantemente a produzir chaves de decifra, de acordo com um intervalo de tempo previamente estabelecido. Por exemplo, se estivermos a 8 de Agosto de 2002 e forem 13 horas e 15 minutos e o intervalo estabelecido (granularidade) for de gerar uma chave de hora em hora, então a próxima chave gerada será a correspondente a "GMT200208081400". A chave que *Bob* necessita será, portanto, gerada pela entidade de confiança apenas a 1 de Janeiro de 2004 às 12:00 horas. (3) À medida que são geradas, as chaves são enviadas para um sistema de distribuição que armazena, indexa e pública pares <IBE chave de cifra, IBE chave de decifra>. O cliente obtém a respetiva chave de decifra a partir deste.

### 2.3.4 Silverline

Em [PKZ11] os autores descrevem um sistema cujo foco é a identificação e encriptação dos dados mais sensíveis, e que não afetam ou impõem limites nas funcionalidade da aplicação. Descrevem um conjunto de ferramentas, Silverline, que efetua essa identificação e atribui chaves de encriptação para conjuntos de dados específicos de modo a

minimizar a complexidade da gestão das chaves e fornecer transparência ao utilizador no acesso aos dados, mantendo este nível de segurança. É um sistema focado, essencialmente, para a execução de aplicações no ambiente da *cloud*.

Os dados possíveis de cifrar, aqueles que nunca são interpretados pela aplicação, são chamados de funcionalmente cifráveis, pois não limitam as funcionalidades da mesma. Estes são identificados descartando todos os dados usados em computações na *cloud*. São também repartidos por subconjuntos, associados a um determinado grupo de utilizadores e com uma chave criptográfica única (criptografia simétrica). As chaves são mantidas do lado dos clientes, preservando a confidencialidade dos dados. Os grupos de utilizadores são constituídos através de uma análise, por parte do sistema, sobre as operações que cada utilizador efetua e, portanto, tem acesso. Cada utilizador acaba com um conjunto de chaves, de acordo com os grupos onde se insere, de modo a poder aceder aos dados que é suposto e, ao mesmo tempo, ter acesso restrito aos dados que não deve. Não são explicitamente utilizados mecanismos de verificação de integridade dos dados, embora sejam mencionadas técnicas de como adaptar o sistema para tal. Existe ainda um componente de monitorização que lida com as alterações que a base de dados pode sofrer ao longo do tempo, nomeadamente, em relação ao acesso dos utilizadores, ou seja, se estes têm acesso aos dados supostos, ou se deixam de ter acesso a determinado conteúdo.

### 2.3.5 iDataGuard

O sistema iDataGuard [JGMSV08] consiste num *middleware* seguro que permite o armazenamento de dados, por parte dos utilizadores, em múltiplas *clouds* de armazenamento de dados heterogéneas, como as mencionadas na secção 2.1.3. Assim como no sistema a desenvolver, as *clouds* são vistas como não confiáveis e não seguras e o sistema funciona como intermediário entre o utilizador e as mesmas. Este *middleware* pretende preservar características de confidencialidade, bem como integridade dos dados armazenados pelas várias *clouds* públicas de armazenamento, através da criação de um sistema de ficheiros seguro, sobre as várias *clouds* heterogéneas. A nível arquitetural, o sistema é composto por três componentes principais: máquina cliente, *middleware* iDataGuard e fornecedores de *clouds* de armazenamento de dados. O *middleware* corre na máquina cliente. Todos os dados armazenados são cifrados ao nível do *middleware* e só depois enviados para as *clouds* públicas de armazenamento, de forma que estas nada possam inferir acerca dos mesmos. Os dados são vistos como objetos, através do modelo de dados abstrato que é suficientemente genérico para poder englobar diversas *clouds* existentes. O mapeamento entre as operações especificas dos provedores e o modelo abstrato do sistema é feito através de *service adapters*.

De forma a garantir confidencialidade dos dados armazenados, cada objeto (ficheiro ou diretoria) é encriptado com uma chave única. Essa é gerada através de uma palavra passe (do utilizador) e do nome do objeto. A integridade dos objetos é verificada através do cálculo de um HMAC. Este é calculado, não só através do conteúdo do objeto, mas

também de um identificador da versão do mesmo. Assumindo o uso de, pelo menos, dois fornecedores de *cloud*, os números da versão dos objetos são guardados sempre numa *cloud* diferente daquela onde o objeto está armazenado (por questões de segurança).

Para além das operações típicas, como criar uma diretoria ou ler um ficheiro, existe ainda a possibilidade de fazer pesquisas por uma dada palavra chave. Para este efeito, é usado um índice duplo, CryptInd++. Um dos índices contém uma lista invertida de todas as palavras chave únicas e um conjunto de documentos associados às mesmas. O outro contém uma lista invertida de várias sub-palavras associadas às palavras chave existentes. Por exemplo, para a palavra "Secure", no segundo índice, poderiam estar associadas as seguintes entradas: < "sec", {Secure} >, < "ecu", {Secure} >, < "cur", {Secure} > e < "ure",{Secure} >.

### 2.3.6 DepSky

Por último, em [BCQAS11] é descrito o sistema DepSky, cujos principais objetivos são no sentido de melhorar a disponibilidade, integridade e confidencialidade dos dados armazenados, através de uma *cloud*-de-*clouds*. Os dados encontram-se replicados, codificados e encriptados por várias *clouds* de armazenamento conhecidas. Como um todo, o DepSky é visto como uma *cloud* segura de armazenamento, composta por diversas *clouds* onde os dados são armazenados de forma segura e confidencial. Problemas como a disponibilidade são endereçados através da replicação dos dados por várias *clouds*, de modo a serem recuperáveis após indisponibilidade de alguma. A perda ou corrupção de dados é contornada com o uso de algoritmos de tolerância a falhas bizantinas (baseados em quórum) e a privacidade dos mesmos é mantida através de esquemas criptográficos, *secret sharing* e *erasure codes*. O problema do *vendor lock-in* é contornado tanto ao nível dos dados estarem armazenados em diversas *clouds*, como devido à utilização de *erasure codes* de modo a guardar apenas uma fração dos dados em cada *cloud*.

Os algoritmos usados no DepSky estão numa biblioteca do lado da aplicação cliente. O protocolo bizantino usado requer $3f + 1$ *clouds* de armazenamento para tolerar $f$ *clouds* falhadas, tal como alguns dos protocolos estudados na secção 2.2.1. O quórum bizantino é obtido por $n - f$, com $n = 3f + 1$. A confidencialidade é obtida através da partilha de um segredo, $s$. Existe uma entidade denominada *dealer* que distribui $s$ por $n$ *clouds*, sendo que cada uma recebe apenas parte de $s$. São precisos $f + 1 <= n$ partes diferentes de $s$ para recuperar o segredo e este não consegue ser descoberto com apenas $f$ ou menos partes disponíveis. Existem dois protocolos, o DEPSKY-A que assegura disponibilidade e integridade e o DEPSKY-CA que fornece, adicionalmente, confidencialidade.

### 2.3.7 Discussão

Nesta secção foi feito um estudo de vários sistemas, diversificados e com objetivos diferentes, mas relacionados com diferentes dimensões dos objetivos e contribuições da

dissertação. Os sistemas estudados também se diferenciam em relação às soluções específicas que propõem. Foram enumerados vários aspetos de segurança que levam à fraca adoção de soluções *cloud* para gestão de conteúdo sensível. Consequentemente, foram estudados vários sistemas que visam contornar problemas do mundo real e que beneficiariam bastante das propriedades da *cloud*, caso houvessem garantias de privacidade e segurança dos dados. Tratam-se de abordagens diferenciadas que apresentam soluções de manutenção de dados privados em repositório de dados públicos, como é o caso do registo de dados médicos e partilha de informação confidencial.

Sistemas como o Depsky [BCQAS11] ou o iDataguard [JGMSV08] apresentam-se como soluções relacionadas com o objetivo da presente dissertação. Têm como propósito usar *clouds* de armazenamento de dados na rede Internet, fornecidas e geridas por terceiros, acrescentando garantias de privacidade e integridade aos utilizadores, estabelecendo uma base de confiança gerida por estes. Estes sistemas possuem conectores para as diferentes *clouds*, que encapsulam uma interface única de acesso para escrita, leitura e pesquisa e que acionam as interfaces de cada uma das *clouds*, replicando os dados de forma cifrada. Tal é necessário, uma vez que as soluções dos fornecedores de *clouds* de armazenamento não garantem a priori confidencialidade e privacidade dos dados. Esse aspeto é deixado para as aplicações que escrevem ou leem os dados diretamente de cada uma das *clouds*.

## 2.4 Segurança dos dados

### 2.4.1 Integridade

Para além da confidencialidade dos dados mantidos na *cloud*, é necessário preservar a sua integridade. Desta forma, é necessário garantir que os mesmos não são alterados enquanto permanecem na *cloud*, quer por parte de atacantes, como do próprio provedor. Em [HPPT08], os autores propõem o recurso a uma estrutura de dados (*skip list*) autenticada e armazenada, onde constam os valores de *hash* associados aos dados armazenados numa *cloud* pública. Por sua vez, é mantido, do lado do cliente, um valor de *hash* de tamanho fixo, associado a um *digest* dessa estrutura. A arquitetura deste sistema consiste em três componentes principais: a *cloud* de armazenamento, o servidor de autenticação, que contém a estrutura com a informação de integridade da informação armazenada e o cliente, que efetua pedidos ou atualiza a informação existente em ambos. Os autores garantem um algoritmo eficiente, com complexidade logarítmica $O(\log(n))$, para a verificação da integridade de um ficheiro, assumindo a existência de $n$ ficheiros armazenados. É assumido que tanto o servidor de autenticação como a *cloud* não são de confiança, daí o cliente ter sempre guardado um valor de tamanho fixo que corresponde a um *hash* da estrutura de dados que verifica a integridade de todo o conteúdo armazenado, enquanto que o servidor de autenticação guarda os valores de *hash* para os vários ficheiros nessa estrutura de dados. A segurança de todo o sistema está diretamente relacionada com a

função usada para o cálculo desses valores.

O sistema HAIL [BJO09] foi criado com o objetivo de fornecer um serviço de elevada disponibilidade com garantia de integridade dos ficheiros. Permite que um conjunto de servidores prove ao cliente que os dados armazenados estão intactos e podem ser acedidos. A redundância é estabelecida pelos vários fornecedores de *cloud*. É assegurada recuperação perante falhas bizantinas com a integridade dos ficheiros verificada do lado do cliente, evitando comunicação entre servidores. O grande objetivo é garantir resistência face a um adversário móvel, que pode corromper todos os $n$ servidores, ao longo do tempo de vida do sistema. Contudo, apenas um subconjunto $b$ de $n$ pode ser controlado pelo atacante num dado instante (denominado época). Em cada época, o cliente efetua um conjunto de testes de integridade sobre um ficheiro $F$ e, ao detetar irregularidades, $F$ pode ser reconstruido a partir da redundância dos servidores em estado correto. No entanto, não basta verificar a irregularidade entre réplicas. É necessário uma prova de integridade sob as mesmas e tal é obtido através de PORs (*proofs of retrievability*) que são provas sobre os dados que garantem que $F$ é recuperável face a uma perda ou corrupção. São apresentadas várias técnicas para obter uma prova de integridade dos ficheiros armazenados, tendo como medida os vários instantes, épocas, que o sistema atravessa. Destaca-se o uso de técnicas de *erasure codes* e algoritmos de autenticação MAC para as provas de integridade e consequente recuperação. Todos os ficheiros são considerados estáticos ao longo do tempo.

### 2.4.2 Confidencialidade e Privacidade

Um grande problema que se põe relativamente ao armazenamento dos dados de forma confidencial diz respeito à forma como é efetuada uma pesquisa sobre os mesmos. O ideal seria esconder, para além dos dados, toda e qualquer operação que o utilizador fizesse sobre os mesmos, nomeadamente no caso de se tratar de uma pesquisa. Existem várias abordagens a ter para a implementação de uma pesquisa segura, como por exemplo em [JGMSV08], onde o mecanismo de pesquisa consiste na indexação de várias palavras chave associadas aos vários ficheiros. No entanto, esta abordagem fica bastante limitada em relação às palavras chave disponíveis e, portanto, permite uma pesquisa pouco exaustiva. Neste seguimento foi estudado o CryptDB [PRZB11], onde, vendo o sistema como uma base de dados SQL, as *queries* são executadas sobre um conjunto de dados cifrados, através de um conjunto de esquemas de encriptação sobre SQL existentes. O desafio consiste em minimizar a quantidade de informação confidencial revelada, mantendo a possibilidade de executar uma grande variedade de *queries* e minimizar a quantidade de dados que poderiam ser expostos na eventualidade de um ataque comprometedor ao sistema. A arquitetura do CryptDB consiste essencialmente num *proxy* que cifra/decifra os dados e operadores da *query*, e só depois os encaminha para o sistema de gestão de bases de dados. Com base no CryptDB, é possível identificar vários esquemas distintos a abordar em diversas *queries* SQL, resumidamente falados em baixo

e que podem ser vistos com mais detalhe em [PRZB11]. Dependendo das operações que se poderão vir a efetuar sobre uma dada coluna, de uma dada tabela, é escolhido o esquema que mais se adequa para cifrar os dados. A ideia é cifrar os dados em camadas, ou seja, os valores de uma coluna são cifrados sucessivamente com diferentes algoritmos, por ordem crescente do grau de segurança que estes fornecem, de modo a permitir flexibilidade nas operações a efetuar sobre os mesmos.

**Random (RND).** Fornece o nível de segurança mais elevado, principalmente contra ataques do tipo *chosen-plaintext*, pois é probabilístico, ou seja, dois valores iguais de *plaintext* podem dar origem a diferentes *ciphertext*. Geralmente é usado um algoritmo como o AES ou o Blowfish em modo CBC com recurso a um vetor de inicialização.

**Deterministic (DET).** Modo ligeiramente mais fraco a nível de segurança pois, para um mesmo *plaintext*, é sempre gerado um mesmo *ciphertext*. É necessário quando se quer efetuar verificações de igualdade de valores ao nível do SELECT, ou mesmo operações de junção, GROUP BY, COUNT, DISTINCT, etc. Recorre-se na mesma a um algoritmo como o AES ou o Blowfish mas, no entanto, o algoritmo de cifra de blocos usado é o CMC.

**Order-preserving encryption (OPE).** Permite que a ordem dos dados seja preservada, sem que os mesmos sejam revelados. Desta maneira, é possível efetuar *queries* que envolvam um determinado intervalo de tuplos, bem como operações de ORDER BY, MIN, MAX, SORT, etc. É um esquema mais fraco a nível de segurança, quando comparado com o DET, devido a revelar a ordenação.

**Join (JOIN and OPE-JOIN).** Aqui é usado um esquema diferente do DET, onde são usadas chaves diferentes para colunas diferentes. Através do uso de um esquema como este, é também possível realizar todas as operações de DET, mas não permite que o servidor detete valores idênticos entre colunas distintas. A variação OPE-JOIN permite junções ordenadas sobre tabelas.

**Word search (SEARCH).** Este esquema oferece suporte a pesquisas através do operador LIKE do SQL. O nível de segurança obtido é quase idêntico ao RND, não sendo revelado ao sistema de gestão de bases de dados se um valor se repete em colunas distintas. Não permite que se façam pesquisas complexas, essencialmente são pesquisas por palavras inteiras.

**Homomorphic encryption (HOM).** Este esquema permite a realização de pesquisas e operações sob os dados cifrados, através de um algoritmo de cifras homomórficas. Este é abordado com mais detalhe seguidamente, em secção própria.

### 2.4.3 Sistemas Criptográficos Homomórficos

Um sistema criptográfico homomórfico (com mecanismos de cifra e decifra) permite que, dado um conjunto de operações para manipulação de dados não cifrados, seja possível definir um conjunto de operações associadas que operam da mesma forma mas em texto cifrado. Permite, por exemplo, que um dado utilizador, não conhecedor da informação confidencial, efetue operações sob a mesma, sem que esta seja revelada. O uso de cifras homomórficas é bastante promissor quando transposto para um ambiente de *cloud computing*, no sentido em que permite salvaguardar os dados (de forma cifrada) e, ainda assim, efetuar operações simples sobre os mesmos, sem nunca arriscar a sua exposição.

Para este efeito, existem esquemas criptográficos completamente homomórficos, que permitem um vasto conjunto de operações sob os dados cifrados, e esquemas parcialmente homomórficos [NLV11]. Estes últimos permitem apenas um número restrito de operações sobre os dados, no entanto, conseguem ter uma performance superior em relação a esquemas completamente homomórficos. Este ultimo ponto é onde os esquemas completamente homomórficos pecam ainda atualmente, pois não existe uma solução suficientemente eficiente. Aplicado, por exemplo, a um sistema de manutenção de registos de saúde, como o EHR [HPPT08], falado anteriormente, podem existir dispositivos dedicados que efetuam estatísticas sobre os dados dos pacientes sem terem necessariamente de conhecer o valor real dos mesmos.

No CryptDB [PRZB11] este esquema (HOM) permite que o servidor execute operações diretamente sobre os dados cifrados, sendo o resultado final decifrado no *proxy*. Provou-se eficiente para certas operações, mais especificas. O algoritmo utilizado é o Paillier cryptosystem e é possível encontrar uma implementação em Java do mesmo em [Pai]. É um algoritmo assimétrico probabilístico e com recurso a chaves criptográficas públicas. Uma propriedade importante, que lhe confere a característica de cifra homomórfica, é que dada a chave pública, $k$, e o *ciphertext* de duas mensagens, $m1$ e $m2$, é possível obter o *ciphertext* de $m1 + m2$. Da mesma forma verifica-se $HOM_k(m1) \cdot HOM_k(m2) = HOM_k(m1+m2)$. No CryptDB estas propriedades são úteis, por exemplo, na implementação de agregações por SUM onde o *proxy* substitui a operação de SUM por uma que vai efetuar uma multiplicação de uma coluna cifrada sobre este esquema (HOM).

### 2.4.4 Discussão

Nesta secção foram abordadas técnicas que permitem garantir a integridade dos ficheiros armazenados na *cloud*. Quer seja por recurso a uma aplicação cliente, independente da própria *cloud*, onde é feita essa verificação a partir de uma estrutura de dados especial, *skip list* [HPPT08], quer através de PORs, que são provas de integridade fornecidas pela *cloud* sobre os dados (HAIL [BJO09]). Finalmente, foi abordado o conceito de confidencialidade dos dados ao nível das pesquisas efetuadas, com o objetivo de esconder, não só os dados, como todas as operações efetuadas sobre os mesmos. Garante-se assim que um

atacante ou a *cloud* não possam inferir nada acerca das operações que o utilizador efetuou. Foram abordadas metodologias de como esconder as *queries* efetuadas (assumindo uma base de dados SQL, como é o caso da abordagem do sistema CryptDB [PRZB11]) bem como o papel de mecanismos baseados em cifras homomórficas. No contexto dos objetivos desta dissertação, é particularmente interessante um esquema de pesquisa idêntico ao do CryptDB, visto o tipo de metadados pretendidos não estarem associados a um tipo de operação específico. O que se pretende, sobretudo, é um algoritmo com características homomórficas que permita efetuar pesquisas (e não necessariamente operações matemáticas) sobre um conjunto de metadados, adaptando-os a repositórios de dados do tipo chave-valor.

## 2.5 Análise crítica face aos objectivos da dissertação

### 2.5.1 Sumário e discussão sobre o trabalho relacionado apresentado

Face aos objetivos da dissertação, não existe um sistema, de entre os vários estudados, que englobe na totalidade as funcionalidades pretendidas. É particularmente notório o facto de que não se encontram sistemas para lidarem com a problemática de pesquisas em tempo real, sobre dados privados mantidos cifrados em múltiplas *clouds* de armazenamento, sem perigo de exposição de chaves ou segredos criptográficos na infraestrutura de cada provedor. O sistema que mais se aproxima da perspetiva da dissertação é o DepSky. Deve notar-se que o sistema Silverline apresenta uma solução que engloba e tem como principal foco *clouds* onde se pode tirar partido do poder computacional para a execução de aplicações. Tanto o Depsky como o iDataGuard não utilizam, porém, suporte de confidencialidade como o que se perspetiva para a dissertação, que pretende integrar mecanismos e cifras homomórficas para suportar pesquisas genéricas sobre dados cifrados. Para além disso, o iDataGuard é visto como um *middleware* a ser executado numa máquina cliente ou mesmo num *proxy*, não sendo perspetivado para vir a ser implementado num ambiente distribuído, como uma *cloud* privada, daí que não seja abordada a problemática da consistência entre réplicas através de protocolos de consenso bizantino.

Os sistemas Bigtable, Cassandra, Dynamo e Riak deixam os aspetos da segurança e privacidade para as aplicações que utilizam essas soluções. No entanto, são muito relevantes para os objetivos da dissertação. Tratam-se de soluções inspiradoras para armazenamento fiável de dados, com base numa estrutura de repositório distribuído do tipo chave-valor e com elevada disponibilidade, sendo portanto bastante promissores para desempenhar o papel de índice dos ficheiros armazenados no sistema. No sistema a desenvolver, o armazenamento dos dados será repartido pelas várias *clouds* de armazenamento usadas, tais como Amazon S3, Luna Cloud, Dropbox e Google Cloud Storage. Deste modo, seguir-se-à uma abordagem idêntica ao DepSky, onde os ficheiros são divididos em blocos que podem corresponder a fragmentos de dados, sendo estes repartidos

pelas várias *clouds*, através de um sistema de indexação e recuperação confiável. O sistema de indexação pressupõe que tenha que ser persistente. Para o efeito, o sistema a conceber precisará de usar uma solução para repositório intermédio de dados, capaz de manter o índice, bem como um sub-sistema de *cache* primária de dados acedidos. Uma hipótese poderia ser suportar este sub-sistema numa base de dados convencional. Mas outra solução mais interessante, como mencionado anteriormente, será usar um sistema como o Riak, beneficiando de uma implementação já existente e garantindo a priori que o sistema a desenvolver possa ser implementado como uma solução suportada numa *cloud* privada, num ambiente institucional com características de grande escala, estando o índice replicado por diferentes nós.

Para replicação dos blocos de dados, e de modo a contornar o problema da fiabilidade e disponibilidade, o sistema utilizará replicação bizantina, de modo a tolerar falhas ou intrusões nas *clouds* de armazenamento que estejam a ser usadas. Para o efeito iremos reutilizar componentes da biblioteca BFT-SMaRt. Tal permite introduzir tolerância a falhas ou intrusões em $f$ *clouds* desde que se garanta a existência de $3f + 1$ réplicas de blocos de dados distribuídos em $3f + 1$ *clouds*. Pensa-se integrar esta implementação já existente, que se revela mais adequada para reutilização no contexto da dissertação, seguindo uma abordagem também usada pelo sistema DepSky.

Por fim, o suporte de integridade deverá seguir uma abordagem semelhante à apresentada em [HPPT08], sendo integrada no sistema de indexação utilizado para a gestão da localização dos vários blocos (ou fragmentos) de dados pelas *clouds*. A confidencialidade dos dados e a capacidade de pesquisa sob os mesmos será endereçada através de uma implementação idêntica à utilizada no CryptDB, sendo uma adaptação Java do código C++ disponibilizado pelos autores. Mais propriamente, será usada uma implementação de um esquema de pesquisa linear, adaptando este mecanismo para cifra de metadados associados aos ficheiros a serem mantidos cifrados (para serem escritos, lidos e pesquisados em repositórios de dados do tipo chave-valor), sem perigo de exposição das chaves ou parâmetros criptográficos.

### 2.5.2 Problemática específica sobre modelo de controlo de concorrência em *clouds* de armazenamento

Como foi inicialmente apresentado na introdução, a dissertação privilegia propor uma solução confiável, controlada pelos utilizadores, que permita a adoção de múltiplas *clouds* de armazenamento, tal como disponibilizadas por provedores Internet. O sistema T-Stratus (enquanto solução proposta) atuará como um conjunto de serviços *middleware*, vistos como um serviço "proxy" (ou no caso mais geral uma *cloud* "proxy") intermediando o acesso dos utilizadores às nuvens de armazenamento.

Tendo em conta esta direção, as múltiplas *clouds* (combinadas como solução de *cloud* de *clouds*) serão usadas como repositórios para manutenção de dados (ou fragmentos de dados), cifrados e replicados, tendo em vista estabelecer um modelo de proteção que

garante as propriedades de segurança pretendidas bem como proteção contra falhas ou ataques bizantinos desencadeados independentemente sobre cada uma das *clouds* que sejam utilizadas. A utilização combinada de diferentes *clouds* (de acordo com as soluções disponibilizadas) permite, desde logo, beneficiar da sua heterogeneidade ou diversidade tecnológica como fator importante de mitigação de falhas ou de risco de ataques por intrusão.

Torna-se no entanto importante discutir o aspeto particular relacionado com os modelos de concorrência disponibilizados pelas atuais *clouds* de armazenamento e como esta problemática se relaciona com a questão do suporte do controlo de concorrência ao nível da solução a propor.

As soluções de *clouds* de armazenamento hoje existentes permitem estabelecer uma base de suporte à solução pretendida que disponibiliza, na maior parte dos casos, um modelo de controlo de concorrência baseado numa semântica do tipo regular, isto é, garantindo ao utilizador final um modelo de controlo de concorrência do tipo "single-writer – multi-reader". Deve dizer-se no entanto que nem todas as *clouds* concretizam este modelo. Por razões de escalabilidade e elasticidade, em alguns casos, existem soluções que suportam outros modelos de concorrência de acesso, como é o caso da solução Amazon S3, que oferece consistência do tipo eventual, ou de semântica do tipo "read-after-write"[16]. Nestes casos, a base de controlo de concorrência torna-se heterogénea, pelo que a adaptação a diferentes *clouds* heterogéneas pode ter que lidar com este problema. A dissertação não pretende lidar com este tipo de problemática, tendo como orientação de implementação um sistema para utilização sem controlo de concorrência em escrita (que deverá ser concretizado pelo suporte específico das aplicações que utilizam o sistema T-Stratus). Deste modo, o sistema *middleware* preconiza uma utilização em que cada utilizador (ou cada aplicação externa) é executado numa instância de execução da solução *middleware* ao nível dos serviços T-Stratus. Não obstante esta ser a orientação base para implementação e avaliação da solução proposta, no capítulo 3 (secção 3.3.1) avança-se uma discussão de generalização da solução para lidar com este tipo de problemática.

Para todos os efeitos, a solução pretendida orienta-se para uma utilização com controlo de concorrência do tipo "single writer – multiple readers", sendo usado enquanto sistema distribuído assíncrono, em que o sistema *middleware* T-Stratus é usado como cliente de múltiplas *clouds*. O acesso às *clouds* para leitura pode falhar arbitrariamente (já que as *clouds* individualmente podem falhar, funcionar com intermitência ou revelarem um comportamento bizantino em relação aos dados acedidos). Contudo, considera-se que as escritas só falharão por falhas (ou ataques) às *clouds* que impliquem a sua paragem (de acordo com um modelo "fail-stop"). Nos objetivos da dissertação não se consideram mecanismos ao nível da solução *middleware* para lidarem com problemas de falhas arbitrárias durante os processos de escrita de dados (ou fragmentos de dados) nas *clouds*, embora estas falhas possam ser toleradas desde que um número adequado de réplicas

[16] http://aws.amazon.com/s3/faqs/#What_data_consistency_model_does_Amazon_S3_employ acedido e verificado a 25-03-2013

possa ter sido escrito com sucesso, tendo em conta a resiliência necessária (ou seja, $f + 1$ réplicas escritas com sucesso, para $n$ *clouds*, em que $f$ *clouds* falhem arbitrariamente). O não se pretender lidar com cenários de recuperação de falhas bizantinas durante operações de escrita do lado das *clouds*, resulta do facto de se pretender garantir que a solução esteja adequada à utilização de *clouds* de armazenamento, isto é, *clouds* onde não se executa código. Para lidar com este tipo de problemas seria necessário executar código do lado das mesmas, o que não é considerado por se entender que esse código poderia ser objeto de ataques de intrusão bizantinos, de qualquer modo.

# 3

# Modelo e arquitetura do sistema

Tal como se introduziu anteriormente, o sistema T-Stratus, foi concebido como sistema *middleware*, permitindo assegurar um conjunto de serviços intermediários entre utilizadores individuais (ou suas aplicações) e múltiplas *clouds* de armazenamento. A ideia é criar um ambiente confiável de *cloud* de *clouds* de armazenamento, com base em soluções de provedores destes serviços na Internet. A solução pretende aproveitar as vantagens dessas soluções, permitindo a manutenção e replicação de dados privados em múltiplos provedores Internet, como solução de *outsourcing* de dados, mas permitindo o controlo dos utilizadores da base de confiança que assegura as propriedades de segurança, privacidade e confiabilidade.

Tendo em vista um modelo de computação confiável de *cloud* de *clouds*, os serviços de *middleware* fornecidos pelo sistema T-Stratus foram ainda concebidos de forma a poder utilizar a solução com flexibilidade, nomeadamente: variante local, variante *proxy* e variante em *cloud*.

**Variante Local.** Nesta variante, o sistema é visto como suporte de *middleware* local, executando num computador de um utilizador e intermediando de forma transparente o acesso das aplicações a múltiplas *clouds* de armazenamento utilizadas como repositórios de dados;

**Variante Proxy.** Nesta variante o sistema é visto como serviço de *proxy*, executando remotamente num servidor confiável, estando este instalado na rede local do utilizador final, e permitindo intermediar o acesso de aplicações de um utilizador às múltiplas *clouds* de armazenamento utilizadas. Esta arquitetura surge como uma generalização da anterior, permitindo por exemplo suportar aplicações baseadas

em dispositivos com poucos recursos de computação, onde apenas executam aplicações finais;

**Variante em *Cloud*.** Neste caso a arquitetura do sistema é formada por um conjunto de servidores que replicam os serviços *middleware* numa arquitetura "core", formando um ambiente distribuído de um grupo de servidores de computação e gestão de dados de controlo. Os servidores podem ser disponibilizados numa rede corporativa de grande escala, sendo cada um deles um sistema confiável. Nesta última ideia, o sistema toma a forma de um *cluster* de servidores, atuando com vantagens de escala e permitindo acesso ubíquo, podendo os utilizadores interagir com qualquer um dos servidores. Estes formam assim uma *cloud* consistente de controlo de intermediação e acesso transparente às múltiplas *clouds* de armazenamento utilizadas. É nesta última visão que a designação T-Stratus aparece associada a uma "nuvem baixa" de controlo, que permite a gestão confiável dos dados privados armazenados de forma transparente nas nuvens de *outsourcing* (ou "nuvens altas"), correspondendo estas a repositórios de dados fragmentados e cifrados, distribuídos segundo um modelo de replicação bizantina pelos diferentes provedores Internet utilizados.

Nas próximas secções deste capítulo descreve-se o modelo de sistema, incluindo o modelo de segurança e discute-se a arquitetura de serviços *middleware* da seguinte forma: na secção 3.1 o sistema será apresentado na variante *proxy*, executando num servidor confiável acessível na rede local do cliente. Na secção 3.2, a anterior visão será então generalizada para o modelo em *cloud* discutindo-se os componentes do sistema que suportam essa generalização.

## 3.1 Solução Proxy

### 3.1.1 Modelo de sistema, arquitetura e segurança

O modelo do sistema na solução *proxy* tem como base uma arquitetura cliente servidor, sendo o servidor concebido como um conjunto de serviços *middleware* (formando um servidor T-Stratus) que intermedeiam uma aplicação de um utilizador e as várias *clouds* de armazenamento que possam estar a ser usadas. O sistema executa num servidor confiável, considerado seguro e livre de intrusões. A interação entre o cliente e este servidor *proxy* faz-se através de uma API que disponibiliza uma interface externa para suportar escritas, leituras ou pesquisas de objetos. Esta API, apresentada mais à frente, tem uma especificação semelhante à de uma *cloud* de armazenamento de objetos, fornecendo um nível de serviço externo semelhante ao de um repositório do tipo "chave-valor". O acesso dos clientes a este serviço é materializado com base em serviços web, de acordo com uma especificação suportada em interações seguras que podem ser baseadas em SSL (ou TLS), por configuração específica.

As *clouds* de armazenamento na Internet são consideradas suscetíveis de exibirem falhas arbitrárias ou ataques por intrusão, eventualmente exploradas com base em vulnerabilidades de sistemas de software usados pelos diversos provedores. Consideram-se ainda potenciais ataques devidos a comportamento indevido de administradores de sistemas ou outro pessoal afeto a cada provedor. Considera-se que o modelo de falhas e o modelo de ataques por intrusão é bizantino mas independente, isto é, não existem ataques "concentrados" em mais do que uma *cloud* de armazenamento, sendo circunscritos de forma independente a cada uma das *clouds* utilizadas.

Tendo em conta um modelo de adversário e de falhas bizantino, a utilização de múltiplas *clouds* de armazenamento permitirá assegurar as condições de resiliência necessárias para garantir fiabilidade dos dados e a sua disponibilidade. As *clouds* de armazenamento podem ser heterogéneas do ponto de vista das soluções de Hardware e Software. Esta heterogeneidade associada à independência do modelo de falhas e intrusões bizantinos promove um ambiente com natural diversidade, que assim minimiza o impacto de falhas ou ataques simultâneos desencadeados contra mais do que uma *cloud*.

Considera-se que poderão existir falhas de comunicação ou ataques às comunicações, entre o servidor *proxy* T-Stratus e cada uma das *clouds* de armazenamento utilizadas. Estes ataques podem ser desencadeados por adversários atuando segundo modelos do tipo "homem-no-meio". Considera-se que estes ataques têm como alvo as propriedades de autenticação, confidencialidade e integridade de dados trocados nas operações de escrita, leitura ou pesquisa de dados, entre o servidor T-Stratus e as *clouds* de armazenamento utilizadas.

### 3.1.2 Requisitos e revisão de objetivos

O objetivo da presente dissertação foi conceber, implementar e testar um sistema de intermediação (concebido como um serviço *middleware*) que permitisse a utilizadores individuais, ou utilizadores de uma organização, utilizar de forma confiável as soluções de *clouds* de armazenamento de dados privados, disponibilizadas como serviços de provedores Internet.

Os critérios de confiabilidade suportados pela solução (numa visão de *dependable system*) prendem-se com os seguintes requisitos fundamentais que se conjugam no problema a resolver:

- Garantia de confidencialidade e privacidade dos dados;
- Garantia de integridade permanente dos dados;
- Garantia de fiabilidade, com tolerância a falhas ou intrusões ao nível da infraestrutura dos provedores;
- Garantia de independência da solução de fiabilidade e disponibilidade dos dados por utilização transparente de múltiplos provedores, com replicação de fragmentos de dados protegidos por *clouds* independentes, sob exclusivo controlo de recuperação dos dados por parte dos utilizadores;

- Acesso aos dados através de um índice replicado, descentralizado, que pode ser executado num local independente do *middleware*;
- Salvaguarda de condições de independência dos utilizadores face a mudanças de regimes de prestação de serviço decididas unilateralmente pelos provedores, seja por condições de políticas de manutenção (ou SLAs – *Service Level Agreement Policies*) seja por práticas que criem eventuais dificuldades de acesso, disponibilidade ou recuperação dos dados (ou *Vendor-Lock-In Practices*);
- Independência do utilizador na garantia de preservação da privacidade de dados alguma vez armazenados em qualquer *cloud* de armazenamento na Internet, mesmo após a conclusão de contratação e utilização do serviço de um provedor;
- Preservação permanente de privacidade dos dados, mesmo em condições de pesquisa e acesso aos mesmos em tempo real, e não apenas em regimes de gestão e obtenção de salvaguarda de dados (ou seja, utilização da *cloud* apenas como repositórios de *backup*);
- Garantia de manutenção dos requisitos de segurança e fiabilidade através do controlo autónomo e independente, por parte dos utilizadores, da base de confiança que sustenta todos os anteriores requisitos.
- Suporte aos anteriores critérios e garantias, preservando todas as vantagens inerentes às soluções atuais de *internet cloud-storage solutions*. Para tal a solução tem que ser avaliada através da usabilidade de soluções reais e pela análise de métricas de eficiência (latência e taxas de transferência) no acesso aos dados, comparando com a utilização direta desse tipo de soluções, sem as referidas condições de confiabilidade, nos moldes atuais.

A conjugação dos anteriores requisitos numa única solução é um problema complexo, cujas vertentes não são hoje endereçadas, tal quanto se sabe, por nenhuma solução da investigação recente em segurança para computação e gestão de dados em *clouds* de computação e armazenamento na Internet. Nas próximas subsecções apresenta-se a abordagem da arquitetura do sistema, seguido de uma descrição dos componentes, bem como da API disponibilizada.

### 3.1.3 Arquitetura de software de referência

A arquitetura de software de referencia é concebida como um conjunto de serviços e componentes de software formando uma solução de sistema *middleware* disponibilizando externamente um sistema de armazenamento confiável e seguro de dados ou objetos privados que serão na verdade armazenados nas múltiplas *clouds* de armazenamento de diferentes provedores Internet, intermediadas pela solução. Os objetos manipulados no sistema *middleware* correspondem a dados escritos, lidos ou pesquisados pelos utilizadores. Para efeitos de concretização, estes objetos são ficheiros genéricos, tendo como ponto de vista aplicações externas de gestão convencional de ficheiros. Nesse sentido,

no contexto da atual dissertação, o enfoque e contribuição foi dirigido de forma a conceber uma especificarão da API de utilização externa do sistema, de modo a poder ser usada como serviço de ficheiros remotos. Desta forma, as aplicações finais (escritas em Java), podem adotar a solução como um repositório transparente de armazenamento de ficheiros, vendo o mesmo como um repositório do tipo chave-valor, com uma interface de serviço WEB (*web services*). Esta interface possui expressividade para escrita, leitura e pesquisa de ficheiros (pesquisa segura baseada em multi-palavras), sendo estas funcionalidades descritas em baixo, na secção 3.1.6. A ideia foi a de adotar uma interface que instância a mesma expressividade que a interface hoje oferecida por uma solução como a Google Cloud Storage ou a Amazon S3. Tal permite a integração transparente de aplicações suportadas, por exemplo, sobre a atual *cloud* Amazon S3.

A Figura 3.1 descreve o modelo arquitetural do sistema, apresentando os seus principais componentes. Este *middleware* apresenta-se de acordo com um modelo *Web-3-Tier*, dividido em camada de Interface, lógica de processamento interno e conectores de integração de dados que interagem com as *clouds* de armazenamento via interfaces WEB (*web services*). Esta Figura 3.1 representa a arquitetura interna do *middleware*. Na camada de interface, constam os modelos de interação com o sistema, sendo estas as interfaces de alto nível que interagem com a interface (API) da lógica do *middleware*.

Na camada de lógica, temos a API uniforme do *middleware*, que serve de agregador de funcionalidades de todo o sistema. Toda a lógica do sistema é agregada pelo componente TStratus, que trata de efetuar conversões, entre os ficheiros e o modelo de objetos próprio do *middleware*, que atua sobre blocos ou fragmentos de dados de diferentes tamanhos (parametrizáveis) e agrega todos os outros módulos. Os restantes módulos conferem as propriedades de segurança dos dados, integridade, compressão, pesquisa e indexação. Esta camada permite o suporte, gestão e recuperação de blocos de dados cifrados na *cloud*, sem exposição externa de chaves ou segredos criptográficos. A comunicação com a camada de dados é efetuada através do módulo de tolerância a falhas bizantinas, responsável pela escrita e leitura dos dados provenientes dos vários provedores utilizados.

Na camada de integração de dados, temos os conectores para as diferentes *clouds* de armazenamento. Estes garantem suporte de heterogeneidade e disponibilizam um conjunto de operações mais comuns (de escrita e leitura) sobre as *clouds* utilizadas.

Este *middleware*, tal como descrito na Figura 3.1, corresponde a uma instância do sistema na sua variante *proxy*. A visão geral desta variante, ilustrada na Figura 3.2, representa a execução do sistema num servidor confiável, na rede local do utilizador final.

Não obstante, o objetivo final é a generalização da arquitetura anterior, formando um conjunto de servidores confiáveis de computação e gestão de dados. Essa visão corresponde à variante *cloud*, onde várias instâncias do *middleware* descrito formam um ambiente distribuído de larga escala (*cluster* de servidores). Esta variante corresponde à visão T-Stratus, de uma *cloud* para controlo de intermediação e acesso transparente de múltiplas *clouds* de armazenamento de dados de provedores Internet, onde os mesmos se encontram fragmentados, cifrados e distribuídos segundo um modelo de replicação

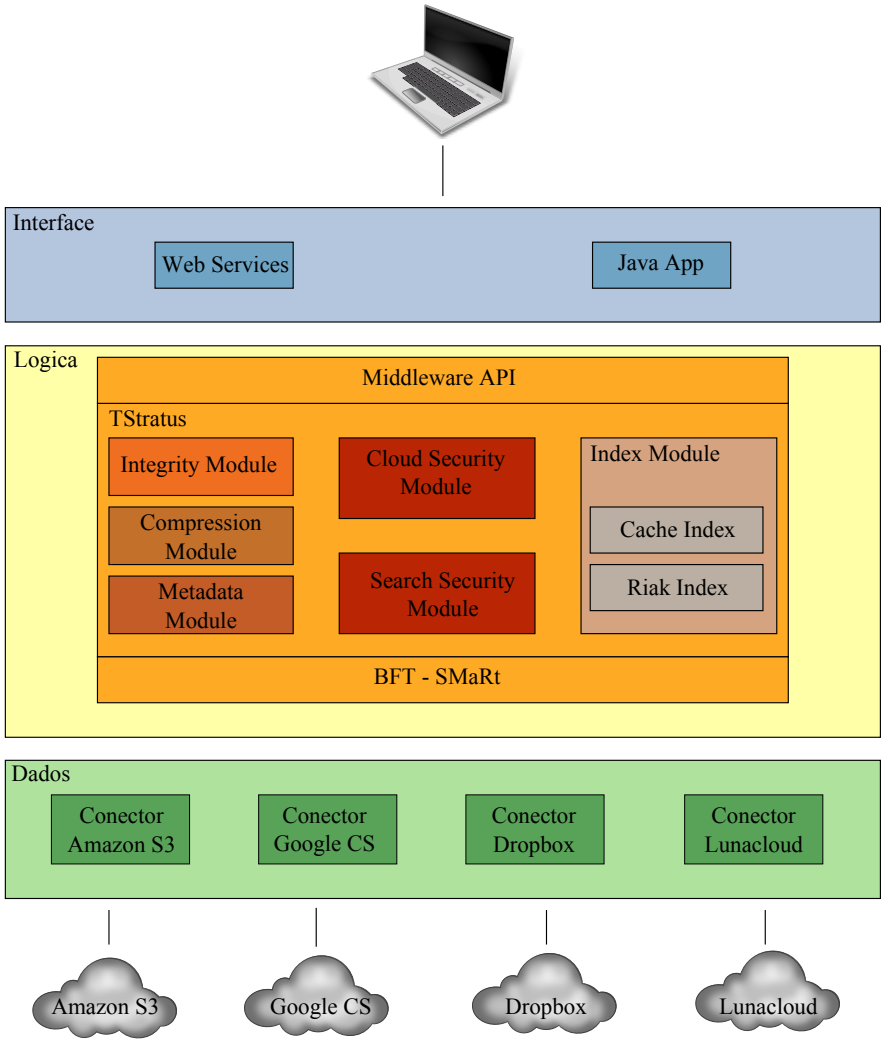



Figura 3.1: Arquitectura interna e componentes do *middleware*

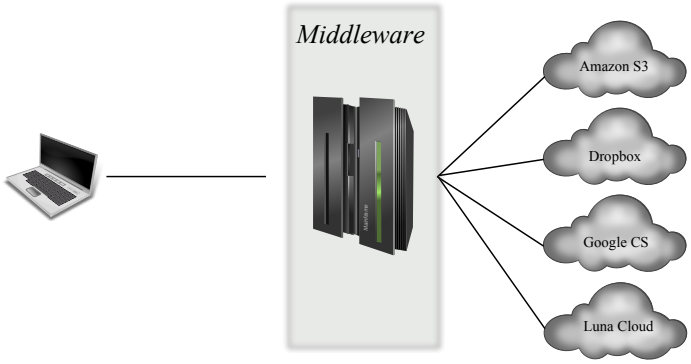



Figura 3.2: Serviço instalado num servidor dedicado, próximo do utilizador

bizantina. Esta generalização é descrita com maior detalhe na Secção 3.2.

### 3.1.4 Componentes e serviços da solução Middleware

Após a breve descrição anterior dos módulos que compõem o sistema, os mesmos são descritos individualmente em baixo, de acordo com as várias contribuições mencionadas anteriormente neste capítulo:

**Web Services -** Através da API fornecida pelo *middleware*, foi desenvolvido um cliente Java que comunica com o mesmo através de serviços REST (*web services*). Desta forma, o *middleware* pode ser executado num *proxy* ou numa *cloud* privada, mantendo-se acessível através de um cliente remoto, em qualquer lugar.

**Aplicação Java Local -** Partindo do pressuposto de um serviço instalado num computador de um utilizador, foi desenvolvido um cliente local, que executa na mesma infraestrutura do *middleware* e invoca serviços diretamente ao mesmo. Numa solução deste tipo, podemos evitar a latência existente na comunicação entre o cliente e o servidor, que pode ser um *bottleneck* considerável, dependendo da distância entre os vários componentes, bem como da capacidade e largura de banda do canal de comunicação. Não obstante, requer uma capacidade de processamento superior por parte do cliente e, para um acesso concorrente por múltiplos utilizadores, a solução mais apelativa é através do acesso remoto. Este cliente local é essencialmente apropriado para a extração de resultados e validação do sistema atual.

**Integrity Module -** Responsável pela geração das provas de integridade sobre os ficheiros alojados nos vários provedores de domínio público. Garante a integridade dos ficheiros armazenados que possam ter sido alterados, quer por falhas na transferência dos mesmos, quer por eventuais ataques de cariz malicioso às *clouds*, que possam pôr em risco (corromper) a informação armazenada. Aqui é endereçada a problemática da garantia de integridade permanente dos dados, pois este módulo permite gerar uma prova de integridade sobre cada ficheiro e fragmento armazenado. Essa prova é armazenada no índice do *middleware* permitindo, mais tarde, a verificação da integridade do ficheiro ao ser recuperado.

**Compression Module -** Este módulo comprime ou descomprime um dado ficheiro. Por um lado, cria uma dificuldade acrescida na tentativa de criptanálise por parte de um atacante, dada a posterior encriptação do ficheiro. Por outro, permite uma redução no tamanho do mesmo, o que, aliado aos modelos de custo dos vários provedores de *clouds* de armazenamento públicos usados, se traduz num custo menor associado ao armazenamento dos mesmos. Não obstante, há que contabilizar que nem todos os ficheiros permitem uma compressão suficientemente significativa que justifique o tempo gasto neste processo, face à consequente redução do custo de armazenamento do mesmo. Através da validação e extração de resultados do sistema, será medido o custo associado às operações deste módulo, sendo tiradas ilações acerca do mesmo em relação ao tempo total de processamento.

**Metadata Module -** Este modulo permite extrair a informação acerca de um ficheiro, a que chamamos de metadados. Esta informação é posteriormente cifrada segundo um esquema que explora técnicas de encriptação homomórfica, oferecendo suporte às pesquisas seguras sobre o conjunto de metadados de um ficheiro. Como o modelo de sistema atual permite a gestão e pesquisa sobre um qualquer conjunto de ficheiros genéricos, os dados a extrair são também comuns a qualquer ficheiro existente. Para tal, tomou-se como ponto de partida a informação genérica armazenada num sistema de ficheiros, num sistema operativo como o Windows. Essa informação fica associada como metadados ao objeto correspondente ao ficheiro armazenado.

**Cloud Security Module -** Efetua as operações criptográficas sobre os dados enviados ou recebidos da *cloud*. Contém o processamento de gestão de chaves e segredos criptográficos necessários à utilização das técnicas criptográficas. Embora numa versão mais complexa e numa visão futura do sistema o Cloud Security possa vir a ser suportado sobre um *tamper-proof module* (ou appliance) de hardware, na presente solução encontra-se totalmente implementado em software. Neste módulo são efetuadas as operações associadas à cifra e decifra dos ficheiros alojados na *cloud*. As pesquisas seguras, por sua vez, não fazem uso do mesmo algoritmo, tendo portanto um módulo dedicado para o efeito, mencionado de seguida. Os vários fragmentos dos ficheiros enviados para a *cloud* são alvo de um conjunto de operações, de entre as quais se destaca a cifra dos mesmos, garantindo uma das propriedades mais importantes do sistema, que é a confidencialidade e privacidade dos dados. Acima de tudo, assegura a independência do utilizador na garantia de preservação da privacidade de dados alguma vez armazenados em qualquer *cloud* de armazenamento na Internet, mesmo após a cessação de contrato e utilização do serviço de um provedor. Para este efeito, é usado um algoritmo de cifra simétrico, pois apresenta uma performance superior em relação aos algoritmos assimétricos existentes.

**Search Security Module -** Aqui são efetuadas as operações que dizem respeito à cifra dos metadados associados aos ficheiros armazenados. É este módulo que, juntamente com os índices, oferece suporte às pesquisas de forma confidencial, interagindo diretamente com o Cache Index e o Riak Index. É responsável pela comparação entre os dados a pesquisar e os pesquisáveis, visto se tratar de um algoritmo não determinista. Este não determinismo, não permite obter o respetivo *plaintext*, a partir de um dado *ciphertext* pois, para um mesmo *plaintext*, o *ciphertext* gerado é diferente. Esta propriedade confere um grau de segurança ainda maior, dificultando qualquer tentativa de criptanálise por parte de terceiros. Trata-se de um esquema que explora técnicas de encriptação homomórfica, face aos dados em claro. Com isto, é possível manter as características de confidencialidade e privacidade dos

dados, mesmo durante uma operação de pesquisa. Garante-se a preservação permanente de privacidade dos dados, mesmo em condições de pesquisa e acesso aos mesmos em tempo real, e não apenas em regimes de gestão e obtenção de salvaguarda de dados. O número de resultados obtidos é parametrizável no sistema e o calculo dos mesmos é baseado num sistema simples (ao contrário de sistemas de pesquisa mais complexos baseados em *rank*). Aqui, é contabilizado o número de vezes que as palavras associadas à pesquisa aparecem no conjunto de metadados de cada ficheiro. A inclusão de um sistema de pesquisa mais complexo encontra-se fora do âmbito da presente dissertação, sendo aqui o principal objetivo o de disponibilizar um sistema seguro de pesquisa sobre o conjunto de metadados.

**Cache Index -** Este módulo diz respeito à cache que é mantida para o acesso rápido dos ficheiros mais requisitados, ou que foram introduzidos no sistema em último lugar, sendo estas as duas filosofias aplicadas na ordenação interna da estrutura utilizada. Esta encontra-se, também, armazenada de forma persistente em disco, de modo a não se perder a informação associada à mesma na eventualidade de um reinicio ao sistema. O acesso à cache permite um relaxamento no tempo total de execução de uma operação de obtenção, remoção ou pesquisa. Não obstante, o seu tamanho é um fator a considerar pois pode ter um crescimento diretamente proporcional ao número de ficheiros no sistema, podendo vir a ter implicações de espaço no servidor onde o mesmo é executado. O número máximo de ficheiros contidos na cache corresponde idealmente a cerca de $20\%$ do número total de ficheiros armazenados no sistema. No sistema atual, o valor é parametrizável.

**Riak Index -** Diz respeito ao índice principal, sendo responsável pela indexação e localização de todos os fragmentos do sistema. Este encontra-se acessível a partir de uma API que implementa um cliente Java para as várias operações disponibilizadas. O (*middleware*) desempenha, entre outros, o papel de cliente para o servidor/*cloud* onde se encontra a ser executado o índice. Através deste componente é possível garantir a indexação e acesso aos dados através de um índice replicado, descentralizado, que pode ser executado num local independente do sistema em si.

**Index Module -** Módulo que agrega as funcionalidades dos dois índices mencionados acima, através de uma interface que permite o acesso às várias operações de forma transparente em relação ao índice usado.

**TStratus -** É neste módulo que se agregam todas as funcionalidades e todos os módulos mencionados anteriormente, de forma a efetuar a sequência de operações necessária para a transformação de um ficheiro no objeto representativo do mesmo no presente sistema. A informação resultante, nomeadamente, os metadados e a distribuição dos vários fragmentos do ficheiro pelas várias *clouds*, é armazenada neste objeto próprio, no índice. O armazenamento é do tipo chave-valor, onde o objeto mencionado é serializável e corresponde ao valor, sendo a chave o nome do ficheiro

em questão. Todo este processo é descrito em baixo, para as diferentes operações disponíveis na API do sistema. Este módulo representa o *middleware* em si e, com a agregação dos restantes, garante a manutenção dos requisitos de segurança e fiabilidade através do controlo autónomo e independente, por parte dos utilizadores, da base de confiança que sustenta todos os requisitos propostos.

**BFT - SMaRt -** Módulo que implementa o protocolo de tolerância a falhas bizantinas e controla todo o armazenamento remoto, bem como os conectores. Este módulo reutiliza uma biblioteca de acordo bizantino para recuperação de blocos replicados em múltiplas *clouds* através de uma API que permite a implementação de um cliente e respetivos servidores bizantinos. O cliente em questão comunica com os quatro servidores utilizados, efetuando pedidos de escrita e leitura às várias *clouds*. Cada um deles é responsável pela escrita e leitura numa das quatro *clouds* de armazenamento públicas utilizadas e o cliente envia os pedidos para o conjunto de servidores bizantinos, que por sua vez vão realizar um algoritmo de consenso e sincronização dos mesmos. É assim garantida a contribuição que prevê a salvaguarda das condições de independência dos utilizadores face a mudanças de regimes de prestação de serviço decididas unilateralmente pelos provedores, quer por condições de políticas de manutenção, quer por práticas que criem eventuais dificuldades de acesso, disponibilidade ou recuperação dos dados.

**Conector Amazon S3 -** Adapta/Mapeia as operações especificas da API da Amazon S3, em operações genéricas conhecidas pelo sistema, através das quais qualquer *cloud* pode ser acedida de forma transparente.

**Conector Google Cloud Storage -** Adapta/Mapeia as operações especificas da API da Google Cloud Storage, em operações genéricas conhecidas pelo sistema, através das quais qualquer *cloud* pode ser acedida de forma transparente.

**Conector Dropbox -** Adapta/Mapeia as operações especificas da API da Dropbox, em operações genéricas conhecidas pelo sistema, através das quais qualquer *cloud* pode ser acedida de forma transparente.

**Conector LunaCloud -** Adapta/Mapeia as operações especificas da API da Lunacloud, que por sua vez faz uso da API da Amazon S3, em operações genéricas conhecidas pelo sistema, através das quais qualquer *cloud* pode ser acedida de forma transparente. A API da Amazon S3 é aproveitada quase na integra por este provedor, bastando apenas mudar o *endpoint* (destino) para o URL correspondente ao da Lunacloud.

Os vários conectores mencionados, juntamente com o protocolo de tolerância a falhas bizantinas, garantem a utilização transparente dos múltiplos provedores, com a replicação dos fragmentos dos dados protegidos por *clouds* independentes, onde o controlo e recuperação dos mesmos está inteiramente a cargo dos utilizadores. Como mencionado

em cima, o sistema T-Stratus utiliza os diversos componentes para suportar operações de leitura, escrita ou pesquisas de objetos (ou ficheiros) nas diferentes *clouds* de armazenamento utilizadas. Ao nível do modulo TStratus propriamente dito faz-se a agregação dos restantes componentes, de forma a efetuar a sequência de operações necessária para a transformação de um objeto ou ficheiro externo num objeto do sistema T-Stratus e vice-versa.

### 3.1.5 Processamento ao nível dos serviços middleware T-Stratus

As cinco operações disponíveis no sistema agregam e fazem uso dos vários módulos de uma forma sequencial e distinta, de forma a realizar as transformações necessárias nos ficheiros armazenados. Para uma melhor perceção dos componentes que intervêm nas várias operações descritas, as cores ilustradas nos diagramas que se seguem correspondem às da Figura 3.1 apresentada anteriormente. A Figura 3.3 apresenta um diagrama com a sequência de transformações aplicadas durante a inserção de um ficheiro no sistema. Inicialmente, o utilizador invoca um pedido de `put` de um ficheiro. O mesmo é lido do disco e enviado para o sistema. Uma vez no sistema, é-lhe extraída a informação dos metadados que o compõem. Estes metadados são, em parte, aqueles que são guardados pelo sistema operativo para qualquer ficheiro, sendo o modelo de sistema o armazenamento de ficheiros genéricos. Fazem parte do conjunto de metadados: o nome do ficheiro; o tipo (ou extensão); a data de criação; o tamanho; a data do último acesso e a data da última modificação. Após a extração destes atributos, os mesmos são cifrados através do modelo implementado no módulo `Search Security`. Após a cifra dos metadados, procede-se à criação do objeto representativo do ficheiro (`CloudObject`) no sistema. Neste, é adicionada a prova de integridade dos dados (*digest*, através do `Integrity Module`) e, para cada fragmento, é realizada uma compressão (`Compression Module`), cifra (`Cloud Security Module`), cálculo do *digest* (`Integrity Module`) e envio do mesmo para o quórum bizantino (`BFT-SMaRt`), onde será enviado para as quatro *clouds*. A informação, nomeadamente, do *digest*, dos metadados e do nome e respetiva prova de integridade de cada fragmento, é armazenada no objeto CloudObject que, por sua vez, é inserido na cache (`Cache Index`) e no índice (`Riak Index`). Não ocorrendo nenhum erro no decorrer do processo, a operação final traduz-se em sucesso por parte da aplicação.

Para a obtenção de um ficheiro, o utilizador realiza um pedido `get` ao sistema. A Figura 3.4 ilustra o conjunto de operações levadas a cabo pelo sistema na sequência de um pedido desta natureza. O utilizador invoca um pedido `get` para um dado ficheiro. O mesmo é conduzido no sistema por uma pesquisa inicial na cache (`Cache Index`), sendo que, em caso de insucesso, essa pesquisa é remetida para o índice (`Riak Index`). Uma vez obtido o objeto correspondente, é extraída a lista de fragmentos do mesmo e, para

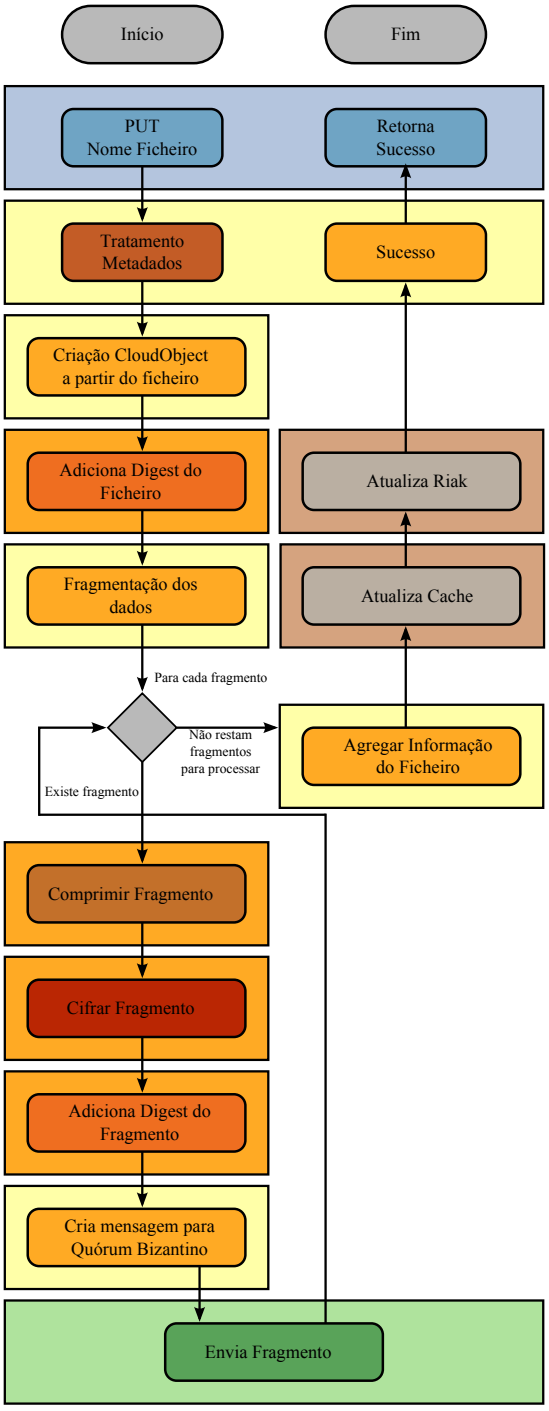



Figura 3.3: Sequência de operações que descrevem um pedido PUT

cada fragmento, é feito um pedido do mesmo ao quórum bizantino (`BFT-SMaRt`), seguido de uma verificação de integridade (`Integrity Module`), decifra (`Cloud Security Module`) e descompressão (`Compression Module`). Uma vez obtidos todos os fragmentos, estes são agregados e é verificada a integridade do resultado final (`Integrity Module`), correspondente aos dados que compõem o ficheiro. Verificada a integridade de

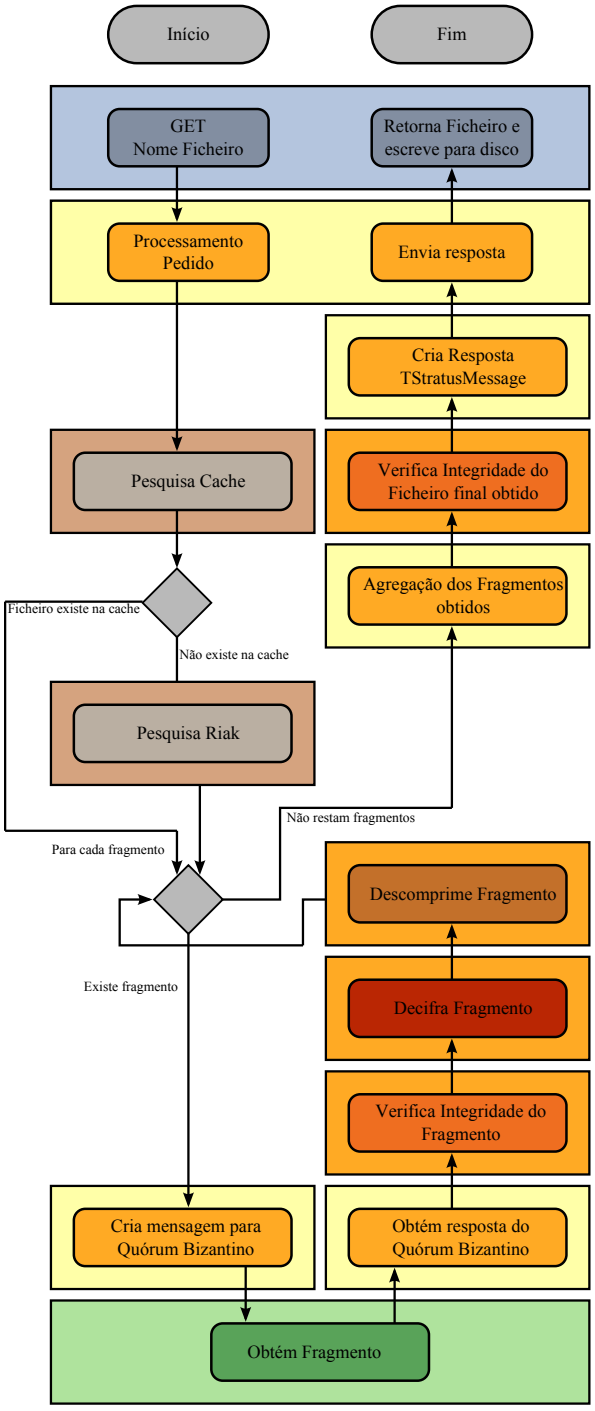



Figura 3.4: Sequência de operações que descrevem um pedido GET

todo o conjunto de dados com sucesso, a resposta é criada e remetida para o utilizador. Este processa o conjunto de dados recebido e escreve o ficheiro obtido para memória persistente.

Na Figura 3.5 encontra-se ilustrado o processo que leva à remoção de um ficheiro do sistema. Uma vez mais, o utilizador invoca o respetivo pedido `remove` para um dado

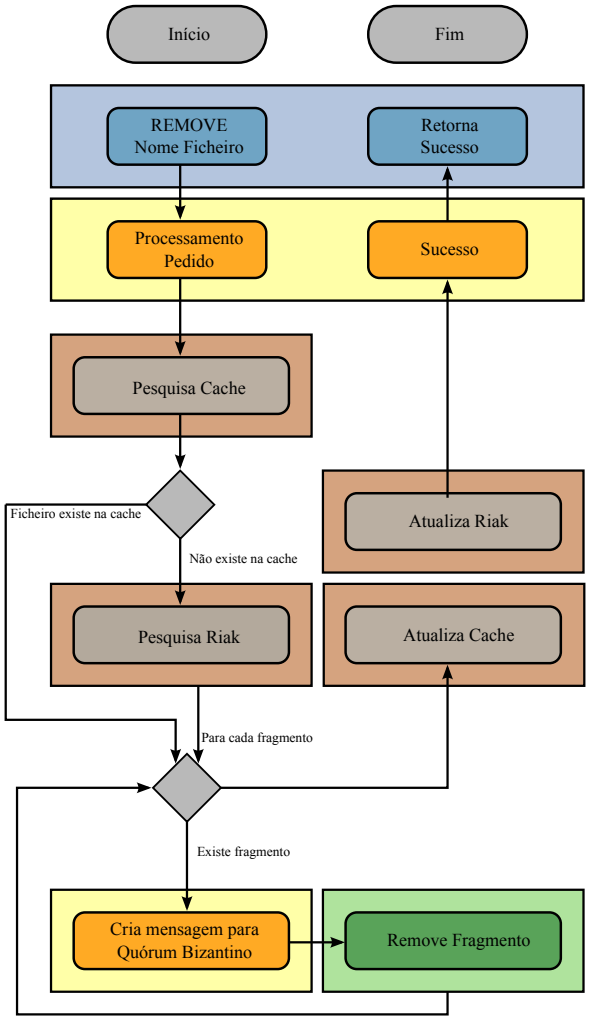



Figura 3.5: Sequência de operações que descrevem um pedido REMOVE

nome de ficheiro. Este é processado pelo sistema, que vai pesquisar pelo ficheiro em questão na cache (`Cache Index`) e, em caso de insucesso, remete a pesquisa para o índice (`Riak Index`). Uma vez retornado o objeto correspondente, obtém a lista dos fragmentos que compõem o ficheiro e, para cada um deles, efetua um pedido de remoção para o quórum bizantino (`BFT-SMaRt`). Uma vez finalizado este processo com sucesso para todos os fragmentos, a cache e o índice são atualizados (`Cache Index` e `Riak Index`) com a remoção do ficheiro em questão. Uma vez mais, não ocorrendo nenhum erro no decorrer do processo, a operação final traduz-se em sucesso por parte da aplicação.

Para finalizar, apresentam-se nas Figuras 3.6 e 3.7 os módulos que intervêm nas operações levadas a cabo durante uma pesquisa realizada sobre o conjunto dos metadados dos vários ficheiros existentes. Os dois tipos de pesquisa diferem no resultado que devolvem. Na primeira, são obtidos apenas os nomes dos ficheiros com maior relevância na pesquisa em questão. Na segunda, são devolvidos os dados dos ficheiros, ou seja, é feito um pedido `get` para cada resultado obtido. A pesquisa é feita sobre todo o conjunto de

dados e é calculado o número de vezes que as palavras pesquisadas ocorrem nos metadados dos ficheiros, sendo esse o valor que atribui a ordenação final, por relevância, da pesquisa.

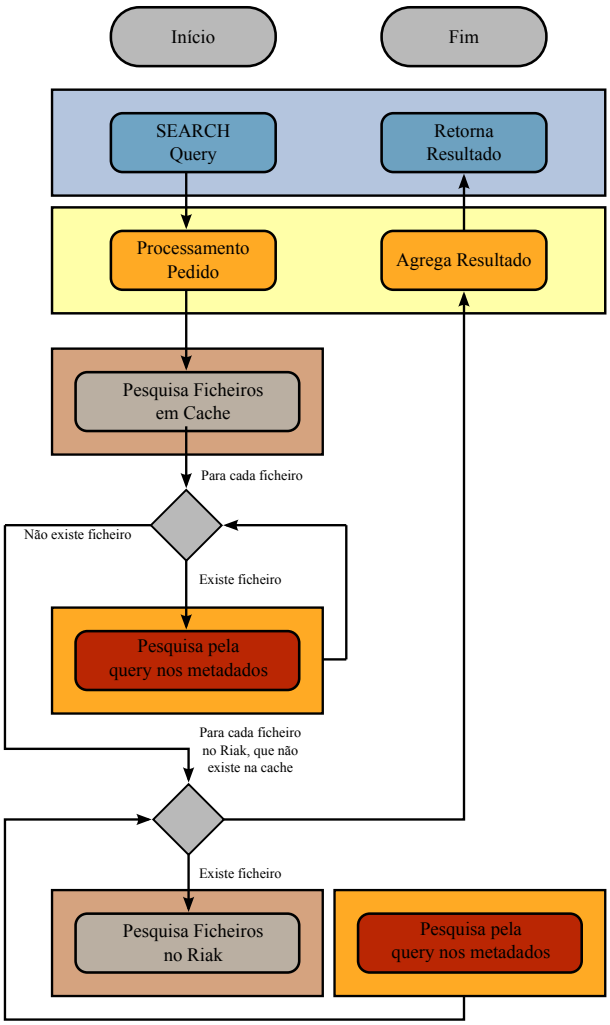



Figura 3.6: Sequência de operações que descrevem um pedido SEARCH

Para esta operação, o utilizador começa por enviar um pedido de `search` ou `search-retrieve` com o conjunto de palavras a pesquisar. Esse pedido é processado pelo sistema, onde os dados da *query* são cifrados (`Search Security Module`). De seguida, a pesquisa é realizada em todos os ficheiros contidos na cache e no índice (`Cache Index`, `Riak Index` e `Search Security Module`), sendo que, no índice, não se irá repetir a pesquisa para os ficheiros já pesquisados na cache. No final, no caso do `search` (Figura 3.6), o resultado consiste no nome dos ficheiros encontrados, do mais relevante para o menos.

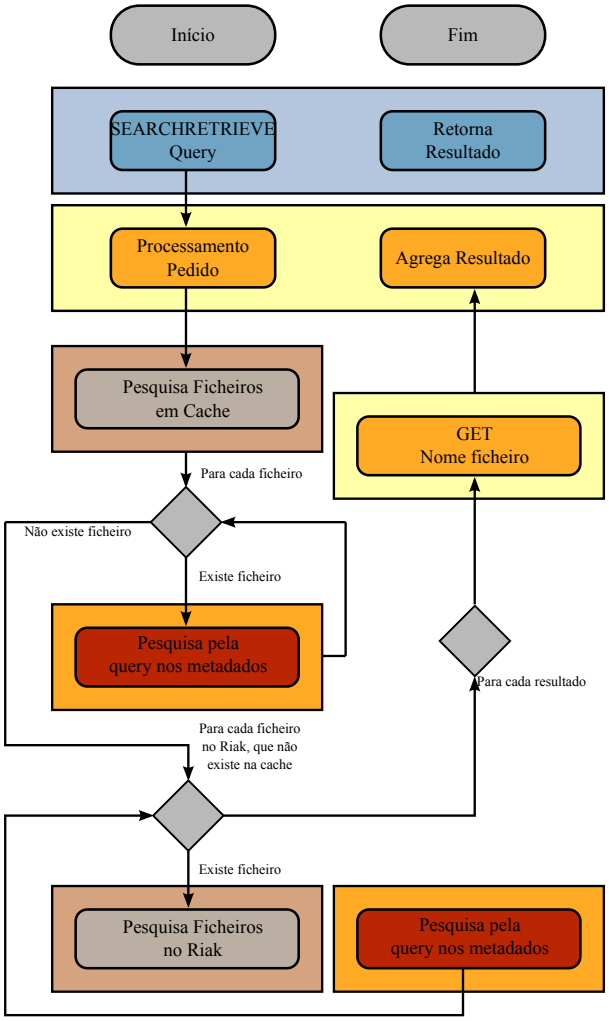



Figura 3.7: Sequência de operações que descrevem um pedido SEARCHRETRIEVE

Já para o caso do `searchretrieve` (Figura 3.7), é feito um pedido `get` para cada um dos resultados obtidos, sendo os mesmos agregados e enviados para o utilizador final.

### 3.1.6 API do sistema

A API do sistema fornece operações idênticas àquelas disponibilizadas pelos vários provedores de *clouds* de armazenamento públicas existentes hoje em dia. Regra geral, todas apresentam o mesmo subconjunto de operações de entre as mais usuais e necessárias para um serviço desde tipo (inserção, obtenção e remoção). Infelizmente, não existe atualmente uma API genérica definida para todos os provedores de *clouds*, de modo a que estes tenham um conjunto de regras a seguir, conduzindo a uma uniformização do acesso a qualquer que seja a *cloud*. A utilização de conectores, descrita acima, provém desta necessidade de criar uma interface única para acesso às várias *clouds*, de forma a dotar o sistema de uma maior transparência neste sentido. Independentemente da *cloud* usada, havendo a implementação do conector correspondente, o acesso torna-se completamente

transparente para o utilizador. Por fim, existe ainda o exemplo da Lunacloud, que reaproveitou a API da Amazon S3, oferecendo quase total compatibilidade com a mesma (pelo menos para as operações mais usuais, mencionadas em cima).

É neste sentido que a API externa do *middleware* foi pensava. Ao dotar o sistema de uma interface comum e que os utilizadores já estão familiarizados, possibilitamos a que os mesmos consigam integra-lo em aplicações previamente desenvolvidas com vista ao uso de outras *clouds*. Estas aplicações, estando já a fazer o uso de *clouds* como a Amazon S3, por exemplo, podem ser facilmente adaptadas para usar a T-Stratus como uma *cloud* segura e confiável para o armazenamento e pesquisa segura sobre os dados. Dada a não uniformização das APIs dos vários provedores, a inserção e obtenção podem variar nos parâmetros/argumentos passados, nomeadamente, muitas usam objetos próprios para a transmissão da informação. No entanto, no *middleware* desenvolvido, e com exceção das pesquisas, optou-se por uma visão mais simplista, onde as chaves são dadas por `Strings`, correspondentes aos nomes dos ficheiros, e o valor é um qualquer conjunto de `bytes`.

A API disponível consiste em cinco tipos de operações (Tabela 3.1): `put`, `get`, `remove`, `search` e `searchretrieve`, sendo a `listfiles` para efeitos de teste no controlo dos ficheiros presentes no sistema.

| **Operação** | **Dados de entrada** | **Resultado** |
|---|---|---|
| **GET** | `String` | `byte[]` |
| **PUT** | `String` e `byte[]` | Sucesso ou Insucesso |
| **REMOVE** | `String` | Sucesso ou Insucesso |
| **SEARCH** | `String[]` | `TStratusMessage` |
| **SEARCHRETRIEVE** | `String[]` | `TStratusMessage` |

Tabela 3.1: Operações disponíveis pela API do sistema *middleware*

A forma como estas se traduzem numa sequência de operações no sistema foi endereçada na subsecção anterior, sendo a descrição seguinte mais focada na invocação das várias operações da API. Um pedido de `put` permite a inserção de um ficheiro no sistema, tendo em conta um repositório do tipo chave-valor, onde o nome do ficheiro corresponde à chave (`String`) e o valor diz respeito aos dados do ficheiro (`bytes[]`). Na verdade, este é o tipo de inserção semelhante à do próprio índice do sistema *middleware*. Neste, o valor diz respeito a um objeto que é serializável e passado como um conjunto de bytes que corresponde à informação acerca do ficheiro armazenado, como por exemplo, o seu conjunto de metadados e os nomes e respetivas provas de integridade sobre os fragmentos que constituem o mesmo. Não obstante, a informação passada pelo utilizador num pedido desde tipo diz respeito ao nome e aos dados do ficheiro em questão. Um pedido `get` permite ao utilizador obter um dado ficheiro, previamente inserido. Para tal, basta enviar como argumento o nome do ficheiro a obter (`String`). Caso se confirme a existência do mesmo no sistema, é retornado um conjunto de bytes (`byte[]`)

correspondente aos dados do mesmo. Um pedido de `remove` também tem como único argumento o nome do ficheiro a eliminar (`String`). O mesmo é removido do índice e das quatro *clouds* públicas de armazenamento usadas. Não obstante, derivado dos modelos de custo dos vários provedores associados ao armazenamento, uma das vantagens do sistema é que não é necessário recorrer à remoção dos ficheiros em caso de cessação do uso de uma das quatro *clouds*, pois os dados nela presentes são ilegíveis por qualquer individuo que possa vir a ter acesso aos mesmos. Por fim, o `search` e o `searchretrieve` correspondem a duas pesquisas idênticas que, como mencionado na subsecção anterior, diferem no tipo de resultado que retornam. Como argumento, qualquer um destes pedidos recebe uma ou mais palavras a pesquisar nos metadados dos ficheiros contidos no sistema (`String[]`). O resultado das pesquisas já envolve uma estrutura de dados diferente, nomeadamente, um objeto capaz de apresentar uma lista dos vários resultados ordenados consoante a relevância (`TStratusMessage`).

## 3.2 Generalização do sistema numa arquitetura em *cloud*

Nas subsecções seguintes é descrita a transposição do sistema *middleware*, definido anteriormente segundo a sua variante *proxy*, para a variante *cloud*.

### 3.2.1 Generalização do modelo inicial

O sistema, tal como se encontra definido na secção 3.1 pressupõe uma instanciação do mesmo num servidor confiável (variante *proxy*, instalado na rede local do utilizador final). Todas as propriedades de segurança e confiabilidade da solução são garantidas e o acesso às múltiplas *clouds* de armazenamento é intermediado por uma instância do *middleware* descrito. Tal como foi dito ao longo deste capítulo, a visão final pressupõe a instanciação do mesmo numa *cloud* privada, com total controlo do lado dos utilizadores, onde várias instâncias do *middleware* seriam executadas. Generalizando o modelo anterior, estas instâncias formam um ambiente distribuído de computação e gestão dos dados dos utilizadores. Visto ser uma generalização da solução anterior, este *cluster* de servidores de instâncias do *middleware* preserva todas as características de segurança e confiabilidade anteriormente definidas. Os utilizadores podem ter acesso ao sistema de forma ubíqua a qualquer uma das instâncias disponíveis.

É esta variante em *cloud*, representada na Figura 3.2.1, que associa metaforicamente a designação de stratus ao *cluster* de servidores que, por sua vez, comunica com as várias *clouds* públicas de armazenamento, bem como o índice distribuído. O sistema, visto como uma nuvem baixa (stratus, Figura 3.9[1]), uniforme, que cobre uma vasta extensão horizontal, "esconde"as restantes nuvens heterogéneas, mais altas e que podem apresentar as mais variadas características. As *clouds* heterogéneas de armazenamento de dados poderiam ser caracterizadas como *Cirrus*, *Cirrocumulus*, *Altocumulus*, *Altostratus*, etc.

[1] retirado de `http://eo.ucar.edu/webweather/cloud3.html` acedido e verificado a 19-03-2013

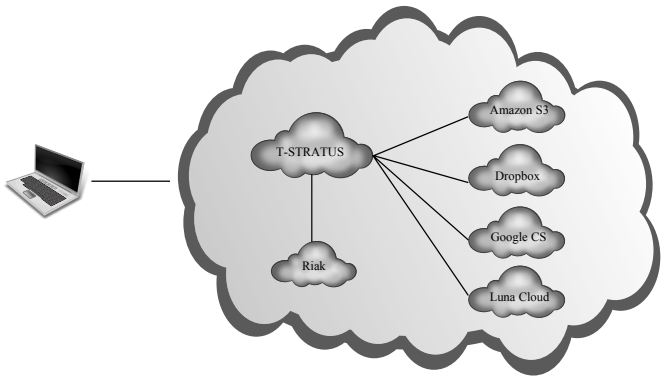



Figura 3.8: Visão geral do sistema, solução suportada num sistema distribuído por diversos nós (*cluster*) numa *cloud* privada

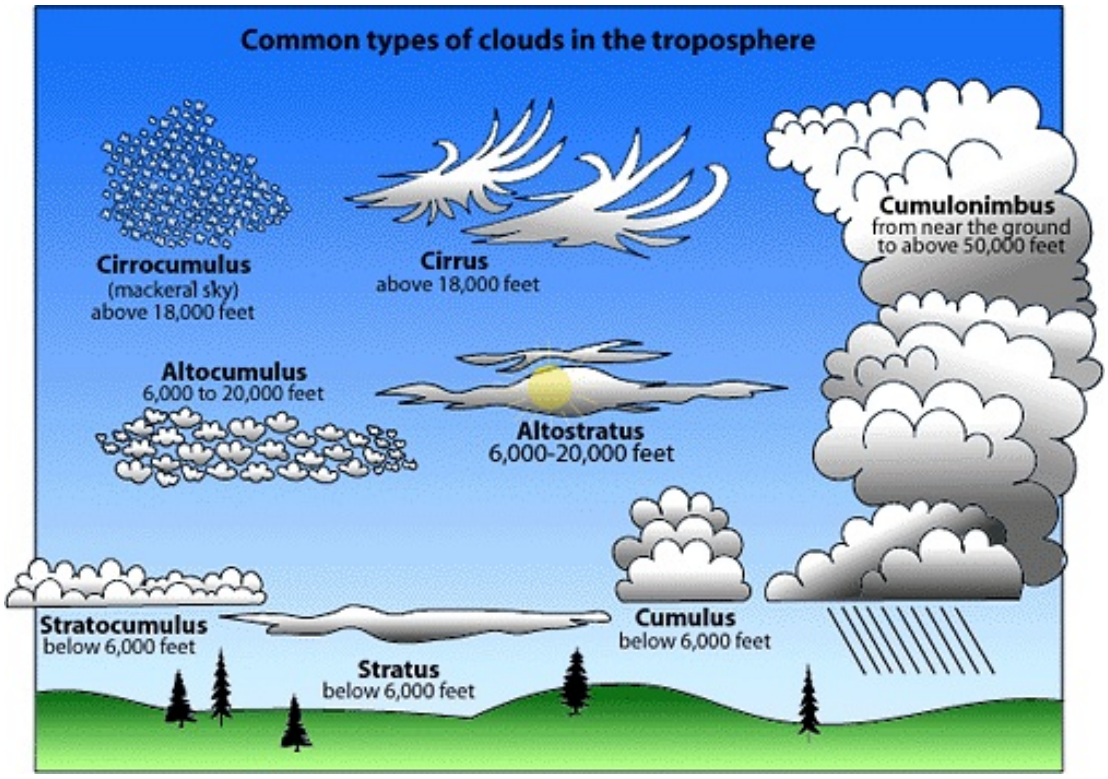



Figura 3.9: Tipos de nuvens, através das quais é caracterizado metaforicamente o sistema T-Stratus

### 3.2.2 Arquitetura da solução

A arquitetura aqui descrita consiste num sistema distribuído constituído por várias instâncias do *middleware* definido na secção 3.1. No seu conjunto, formam um *cluster* de servidores, sendo que a replicação é feita por uma camada *out of box*, suportada pelo sistema de indexação Riak. Na verdade a utilização deste componente não é vista como um aspeto focado nos objetivos da dissertação, no entanto, por esta via, a formação deste *cluster* permite ter uma arquitetura *cloud* (Figura 3.10). Os utilizadores têm acesso a uma destas instâncias que, nesta visão, incorpora um novo módulo, responsável pela gestão concorrente de múltiplos utilizadores, bem como a distribuição uniforme dos mesmos pelos vários nós, de maneira a distribuir o processamento das várias instâncias e não sobrecarregar o sistema. Tal como descrito na arquitetura do *middleware*, cada nó do sistema

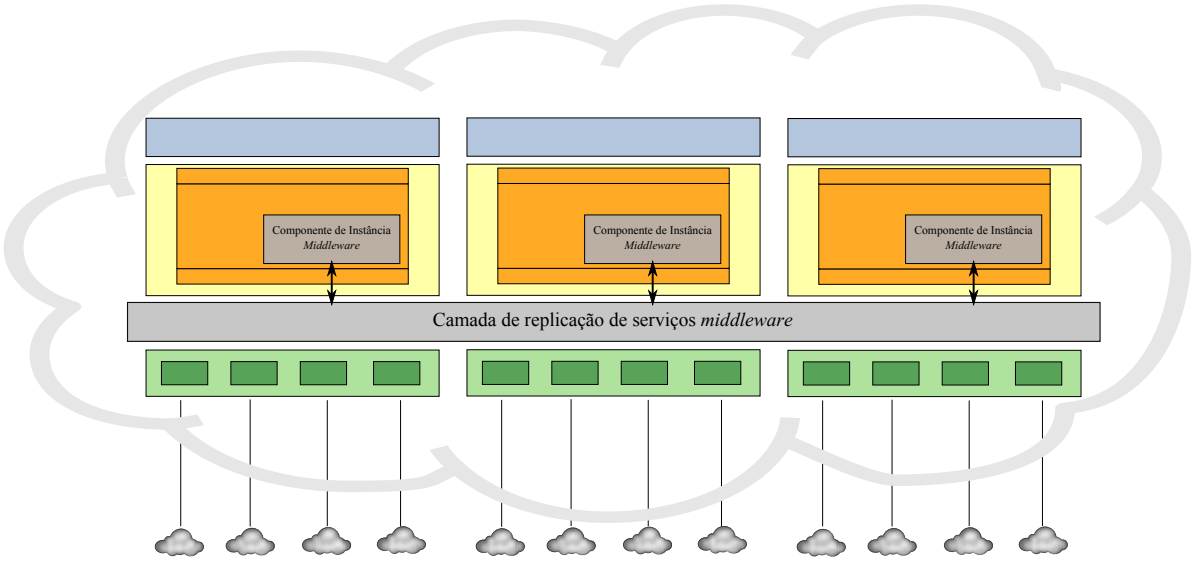



Figura 3.10: Visão arquitetural da variante *cloud* do sistema T-Stratus

possui a sua própria instância cliente de acesso ao índice Riak. Este, no entanto, é comum a todas as instâncias e poderá representar um papel chave (como será abordado na subsecção 3.2.3) para o mecanismo de controlo de concorrência, através do seu modelo de consistência de dados. Os segredos criptográficos que asseguram a autenticidade, privacidade e pesquisa dos dados armazenados são distintos para cada utilizador. Este fator permite que um único índice, como o Riak (com vários nós que formam também um *cluster* próprio), possa suportar todo o armazenamento dos objetos associados aos ficheiros introduzidos, num ambiente altamente escalável, tal como já suporta para uma instância do *middleware* apenas.

O novo módulo, visível na figura anterior, permite a comunicação entre as várias instâncias, de forma a assegurar as propriedades anteriormente definidas. É nesta visão final que todo o sistema pode operar num ambiente *cloud*, privado, onde o controlo da

base de confiança permanece do lado dos utilizadores finais, mantendo as *clouds* dos vários provedores apenas como *outsourcing* para o armazenamento dos dados.

### 3.2.3 Papel do componente Riak

Como mencionado em cima, a base de dados distribuída Riak [Ria] desempenha um papel importante para atingir o objetivo da variante em *cloud* que se pretende. Consiste num sistema distribuído composto por vários nós, de forma a formar um *cluster*. Estes nós podem ser distribuídos ao nível de servidores num único centro de dados, bem como ao nível de múltiplos centros de dados. Associado a esta distribuição dos vários constituintes, torna-se uma solução altamente tolerante a falhas, correspondendo aos objetivos pretendidos.

Os dados armazenados, neste caso, objetos com informação associada aos ficheiros colocados pelos utilizadores finais, encontram-se replicados pelo menos por três nós diferentes (valor por defeito), podendo os mesmos pertencer a diferentes centros de dados. Este modelo de replicação é particularmente importante para a solução perspetivada pois garante a recuperação dos dados na eventualidade de falha de um ou mais nós que compõem o Riak. Neste cenário, um nó vizinho do que falhou (por razões arbitrárias) assume as operações de escrita e leitura deste último, enquanto o mesmo não for recuperado. Cada nó mantém os dados armazenados através do Bitcask [SSBT10], detalhado na secção 2.1.2. Esta persistência dos dados assegura que a informação introduzida no Riak acerca dos vários ficheiros dos utilizadores está rapidamente acessível e com baixa latência[2]. Mesmo na eventualidade de falha, o modelo de replicação assegura a sua recuperação.

O processamento dos pedidos de escrita e leitura pode ser feito por qualquer um dos nós existentes no sistema. Um utilizador pode requerer um pedido a um nó qualquer e, o próprio, na eventualidade de não possuir os dados pretendidos, reencaminha o pedido para um nó capaz de fornecer a resposta. Para garantir que o processamento é distribuído de maneira uniforme, a distribuição das chaves é feita com recurso a funções de *hash* consistente. Assim, o uso do sistema T-Stratus por múltiplos utilizadores em simultâneo, com acessos recorrentes ao índice, não sobrecarrega apenas um nó do sistema de indexação, pois qualquer um pode coordenar uma operação de leitura ou escrita.

Posto isto, o uso do Riak como índice persistente permite assegurar as propriedades do sistema atual e ainda dar suporte ao uso por parte de múltiplas instâncias, de forma transparente. É garantida uma elevada disponibilidade, bem como escalabilidade, pois o processo de adição de um nó ao sistema Riak é dinâmico e a distribuição do processamento e das chaves é feita de forma automática. Esta propriedade permite dotar o sistema de uma maior capacidade de resposta, caso se verifique que os servidores existentes se começam a revelar ineficientes para poder dar resposta ao número de pedidos efetuados e à quantidade de informação armazenada.

[2] http://docs.basho.com/riak/1.2.0/tutorials/choosing-a-backend/Bitcask/ acedido e verificado a 22-03-2013

Contudo, outras abordagens como o Cassandra (descrito anteriormente) poderiam ser adotadas como índice distribuído para o sistema T-Stratus, não fosse a semântica e contexto do Riak preferencial para este caso. Assumindo que as operações predominantes no sistema são de leitura, que o modelo de tolerância a falhas é bastante importante, a disponibilidade permanente é fundamental e a escalabilidade pode ser facilmente controlada com a adição dinâmica de nós ao *cluster*, o Riak torna-se um forte candidato[3]. Embora o Cassandra consiga dar resposta às propriedades anteriores, o seu desenho não é tão orientado a *key-value-store* (como o Riak), mas mais *SQL-Like*, característica que não é valorizada no sistema atual. No Cassandra, a agregação dos dados em colunas e famílias de colunas, com uma abordagem semelhante à de uma base de dados relacional, confere complexidade desnecessária ao modelo de dados em relação à solução que se pretende, de forma a satisfazer os objetivos pretendidos. O modelo de dados do Riak apresenta um conceito simples e familiar de chave-valor, ao passo que o Cassandra apresenta o seu próprio modelo de dados. No fundo, o Cassandra é mais orientado para uma solução Bigtable e não tanto Dynamo, como o Riak.

No entanto, o acesso concorrente não se resolve, por si só, com a utilização do Riak, dado o seu modelo de consistência. Se assumirmos que não existem escritas concorrentes sobre o mesmo ficheiro (*single-writer multi-reader*), o modelo de consistência do Riak (*eventual consistency*) acaba por ser suficiente. No entanto, tendo como visão final a utilização do sistema por diversas aplicações, podendo as mesmas aceder concorrentemente aos mesmos dados, esta problemática tem de ser contornada com novas abordagens, sendo essa a discussão da subsecção 3.3.1, em baixo.

### 3.2.4 Ambiente de interligação da *cloud* T-Stratus

De forma a garantir resposta em tempo útil, face à latência verificada entre as comunicações das várias instâncias do sistema como variante *cloud*, bem como o índice Riak e o protocolo de tolerância a falhas bizantinas, é aconselhável a utilização de canais de comunicação dedicados entre os mesmos. Estes devem ter capacidade e taxas de transferência elevadas, preferencialmente, com ligações diretas de elevada performance não descartando, contudo, caminhos alternativos na eventualidade de falha de uma ou mais ligações.

No geral, o sistema forma uma rede corporativa (a proteção das comunicações pode fazer-se por SSL ou TLS), pois é considerada a possibilidade de ataques às comunicações entre os vários nós. Não obstante, os servidores são confiáveis, tal como o eram na sua variante *proxy*, agora generalizada. No final, obtém-se a nuvem de computação T-Stratus, como uma base global de confiança dos utilizadores.

A rede poderá ser Internet, podendo ser adequada uma solução em que o *cluster* que forma a nuvem T-Stratus é interligado por *IPSec Tunneling* por razões de usabilidade e eficiência, protegendo o tráfego entre dois pontos. No entanto, este ambiente e o seu

[3] `http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis` acedido e verificado a 20-03-2013

impacto não está no foco da dissertação, estando a mesma focada nos aspetos de performance, segurança e confiabilidade de uma instância do *middleware* da qual seria possível a generalização para o sistema descrito ao longo desta secção.

## 3.3 Aspetos complementares ou de extensão ao modelo de sistema

Nesta secção são discutidos aspetos de extensão do modelo atual, nomeadamente na resolução de problemas de concorrência, que estão diretamente associados ao modelo de consistência considerado.

### 3.3.1 Aspetos de controlo de concorrência

Como mencionado em cima, existem aplicações onde o modelo *eventual consistency* é insuficiente, devido à possibilidade de acesso concorrente aos mesmos dados, por parte de uma ou mais aplicações. Este modelo assume que a escrita dos dados é, eventualmente, efetuada. No entanto, não previne situações em que uma leitura pode obter a versão errada, quando o processo de escrita ainda se encontra a ser propagado pelos vários nós, ao contrário de modelos em que aguardam expressamente pela confirmação de escrita nos vários nós e só depois disponibilizam os dados para leitura. Esta problemática é também abordada no DepSky [BCQAS11] e podemos facilmente transpor para a T-Stratus. Como os próprios autores referem, se o sistema fosse executado numa rede local, um sistema de coordenação como o Zookeeper poderia ser usado de forma a coordenar as operações de escrita. Contudo, pode constituir um problema quando transposto para a escala da Internet, com os componentes dispersos por qualquer ponto, principalmente por questões de disponibilidade do serviço face a problemas na rede.

Uma solução proposta pelos autores passa pelo uso do próprio sistema (neste caso, T-Stratus) para criar um mecanismo de *locks* sobre os dados acedidos para escrita. A ideia consiste em poder ter, em qualquer instante, um *lock* associado a um ficheiro existente no sistema. Esse *lock* consiste num ficheiro, também ele armazenado num quórum de *clouds*. Assim, qualquer processo $p$ que queira escrever num dado ficheiro, primeiro procura pelo respetivo *lock* no sistema. Se o mesmo for encontrado no quórum de *clouds*, e se o tempo de vida associado não expirou, significa que alguém está a escrever nesse ficheiro. Caso contrário, $p$, que pretende escrever, pode criar o *lock* respetivo e envia-lo para o quórum. No final, de forma a que $p$ possa ter a certeza que é o único a escrever para o ficheiro em questão, efetua uma procura sobre possíveis *locks* associados ao mesmo, para além do seu. Se encontrar algum, remove o seu *lock* do sistema, caso contrário, está seguro de que é o único a escrever.

Na verdade, é possível parametrizar o modelo de consistência do Riak de modo a poder usá-lo como mecanismo de gestão de concorrência. Contudo, esta parametrização

envolve um *tradeoff* entre a consistência, disponibilidade e particionamento, três características bastante importantes de preservar no sistema.

Por fim, a consistência também pode ser um problema ao nível das *clouds* de armazenamento de dados usadas, que consequentemente definem o modelo de consistência do sistema T-Stratus como sendo o mais fraco de entre os modelos das várias *clouds* de armazenamento usadas. Tanto a Dropbox, como a Amazon S3, apresentam *eventual consistency*, embora em alguns casos a Amazon S3 apresente *read-after-write*. Já a Google Cloud Storage apresenta apenas consistência *read-after-write*. O sistema T-Stratus teria muito possivelmente *eventual consistency*, embora não tenha sido encontrada informação acerca do modelo de consistência da Lunacloud. Uma forma de contornar este problema seria, após uma escrita, efetuar a respetiva leitura até obter com sucesso os dados de um quórum.

### 3.3.2 Aspetos de reconfiguração dinâmica

Na eventualidade de substituição de uma *cloud*, tanto por questões de custos, como problemas de confiabilidade, seria interessante a disponibilização de um protocolo de reconfiguração do sistema. Este iria remover os dados da *cloud* que pretendemos deixar de usar e consequentemente distribui-los pela nova *cloud* adotada. Este processo passaria pela leitura dos dados existentes no sistema, seguido da execução do protocolo de escrita dos mesmos, tanto nas *clouds* existentes, como na que foi adicionada. Finalmente, os dados da *cloud* a remover seriam eliminados e esta deixaria de pertencer ao sistema. Cada *cloud*, após o processo de reconfiguração, iria criar um ficheiro especial, pré definido. Quando fosse possível obter esse ficheiro de um quórum de *clouds*, podíamos concluir que o processo tinha finalizado com sucesso.

### 3.3.3 Possibilidade de armazenamento de versões de objetos e fragmentos

Por fim, poderia ser explorada a existência de várias versões de um mesmo ficheiro de modo a poder recuperar ou reparar um conjunto de dados, se necessário. Esta abordagem seria semalhante à Dropbox, que também mantém várias versões de um ficheiro ao longo do tempo. Não obstante, por questões de custo, não é viável permitir o armazenamento de versões ilimitadas, sem imposição de um limite sobre as mesmas. Para contornar esta situação, poderia ser executado um processo em *background* que iria proceder à remoção das $n$ últimas versões dos vários ficheiros armazenados no sistema.

# 4

# Implementação

Neste capítulo é descrita a implementação do sistema, com base nas decisões tomadas no capítulo anterior relativamente às contribuições desta dissertação. Inicialmente é descrita a estrutura/organização do código, seguido de uma descrição mais pormenorizada de como os componentes se encontram implementados, com ênfase nos pontos mais relevantes e de maior destaque.

## 4.1 Estrutura do Sistema

O *middleware* encontra-se implementado em Java. Na conceção da estrutura/organização do código esteve sempre presente o objetivo de ser o mais fiel possível em relação à arquitetura descrita na Figura 3.1. Deste modo tornasse clara e percetível a separação dos vários componentes, apenas ao olhar para o código, tal como o é na figura de referência. Por outro lado, esta organização facilita na reutilização dos vários componentes e permite que a eventual substituição de um deles possa ser feita sem grande intrusão no código, pois basta que o mesmo implemente os métodos da interface em questão. As várias classes que compõem o sistema encontram-se organizadas em pacotes (*packages*), de acordo com os vários componentes. No nível mais acima, temos três *packages*: `exec`, `tstratus` e `utils`.

No `exec` constam várias classes que não dizem respeito a funcionalidades do sistema propriamente dito, mas que são necessárias para dar inicio à execução do mesmo. É a partir das classes existentes neste *package* que se executam os vários componentes do sistema. A classe `BizServer` permite iniciar os quatro servidores bizantinos, que vão realizar as leituras e escritas para as quatro *clouds* usadas. A classe `TStratusServer` implementa os métodos da interface `ITStratusServer` e representa uma instância

do *middleware* que aceita pedidos através de *web services*, por parte de clientes remotos. Estes pedidos recorrem a um objeto próprio para o retorno do resultado da respetiva operação. Este corresponde à classe `TStratusMessage`, que representa a transferência de dados entre o cliente remoto e o *middleware*, constituindo um ou mais objetos do tipo `TStratusObject`. Já a `TStratusLocalClient` representa uma instância do *middleware* com um cliente local, a executar na mesma máquina.

No *package* `utils` existem vários utilitários usados no sistema tais como operações sobre `Strings`, chaves criptográficas e indicadores de performance com base nos tempos medidos. A classe `ObjectStats` representa um objeto com os tempos medidos para uma dada operação efetuada, enquanto que a `TimeMeasure` realiza a escrita desta informação num ficheiro próprio, em memória persistente, de forma a ser possível a extração de resultados para posterior avaliação. A `KVMessage` corresponde ao objeto usado na troca de informação (comunicação) entre o cliente bizantino e os respetivos servidores. A classe `keyUtils` permite a criação e escrita de chaves simétricas, vetor de inicialização e *seed* para disco, de modo a serem usados pela aplicação, e a classe `SamplesUtils` é responsável pela leitura dos parâmetros registados em ficheiros de configuração para os vários componentes do *middleware*. Por fim, neste *package*, existe a classe `Pair`, que representa precisamente um par de objetos de qualquer tipo e as classes `Utils` e `UtilsBase` que comportam operações sobre bytes, segredos criptográficos, bem como diversas conversões entre `bytes` e `Strings`.

Finalmente, no *package* `tstratus` (Figura 4.1) encontramos diversos *subpackages*, nomeadamente: `bizquorum`, `cloudobject`, `compression`, `connectors`, `cryptosystem`, `index`, `integrity`, `metadata` e `proxy`, correspondentes aos vários componentes descritos no modelo de arquitetura. No `bizquorum` encontra-se a classe `BFTSMaRt`, sendo que uma instância da mesma corresponde a um servidor do protocolo de tolerância a falhas bizantinas usado. Por sua vez, encontra-se neste *subpackage* também a classe `TStratusBizClient`, que representa o cliente bizantino. De seguida, o `cloudobject` contém a classe representativa de um objeto armazenado no índice (classe `CloudObject`), bem como os seus metadados associados (interface `IObjectMetadata` e classe `ObjectMetadata`). O `compression` contém a classe responsável pela compressão e descompressão dos ficheiros a armazenar. O algoritmo usado, implementado na classe `ZipCompression`, implementa as funcionalidades da interface `ICompression`. Para a implementação de um novo algoritmo, basta que este respeite a mesma, sendo que esta filosofia se aplica para os restantes módulos e respetivas interfaces. De seguida, no `connectors` temos os quatro conectores (`AmazonS3Cloud`, `LunaCloud`, `DropboxCloud` e `GoogleCloudStorage`) para as quatro *clouds* adotadas. Estes implementam a interface `ICloud`, que permite dotar o sistema de acesso transparente, independente da *cloud* em questão e da sua API. No `cryptosystem` estão as classes e respetivas interfaces que dizem respeito à cifra/-decifra dos fragmentos enviados para a *cloud* (`CloudCipher` e `ICloudCipher`) bem como a cifra e verificação dos metadados (`SearchCipher` e `ISearchCipher`). Ao nível do índice (*subpackage* `index`) temos a classe `HashMapIndex` que implementa a cache

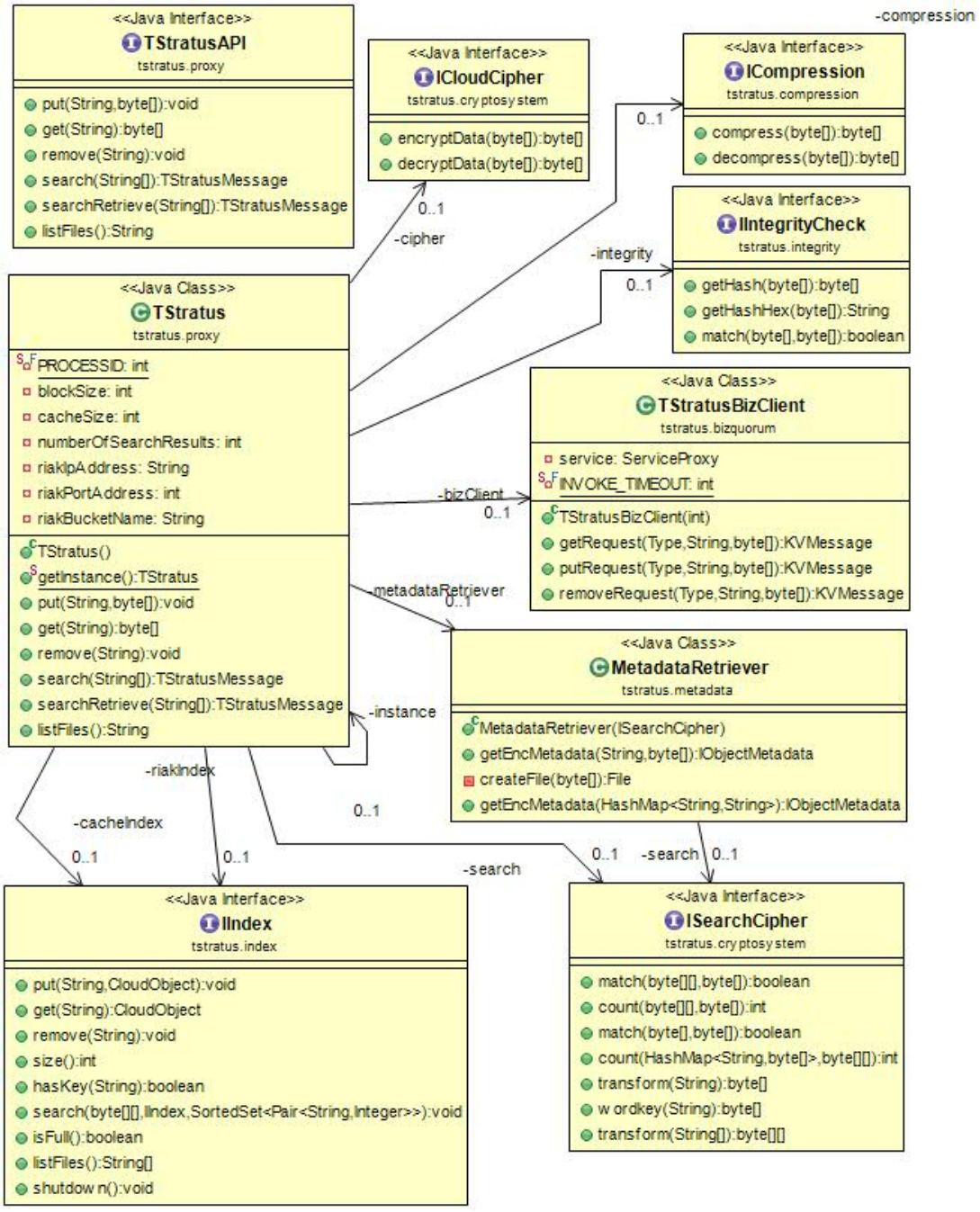



Figura 4.1: Componentes do *middleware* e forma como se relacionam (*package* `tstratus`).

e a classe `RiakTStratusIndex` que implementa o índice propriamente dito, como uma instância cliente do Riak. Ambas implementam os métodos da interface `IIndex`, constituindo dois índices acessíveis de forma transparente. De seguida, no `integrity` consta uma implementação de uma função de *hash* seguro (classe `Sha2`) que implementa os métodos genéricos da interface `IIntegrityCheck`. O *subpackage* `metadata` permite extrair os metadados dos ficheiros (classe `MetadateRetriever`) a armazenar, sendo esta a primeira operação realizada pelo *middleware* ao processar um pedido de `put`. Por fim, o *subpackage* `proxy` representa o componente TStratus, agregador de todas as funcionalidades do *middleware*. A classe `TStratus` contém todo o processamento, com a agregação dos vários módulos descritos acima, sendo feita uma junção das operações dos mesmo que vão constituir uma resposta a um pedido. Os métodos disponíveis pela API encontram-se na interface `TStratusAPI`, descrita na subsecção 3.1.6, que permite o uso da aplicação pela camada acima (interface).

## 4.2 O protocolo de tolerância a falhas bizantinas

A escolha do protocolo de tolerância a falhas bizantinas a usar constituiu uma parte importante no que diz respeito ao desenho do sistema inicial. Após o estudo efetuado no Capítulo 2, tomou-se como hipótese considerar o algoritmo Paxos. No entanto, este não permite a tolerância total em relação a falhas bizantinas, como poderia ser expectável. O algoritmo Paxos apenas permite que se estabeleça um consenso entre as várias partes envolvidas, de modo a manter a mesma sequência ordenada de operações nas várias réplicas. O `BFT-SMaRt`, por sua vez, mostrou ser um forte candidato ao sistema a implementar. O seu desenho, definido como um protocolo de tolerância a falhas bizantinas altamente robusto e de elevada performance, cumpre os objetivos pretendidos para a escolha do algoritmo em questão. Para além disso, a sua forte orientação para a *cloud* (usado em projetos como TClouds[1]) e o facto de ser implementado como uma biblioteca Java, constituem outros dois fatores que favorecem o uso desta biblioteca, no sistema implementado.

### 4.2.1 Servidores Bizantinos

Esta biblioteca `BFT-SMaRt` pressupõe a existência de quatro ou mais servidores, de acordo com o número de falhas que se pretende tolerar. Nesse sentido, e de acordo com o modelo $3f + 1$ em que $f$ réplicas podem falhar, não seria relevante o uso deste protocolo para um número de réplicas inferior a quatro. Deste modo, o sistema foi dotado da capacidade de tolerar a falha de uma *cloud*.

Para a implementação de um servidor bizantino através desta biblioteca, são necessárias algumas convenções. A classe correspondente (Listagem 4.1) tem de implementar `SingleExecutable` e `Recoverable`. Por sua vez, estas requerem a implementação de

[1] `http://www.tclouds-project.eu/` acedido e verificado a 24-03-2013

vários métodos. O `executeUnordered`, invocado em pedidos do tipo `get`, visto que não precisa de preservar a ordem por se tratar de uma leitura apenas. Já para um pedido do tipo `put` ou `remove`, é usado o método `executeOrdered`, pois são operações de escrita que vão provocar alterações ao estado da réplica e que, portanto, devem ser efetuadas pela mesma ordem nas mesmas.

De forma a manter o estado concordante entre as várias réplicas, é exigido pelo protocolo que as mesmas partam do mesmo estado inicial. Para tal, o método `setState` obtém inicialmente uma listagem de todos os ficheiros existentes no índice, colocando-os numa estrutura do tipo `HashMap`, a ser usada por cada uma das réplicas. O estado atual de uma réplica é obtido através do `getState`, também necessário implementar para o funcionamento do protocolo. Este estado é armazenado num objeto próprio, serializável, do tipo `ICloudStateIndex`. Por fim, a *cloud* para a qual será efetuada a escrita e leitura dos ficheiros, corresponde a um objeto do tipo `ICloud`, que diz respeito à interface que os vários conetores implementam.

```
1  public interface IBFTSMaRt {
2
3   public byte[] executeUnordered(byte[] command, MessageContext msgCtx);
4
5   public byte[] executeOrdered(byte[] command, MessageContext arg1);
6
7   public ApplicationState getState(int arg0, boolean arg1);
8
9   public int setState(ApplicationState state);
10 }
```

Listagem 4.1: Código correspondente a um servidor bizantino

### 4.2.2 Cliente Bizantino

De seguida, para que se pudesse estabelecer uma comunicação entre a aplicação e os servidores bizantinos, nomeadamente, o servidor correspondente à réplica primária, foi necessária a implementação do respetivo cliente `BFT-SMaRt` (Listagem 4.2). Os pedidos

```
1  public interface ITStratusBizClient {
2
3   public KVMessage getRequest(KVMessage.Type requestType, String objName, byte[]
        data);
4
5   public KVMessage putRequest(KVMessage.Type requestType, String objName, byte[]
        data);
6
7   public KVMessage removeRequest(KVMessage.Type requestType, String objName,
      byte[] data);
8  }
```

Listagem 4.2: Código correspondente ao cliente bizantino

são remetidos para o quórum bizantino através do `ServiceProxy`, que é executado em

função de cada um dos três tipos de pedidos existentes: `getRequest`, `putRequest` e `removeRequest`. Para os dois últimos, como mencionado na subsecção acima, são efetuados pedidos do tipo `invokeOrdered`, enquanto que para o primeiro são efetuados pedidos do tipo `invokeUnordered`. O objeto serializável usado na troca de pedidos entre o cliente e as várias réplicas é o `KVMessage`, reaproveitado de um exemplo da própria biblioteca BFT-SMaRt. Este tem apenas o tipo de operação, a chave usada (`String`) e o valor (conjunto de `bytes`).

### 4.2.3 Conectores

A interface que permite o uso transparente das APIs das várias *clouds* de armazenamento intervenientes no sistema encontra-se na Listagem 4.3. Aqui são apresentados os métodos que permitem a realização das três operações pretendidas, `putObject`, `getObject` e `deleteObject`, com a particularidade de que as operações especificas de cada API estão implementadas nos respetivos conectores e são, desta maneira, expostas de igual forma.

```
1  package tstratus.connectors;
2
3  public interface ICloud {
4
5   /**
6    * Stores data in the cloud
7    * @param key name of the file
8    * @param b data
9    */
10  public void putObject(String key, byte[] b);
11
12  /**
13   * Retrieves data from the cloud
14   * @param key name of wanted file
15   * @return the file's data
16   */
17  public byte[] getObject(String key);
18
19  /**
20   * Deletes a file from the cloud
21   * @param key name of the file to remove
22   */
23  public void deleteObject(String key);
24 }
```

Listagem 4.3: Código correspondente à interface de acesso às *clouds*

## 4.3 O índice distribuído Riak e a cache para acesso rápido

A escolha da base de dados NoSQL Riak como índice principal do sistema recai principalmente na sua implementação fiel ao Dynamo, de um repositório do tipo chave-valor, que

apresenta as características pretendidas para o sistema implementado[2]. Elevada disponibilidade, escalabilidade, tolerância a falhas e replicação. Proporciona uma grande modularidade/independência, pois pode ser executado numa máquina independente daquela onde o *middleware* se encontra. Por outro lado, o sistema não assume uma divisão dos ficheiros em conjuntos ou grupos (como seria possível, embora neste caso não necessário, no Cassandra, com a atribuição de famílias de colunas), pois assume que o índice contém informação de ficheiros genéricos armazenados pelo utilizador. Como mencionado anteriormente, a implementação do Cassandra é um misto de Dynamo e Bigtable e seria de considerar se as operações predominantes fossem de escrita e se o acesso aos dados envolvesse uma distinção semelhante à das bases de dados relacionais, por tabelas. No modelo de arquitetura definido, assume-se que a operação predominante é a de leitura e é na performance a esse nível que nos focamos, para além das restantes características mencionadas acima. Posto isto, no Riak, é ainda possível a criação de *buckets* para uma melhor gestão dos dados, à semelhança das *clouds* usadas. Desta forma, conseguimos obter uma estrutura simples de usar, do tipo chave-valor, não centralizada e onde toda a gestão de concorrência e replicação é deixada do lado da mesma, já de si bastante eficiente. Por fim, a disponibilização de uma API acessível em Java por parte do Riak, permite o acesso direto às várias operações disponibilizadas, sendo a sua incorporação na solução implementada bastante direta.

Não obstante, a necessidade de proporcionar um acesso rápido aos ficheiros mais requisitados ou mais recentemente introduzidos, não permite abdicar de um sistema de cache que permita essa funcionalidade. Contudo, o seu tamanho pode influenciar, tanto a capacidade de armazenamento do servidor onde o *middleware* se encontra a executar (cache demasiado grande), como se pode tornar obsoleto para o caso dos pedidos serem praticamente todos remetidos para o índice principal (demasiado pequena). O tamanho da cache, em número de ficheiros, é parametrizável através da configuração inicial do sistema. Estima-se que o mesmo deve corresponder a cerca de 20% do número total de entradas no índice (ficheiros no sistema).

As operações que qualquer um destes índices implementa encontram-se na interface `IIndex`, na Figura 4.1. Para obtenção, remoção e inserção encontramos, respetivamente, os métodos `get`, `remove` e `put`, sendo o `get` associado a um `CloudObject`, objeto este que armazena toda a informação acerca do ficheiro no sistema, com exceção dos dados que o constituem. Apesar dos vários métodos serem auto-explicativos, existem dois que não se aplicam a um dos casos. O método `shutdown`, que se aplica apenas ao Riak e o `isFull`, que se aplica apenas à cache, pois o índice principal não tem tamanho limite. O método de pesquisa (`search`) é explicado em baixo, na subsecção 4.4, referente ao módulo responsável pelas pesquisas seguras.

[2] `http://docs.basho.com/riak/1.2.0/references/appendices/comparisons/Riak-Compared-to-Cassandra/` acedido e verificado a 27-02-2013, estudo comparativo de vários sistemas de armazenamento distribuídos

### 4.3.1 Cache

Para o sistema de cache foi escolhida a estrutura de dados `LinkedHashMap`. Esta estrutura de dados permite estipular a ordenação do conjunto de chaves (essa ordenação não existe no `HashMap` convencional). Ao indicar que a ordem pretendida deve corresponder à ordem de acesso, os ficheiros mais requisitados mantêm-se na cache e uma remoção vai sempre eliminar aquele que menos acessos teve. Por outro lado, são preservadas as propriedades de acesso direto do `HashMap` convencional, através da chave representativa do mesmo, com complexidade $O(1)$. Sempre que há uma inserção de um novo ficheiro no sistema, este é adicionado à cache. Se a mesma tiver atingido o seu limite, o último ficheiro (aquele com menor número de acessos) é removido.

### 4.3.2 Riak

O acesso ao índice principal (Riak) é possível através da API disponibilizada. Para tal, é necessária a criação de um `IRiakClient` que por sua vez recebe um endereço IP, porta e `Bucket` de (e para) onde serão efetuadas as leituras e escritas. As operações em si são invocadas através do `Bucket`, associado ao `IRiakClient` criado. Assim, como exemplo das três operações principais pretendidas temos `myBucket.store(key, objectToBytes(co)).execute()` para a inserção de um objeto; `myObject = objectFromBytes(data.getValue())` para a obtenção de um objeto do tipo `IRiakObject` e `myBucket.delete(objName).execute()` para a sua remoção.

## 4.4 Pesquisas seguras no índice

Para que os metadados possam ser pesquisados e mantidos de forma segura no índice, é necessário um mecanismo de cifra com propriedades homomórficas que permita que a comparação entre dois *ciphertext* dê o mesmo resultado que a comparação entre os respetivos *plaintext*. Para tal, como referido anteriormente neste documento, foi usado o esquema de Pesquisa Linear [FD12], correspondente a um algoritmo não determinista e com as propriedades homomórficas desejadas. A chave simétrica usada para o HMAC é de 128 bits criada com especificação do algoritmo AES (provedor *SunJCE*). As funções de *hash* (neste caso, o algoritmo de pesquisa usa o SHA-1) são as do provedor *BouncyCastle*.

A interface que expõe as várias operações deste módulo encontra-se na Figura 4.1 e na Listagem 4.4 podemos encontrar a operação de pesquisa ao nível do índice propriamente dito. Aqui, os métodos `match` permitem comparar dois valores cifrados e verificar se os mesmos são idênticos, sem nunca obter os respetivos *plaintext*. O `count` usa de forma transparente o `match` e obtém uma contagem do número de ocorrências encontrado. Este método é particularmente importante, pois é usado no `search` do índice (Listagem 4.4). Aqui, é passado como argumento um `wordkey`, que corresponde a um *hash* das palavras que constituem a pesquisa, com recurso a uma `masterkey`. É também passado o índice cache, de forma a não repetir a pesquisa para objetos já pesquisados no mesmo e, por

```
/**
 * Searches for a given query in the files metadata
 * @param wordkey encrypted query to search
 * @param cache Cache index, to avoid looking in files already looked
 * @param result Set with the result of the search
 */
public void search(byte[][] wordkey, IIndex cache, SortedSet<Pair<String,
    Integer>> result) {
 Iterator<String> it = null;
 try {
  it = myBucket.keys().iterator();
 } catch (RiakException e) {
  e.printStackTrace();
 }
 if (it != null) {
  while (it.hasNext()) {
   String keyToLook = it.next();
   if (!cache.hasKey(keyToLook)) {
    CloudObject co = get(keyToLook);
    int numberOfResults = isc.count(co.getMetadata().getData(), wordkey);
    result.add(new Pair<String, Integer>(co.getObjName(), numberOfResults));
   }
  }
 }
}
```

Listagem 4.4: Código correspondente à interface do módulo Search Security

fim, um conjunto ordenado (`result`) que irá retornar o resultado da pesquisa, de acordo com o número de ocorrências encontrado nos metadados de cada objeto armazenado. O próprio objeto `ISearchCipher` é passado como argumento no construtor, tanto da cache (`HashMapIndex`), como do Riak (`RiakTStratusIndex`).

Por fim, o método `wordkey`, como mencionado, faz a computação de uma `String` num *hash*, com base numa chave secreta. Já o `transform` efetua, não só o método `wordkey`, como as restantes transformações necessárias à cifra do *plaintext* em questão.

## 4.5 Armazenamento seguro e confidencial dos dados nas *clouds*

Para garantir que um ataque à infraestrutura onde os ficheiros se encontram armazenados não compromete a confidencialidade dos mesmos, foi adotado o algoritmo AES (do provedor *SunJCE*). Assumindo uma chave de 256 bits, é impraticável nos dias de hoje um ataque do tipo força bruta com sucesso, em tempo útil. Posto isto, e derivado de uma quebra de performance pouco significativa em relação ao uso de uma chave de 256 bits em vez de 128 bits, optou-se pelo uso da primeira, dado que o ganho a nível de segurança é bastante superior, e não é expectável que a performance seja significativamente afetada. Este algoritmo é atualmente o standard adotado pelo *National Institute of Standards and Technology* (NIST) [Bur03] e não existem ataques conhecidos com sucesso. Não obstante, o uso de um algoritmo por si só não garante total segurança, nomeadamente, quando se

tratam de ataques por criptanálise, através da observação de padrões nos *ciphertext* gerados. Para tal, é importante a escolha do modo de cifra de blocos usado, associado ao algoritmo criptográfico. Na presente dissertação, é usado o *Cipher Block Chaining* (CBC). Este, aceita como parâmetro inicial um conjunto de *bits*, denominado vetor de inicialização, que irá ser usado na geração do primeiro bloco, por meio de uma função que recebe o primeiro conjunto de dados e o vetor de inicialização e devolve o valor resultante. Por sua vez, o bloco seguinte será gerado a partir do anterior, por meio dessa mesma função, e assim sucessivamente. Desta forma, para um mesmo *plaintext* é sempre gerado um *ciphertext* diferente. A interface que expõe os métodos disponíveis pelo componente de armazenamento seguro dos dados é apresentada na Listagem 4.5. Esta apresenta apenas

```
1 package tstratus.cryptosystem;
2
3 public interface ICloudCipher {
4
5  public byte[] encryptData(byte[] plaintext);
6
7  public byte[] decryptData(byte[] ciphertext);
8 }
```

Listagem 4.5: Código correspondente à interface do módulo Cloud Security

dois métodos. O método `encryptData` que, dado um conjunto de *bytes* (*plaintext*), devolve o respetivo *ciphertext* e o método `decryptData` que, dado um conjunto de *bytes* correspondente a um *ciphertext*, devolve o respetivo *plaintext*.

## 4.6 Integridade da informação armazenada

Como prova de integridade sobre os dados armazenados foi usado o algoritmo SHA-512. Este faz parte dos standards atuais e permite obter com rapidez e performance um *digest* que permite identificar unicamente um conjunto de bytes. A não adoção do SHA-1 como algoritmo usado para este efeito recai sobretudo sobre a sua vulnerabilidade, pois são conhecidos ataques com sucesso ao mesmo. Não obstante, a família de algoritmos SHA-2, nomeadamente, o SHA-512, apresenta resultados bastante bons, com performance bastante próxima do SHA-1 e, por vezes, superior ao SHA-256, partindo do pressuposto que o sistema é executado numa plataforma com arquitetura de 64 bits [GJW11], como é o caso. Após o calculo deste *digest* para o conjunto de dados de um ficheiro e para cada um dos seus fragmentos, a informação é armazenada no índice, juntamente com a restante informação associada ao ficheiro em questão, contida no respetivo objeto. Este módulo apresenta apenas dois métodos caraterísticos, como é possível verificar na Figura 4.1. O método `getHash` permite obter uma representação (sob um `byte[]`) do resultado do *hash* do conjunto de *bytes* em questão. Já o `getHashHex` é idêntico, com a diferença de que o valor retornado corresponde à representação hexadecimal do *hash* calculado. Por fim, o método `match` permite comparar um dado *digest*, já calculado, com um `byte[]`

```
1  /**
2   * Constructor
3   */
4  public Sha2() {
5   try {
6    hash = MessageDigest.getInstance("SHA-512", "BC");
7   } catch (NoSuchAlgorithmException e) {
8    e.printStackTrace();
9   } catch (NoSuchProviderException e) {
10   e.printStackTrace();
11  }
12 }
```

Listagem 4.6: Instanciação do algoritmo usado no Integrity Module

que representa os dados propriamente ditos. A Listagem 4.6 mostra a instanciação do algoritmo onde o provedor usado é o *BouncyCastle*.

## 4.7 Compressão da informação

Para a compressão dos dados, foi usado um algoritmo bastante conhecido que é o ZIP. A principal motivação é, dados os custos associados ao armazenamento dos dados nas várias *clouds* de armazenamento existentes, poder obter alguma redução com a compressão dos mesmos, ainda que isso signifique sacrificar alguma performance de todo o sistema. Apesar disso, este é um processo relativamente rápido e não se prevê que o mesmo vá constituir um problema na performance das operações de `put` e `get` do *middleware*. A classe que implementa estas operações de compressão e descompressão encontra-se na Listagem 4.7 e implementa a interface `ICompression` (Figura 4.1). À semelhança do módulo `Cloud Security`, apresenta apenas dois métodos, que são auto-explicativos. Para a escrita e leitura dos dados comprimidos/descomprimidos foram usadas as classes `GZIPInputStream` e `GZIPOutputStream`, do próprio Java (`java.util.zip.GZIPInputStream`).

## 4.8 Informação armazenada no índice

A informação armazenada no índice encontra-se num objeto do tipo `CloudObject` (Listagem 4.8).

Este possui um nome (`objName`), uma prova de integridade sobre o conjunto total de dados que compõem o ficheiro (`digest`), um conjunto de metadados associados (`metadata`), o tamanho do ficheiro em bytes (`objSize`) e um `LinkedHashMap` onde a informação acerca dos fragmentos se encontra armazenada. Neste último, a chave corresponde à representação hexadecimal do `digest` do fragmento em questão e o valor à sua representação em `byte[]`. Optou-se por esta estrutura devido a manter a ordem pela qual os fragmentos são inseridos, pois tal é necessário para posterior obtenção e concatenação pela ordem correta, de modo a obter o ficheiro final.

```
1  /**
2   * Decompress a given compressed data
3   * @param data data to decompress
4   * @return decompressed data
5   */
6  public byte[] decompress(byte[] data) {
7   ByteArrayOutputStream out = new ByteArrayOutputStream();
8   try {
9    IOUtils.copy(new GZIPInputStream(new ByteArrayInputStream(data)),
10     out);
11  } catch (IOException e) {
12   throw new RuntimeException(e);
13  }
14  return out.toByteArray();
15 }
16
17 /**
18  * Compress a given data
19  * @param data data to compress
20  * @return compressed data
21  */
22 public byte[] compress(byte[] data) {
23  ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
24  try {
25   GZIPOutputStream gzipOutputStream = new GZIPOutputStream(
26     byteArrayOutputStream);
27   gzipOutputStream.write(data);
28   gzipOutputStream.close();
29  } catch (IOException e) {
30   throw new RuntimeException(e);
31  }
32  return byteArrayOutputStream.toByteArray();
33 }
34 }
```

Listagem 4.7: Código correspondente à interface do Compression Module

Como mencionado no capítulo anterior, existe um componente responsável pela extração dos metadados de um dado ficheiro genérico. Essa extração é feita através da interface `BasicFileAttributeView`[3], disponível desde o Java 1.7 e que veio facilitar bastante este processo. Aqui, são extraídos: o nome do ficheiro; o seu tipo ou extensão; a sua data de criação; o seu tamanho em bytes; a data do último acesso e a data da última modificação. Estes nomes de atributos correspondem a chaves no mapa de dispersão devolvido pela função `getEncMetadata` e armazenado no objeto `IObjectMetadata`. Como valor desse mapa de dispersão encontra-se o valor associado ao atributo/chave em questão, cifrado de acordo com o esquema homomórfico referido para as pesquisas (Pesquisa Linear [FD12]). Os metadados associados a cada ficheiro ou, mais propriamente, a cada objeto `CloudObject`, constituem duas operações, descritas pela interface 4.9.

Aqui, é possível verificar a existência de duas operações distintas. Ambas retornam o mesmo resultado, sendo a única diferença que uma trabalha sobre os metadados já

[3] http://docs.oracle.com/javase/7/docs/api/java/nio/file/attribute/BasicFileAttributeView.html acedido e verificado a 22-03-2013

```java
public class CloudObject implements Serializable {

 private static final long serialVersionUID = 7185779507745208745L;

 /**
  * File name, without format
  */
 private String objName;

 /**
  * File digest
  */
 private byte[] digest;

 /**
  * Metadata object for this file
  */
 private IObjectMetadata metadata;

 /**
  * Fragments and respective digest
  */
 private LinkedHashMap<String, byte[]> fragments;

 /**
  * Object size in bytes
  */
 private int objSize;

 /**
  * Fragment size in bytes
  */
 private int fragmentSize;
```

Listagem 4.8: Código correspondente à informação armazenada no CloudObject

extraídos de um ficheiro, onde a chave representa o nome do atributo (por exemplo, "Tamanho do ficheiro") e o valor representa o valor do mesmo, e outra recebe os dados do ficheiro propriamente dito e efetua a extração dos metadados. Ambas retornam um objeto do tipo `IObjectMetadata` que contém um `HashMap<String,byte[]>` onde a chave corresponde na mesma ao nome do atributo, enquanto que o valor corresponde à cifra do mesmo, através do módulo `Search Security`.

## 4.9 O módulo principal T-STRATUS

Por fim, é este o módulo que agrega todos os restantes e efetua as operações para as quais disponibiliza a API descrita na secção 3.1.6. Na secção 3.1 foram descritas as várias operações, com os módulos responsáveis pela execução das mesmas.

Aqui, apresentam-se as quatro operações principais num âmbito mais descritivo de acordo com a sua implementação propriamente dita. A Operação 1 descreve um `put`.

```java
public interface IMetadataRetriever {

 /**
  * Retrieves the metadata from a given file and encrypts it
  * @param name Name of the file
  * @param fileBytes content of the file
  * @return encrypted metadata of the file
  * @throws IOException
  */
 public IObjectMetadata getEncMetadata(String name, byte[] fileBytes) throws
     IOException;

 /**
  * Given some unencrypted metadata, encrypts it
  * @param metadata unencrypted
  * @return encrypted data
  */
 public IObjectMetadata getEncMetadata(HashMap<String, String> metadata);
}
```

Listagem 4.9: Código correspondente aos metadados armazenados para cada ficheiro

**Operação 1** Operação de PUT

**procedure** PUT($key, value$)
  $metadata \leftarrow getMetadata(key, value)$
  $size \leftarrow length(value)$
  $co \leftarrow newCloudObject(key, value, size)$
  $co \leftarrow setMetadata(metadata)$
  $digest \leftarrow getHash(value)$
  $numberOfBlocks \leftarrow smallestIntegerValueGreaterThan(size/blockSize)$
  $digest \leftarrow getHash(value)$
  $bytesRead \leftarrow 0$
  **for** $i \leftarrow 0, numberOfBlocks$ **do**
    **if** $bytesRead + blockSize > length(value)$ **then**
      $b \leftarrow value[bytesRead..(length(value) - bytesRead)]$
    **else**
      $b \leftarrow value[bytesRead..(length(value) - blockSize)]$
    **end if**
    $compressedData \leftarrow compress(b)$
    $encryptedData \leftarrow encrypt(compressedData)$
    $blockDigest \leftarrow getHash(encryptedData)$
    $blockName \leftarrow getHexadecimal(blockDigest)$
    $sendToBizQuorum(blockName, encryptedData)$
    $co \leftarrow addBlockToObject(blockName, blockDigest)$
    $start \leftarrow start + blockSize$
  **end for**
  $storeInCache(key, co)$
  $storeInRiak(key, co)$
**end procedure**

Inicialmente, são obtidos os metadados do ficheiro. De seguida, é calculado o tamanho do mesmo e é criado o `CloudObject` associado, com a informação já disponível. O número de blocos é calculado através da variável `blockSize` passada como parâmetro de configuração do sistema. Para cada bloco, são obtidos os respetivos dados e são efetuadas as operações de compressão, encriptação, geração de *digest* e envio para o quórum bizantino. Esta sequência de operações deve-se ao facto de a compressão não constituir propriedades determinísticas, logo convém que seja efetuada primeiro. A obtenção do *digest* em último permite que seja a primeira a ser verificada num `get`, evitando computações desnecessárias no caso do fragmento ser descartado pelos dados se encontrarem corrompidos. No final, a informação dos vários fragmentos fica associada ao `CloudObject` criado, numa estrutura de dados própria, como visto em 4.8, e são atualizados a cache e o índice Riak.

De seguida, para uma operação de `get` (Operação 2) é feita a verificação/pesquisa na cache e, se necessário (ou seja, se não existir na mesma), no índice Riak. Em caso de sucesso, o conjunto de fragmentos é percorrido. Começa-se por verificar a integridade do fragmento em questão, pela razão mencionada acima e em caso de sucesso realizam-se as operações de `put` pela ordem inversa, desencriptação e descompressão. No final, depois de verificada a integridade de todo o ficheiro, com a confirmação de que todos os fragmentos constituintes se encontram presentes, este é devolvido com sucesso.

**Operação 2** Operação de GET

```
1:  procedure GET(key)
2:      if cacheContains(key) then
3:          cloudObject ← getFromCache(key)
4:      else
5:          cloudObject ← getFromRiak(key)
6:      end if
7:      result ← createByteBufferWithSize(getSize(cloudObject))
8:      for all (blockName, blockDigest) ∈ getFragments(cloudObject) do
9:          b ← retrieveFromBizQuorum(blockName)
10:         if checkIntegrityMatch(b, blockDigest) then
11:             decryptedData ← decrypt(b)
12:             decompressedData ← decompress(decryptedData)
13:             result ← result + decompressedData
14:         else
15:             return ← error
16:         end if
17:     end for
18:     if checkIntegrityMatch(getDigest(cloudObject), result) then
19:         return ← result
20:     else
21:         return ← error
22:     end if
23: end procedure
```

A operação de `remove` (Operação 3) consiste em, à semelhança do `get`, procurar inicialmente pelo ficheiro na cache e, em caso de insucesso, no índice Riak. Partindo do pressuposto que é encontrado, é extraído o conjunto dos vários fragmentos associados ao mesmo e, para cada um deles, é enviado um pedido de remoção ao quórum bizantino. No final, o par chave-valor é removido do índice Riak e da cache, caso exista na mesma.

**Operação 3** Operação de REMOVE

```
1:  procedure REMOVE(key)
2:      if cacheContains(key) then
3:          cloudObject ← getFromCache(key)
4:      else
5:          cloudObject ← getFromRiak(key)
6:      end if
7:      for all (blockName, blockDigest) ∈ getFragments(cloudObject) do
8:          removeFromBizQuorum(blockName)
9:      end for
10:     removeFromCache(key)
11:     removeFromRiak(key)
12: end procedure
```

Por fim, temos a operação de `search` (Operação 4). Esta, é a única que não acede às *clouds* através do protocolo de tolerância a falhas bizantinas. As pesquisas são realizadas diretamente sobre o índice e a cache. Inicialmente, os dados da pesquisa são encriptados para que possam ser comparados com os metadados dos vários ficheiros (já mantidos de forma cifrada). Esta verificação é feita para todos os ficheiros existentes na cache e vai sendo construído um conjunto ordenado de forma descendente de pares <nome do ficheiro, número de ocorrências>. De seguida a mesma verificação é feita para todos os ficheiros existentes no índice Riak (com exceção daqueles já pesquisados na cache). No final, é obtida uma lista de ficheiros, cujo tamanho está de acordo com um parâmetro passado inicialmente para o sistema, que estipula o número máximo de ocorrências que se pretende adquirir numa pesquisa (`numberOfResults`). A diferença em relação ao `searchretrieve` é numa operação de `get` para cada ficheiro obtido, para um máximo de `numberOfResults` (resultados que se pretendem obter).

**Operação 4** Operação de SEARCH

```
1: procedure SEARCH(query)
2:     encryptedQuery ← encryptQuery(query)
3:     sortedSet ← newSetof(key = String, value = Integer)sortedbyvalue
4:     sortedSet ← sortedSet + searchCache(query)
5:     sortedSet ← sortedSet + searchRiak(query)
6:     result ← newList(numberOfResults)
7:     for i ← 0, numberOfResults do
8:         result ← result + removeFirst(sortedSet)
9:     end for
10:    return ← result
11: end procedure
```

## 4.10 Cliente Java para acesso à API do *middleware*

Como referido no capítulo 3, existem três tipos de modelos de interação com o sistema. Estes correspondem à variante local, variante *proxy* e variante *cloud*. No fundo, um deles utiliza diretamente a API do *middleware*, enquanto que os dois últimos acedem às varias operações remotamente, através de *web services*. No entanto, nenhum dos dois clientes disponíveis possui interface gráfica, sendo todas as operações invocadas a partir da linha de comandos, pois tal não se enquadrava no âmbito da presente dissertação. Na tabela 4.1 encontram-se os comandos necessários à invocação das 5 operações existentes. De notar que uma operação como o `put` parte do pressuposto que o utilizador está a indicar um caminho para o ficheiro através da diretoria corrente, que pode ser parametrizada no ficheiro de configuração da T-Stratus. Por outro lado, um `get` ou um `searchretrieve` também grava os ficheiros obtidos numa diretoria previamente parametrizada pelo utilizador.

| Operação | Invocação |
|---|---|
| **GET** | `get <nome do ficheiro>` |
| **PUT** | `put <nome do ficheiro>` |
| **REMOVE** | `remove <nome do ficheiro>` |
| **SEARCH** | `search <palavras chave (pc) separadas por espaço>` |
| **SEARCHRETRIEVE** | `searchretrieve <pc separadas por espaço>` |

Tabela 4.1: Invocação das operações disponíveis pela API do sistema

Por fim, a implementação do cliente remoto, bem como do servidor T-Stratus que aguarda pedidos do mesmo, foi concretizada através de uma framework de serviços REST, disponível em Java, denominada Restlet[4]. Esta framework permite definir um

[4] `http://restlet.org/` acedido e verificado a 22-03-2013

cliente e um servidor REST por meio de anotações específicas no código, nomeadamente nos métodos disponibilizados. Estas anotações podem ser de três tipos: `@Get`, `@Put` ou `@Remove` e os métodos disponibilizados do lado do servidor têm de corresponder exatamente àqueles existentes na interface que o cliente possui do seu lado, com as respetivas anotações.

# 5

# Avaliação

Neste capítulo é feita a avaliação ao sistema implementado, no que diz respeito ao seu desempenho e garantias de segurança que oferece. Inicialmente, é feita uma comparação entre o uso direto das várias *clouds* públicas de armazenamento geridas por terceiros, não necessariamente seguras e confiáveis, face ao uso do *middleware* como intermediário na gestão de ficheiros armazenados de forma segura na *cloud*, tendo em mente as propriedades de segurança e privacidade dos dados que oferece. São analisadas as funcionalidades de inserção, obtenção e remoção de ficheiros. De seguida, é feito um estudo do impacto no sistema causado pela alteração de métricas como o tamanho dos fragmentos em que um objeto é repartido, bem como a análise do tempo de processamento consumido pelos vários componentes, face ao tempo total de execução de cada operação. Por fim, é analisado o impacto que o uso da cache tem no sistema face ao índice Riak [Ria], bem como o impacto das pesquisas com o aumento do número de ficheiros existentes no sistema.

## 5.1 Ambiente de testes

Nos testes realizados, o sistema *middleware*, bem como os quatro servidores bizantinos, são executados na mesma máquina, com as seguintes características:

- Sistema operativo: Microsoft Windows 8 Professional
- Tipo de arquitetura: 64 bit
- Processador: Intel Core2 Quad Q6600 @ 3.00GHz (Overclocked)
- Número de cores físicos: 4
- Memória Física (RAM): 4,00 GB
- Disco Rígido: 160 GB, 7200 rpm, interface SATA

Cada um desses servidores efetua escritas e leituras para uma, e apenas uma, das *clouds* adotadas. Não existem, no entanto dois servidores a escrever ou ler para a mesma. À exceção da Dropbox, todas as *clouds* adotadas permitem a escolha da localização onde queremos que a nossa informação esteja alojada e para onde os pedidos são encaminhados. Tendo este aspeto em vista, foi escolhida a localização mais próxima do utilizador de teste, encontrando-se o mesmo em Portugal. A instância da Amazon S3 situa-se na Irlanda, assim como a da Google Cloud Storage. Já a Lunacloud possui um centro de dados em Portugal. Por fim, quanto à Dropbox, não foi encontrada informação a respeito da localização dos seus servidores, embora no final o armazenamento seja direcionado para a Amazon S3.

O *middleware*, por sua vez, comunica com a base de dados NoSQL distribuída Riak, que se encontra a ser executada num servidor na *cloud*, numa instância *Cloud Server* da Lunacloud, com as seguintes características:

- Sistema operativo: Red Hat Enterprise Linux (Debian 6.0)
- Tipo de arquitetura: 64 bit
- Processador: 1500 MHz
- Número de cores físicos: 2
- Memória Física (RAM): 4,00 GB
- Disco Rígido: 250 GB
- Largura de banda: 10240 Kbit/sec

De modo a melhor simular um ambiente distribuído, com várias instâncias do Riak, este encontra-se a ser executado com quatro nós, cada um responsável por uma dada partição das chaves existentes no sistema. Não obstante, essa partição é totalmente transparente para o utilizador, bem como as aplicações que fazem uso deste repositório do tipo chave-valor. Os vários nós executam na mesma máquina, descrita acima, e constituem o cluster para onde os pedidos são encaminhados. O cliente é executado localmente, na mesma máquina do *middleware*, numa rede com largura de banda de 30 Mbps de download e 3 Mbps de upload. O cenário de avaliação, com base nos recursos mencionados, é aquele descrito na Figura 3.2, como um serviço instalado num servidor dedicado, próximo do utilizador. No entanto, o cliente não é remoto, efetuando pedidos diretamente ao sistema, sem a latência associada à comunicação utilizador-sistema.

Por fim, para uma melhor visualização dos dados obtidos, foi utilizada uma escala logarítmica na grande maioria dos gráficos. Não obstante, todos se fazem acompanhar de uma tabela onde é possível verificar os valores medidos com exatidão. Esta decisão partiu da necessidade de visualizar a evolução das diversas métricas, tanto para valores muito pequenos, como para valores elevados. Sem recorrer a este tipo de escala, esta observação torna-se menos percetível.

## 5.2 Operações diretas na *cloud*, sem fragmentação

Neste conjunto de testes iniciais (três subsecções seguintes), são comparadas as quatro *clouds* entre si, para as três operações principais, inserção, obtenção e remoção. O objetivo deste teste é, não só ter uma ideia da variação do tempo em função do tamanho do ficheiro manipulado, para cada uma das *clouds* utilizadas, como obter um conjunto de valores referência que possam ser, mais tarde, comparados com os resultados obtidos com o uso do *middleware*.

### 5.2.1 `Put` de ficheiros diretamente para a *cloud*

Neste primeiro conjunto de testes, são calculados os tempos de inserção de um conjunto de bytes (dados de um ficheiro) para as diferentes *clouds* de armazenamento de domínio público usadas.

| Tamanho do ficheiro | Amazon S3 | Google Cloud Storage | Dropbox | Lunacloud |
|---|---|---|---|---|
| **0,01 MB** | 0,578 | 2,667 | 1,011 | 2,303 |
| **0,1 MB** | 1,466 | 3,445 | 2,055 | 3,186 |
| **1 MB** | 12,631 | 12,432 | 14,593 | 5,44 |
| **10 MB** | 138,021 | 89,674 | 115,152 | 33,843 |
| **30 MB** | 341,138 | 121,841 | 364,7 | 108,09 |

Tabela 5.1: Tempos medidos (em segundos) para operações de escrita de ficheiros de diferentes tamanhos nas várias *clouds* utilizadas

Na Tabela 5.1 encontram-se os resultados obtidos (em segundos) para a inserção de ficheiros de diferentes tamanhos para as várias *clouds* usadas. Foram realizadas 10 inserções para cada tamanho de ficheiro (para cada *cloud*) e calculada a média dos tempos obtidos. O gráfico da Figura 5.1 ilustra estes dados, de onde podemos tirar algumas ilações.

Em primeiro lugar, e como seria expectável, verifica-se um aumento gradual dos tempos medidos, com o aumento do tamanho do ficheiro, que requer a transmissão de dados através da rede. Quanto maior o volume de dados, mantendo as características dos canais de comunicação, maior o tempo necessário para transferir os mesmos. No entanto, verifica-se a existência de uma variação relativamente aos diferentes tamanhos de ficheiros para a mesma *cloud*, onde tempos melhores registados para tamanhos mais baixos (comparativamente com as restantes *clouds*) não significam igual relação à medida que o tamanho dos mesmos aumenta. Ou seja, podemos verificar que, para tamanhos relativamente baixos, como $0,01$ MB e $0,1$ MB, a Amazon S3 e a Dropbox apresentam melhor performance, enquanto que, para valores superiores a $1$ MB verifica-se o contrário, estando a Lunacloud e a Google Cloud Storage com as melhores medições registadas. Por fim, para valores superiores a $1$ MB verificamos que a Lunacloud se distancia das outras

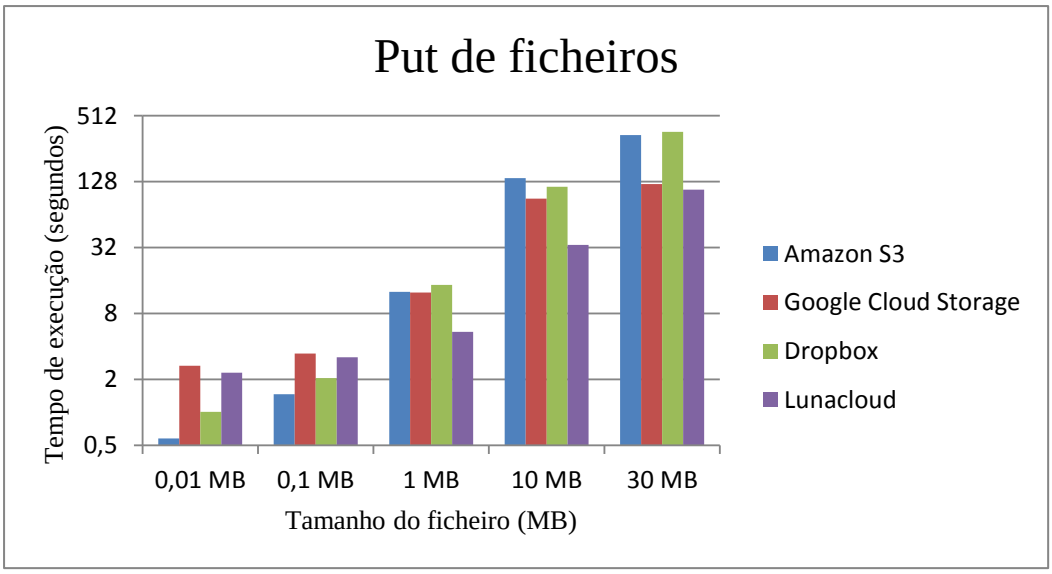



Figura 5.1: Gráfico correspondente aos valores da Tabela 5.1

consideravelmente, voltando a estar bastante próxima da Google Cloud Storage para ficheiros de 30 MB. Esta discrepância pode influenciar fortemente a escolha do tamanho dos fragmentos a adotar (parametrizar) no sistema, de forma a maximizar a performance da operação. Por fim, a elevada performance da Lunacloud (notória para ficheiros maiores) poderá estar associada à sua localização privilegiada (Portugal), comparativamente com as restantes.

### 5.2.2 `Get` de ficheiros diretamente da *cloud*

Após a comparação das escritas para as quatro *clouds* usadas, foi testada a performance das mesmas para operações de leitura de ficheiros armazenados. Na Tabela 5.2 encontram-se os valores medidos, em segundos. Para a obtenção dos mesmos, foi novamente feita a média de 10 pedidos `get` para cada tamanho considerado (para cada *cloud*), à semelhança do que foi feito para a operação de `put`. Os dados apresentados correspondem à média desses pedidos.

| Tamanho do Ficheiro | Amazon S3 | Google Cloud Storage | Dropbox | Lunacloud |
|---|---|---|---|---|
| **0,01 MB** | 0,308 | 0,394 | 0,676 | 2,674 |
| **0,1 MB** | 1,388 | 0,493 | 1,318 | 4,363 |
| **1 MB** | 10,194 | 3,625 | 3,018 | 5,638 |
| **10 MB** | 65,159 | 18,588 | 12,502 | 17,877 |
| **30 MB** | 217,666 | 34,221 | 35,77 | 64,538 |

Tabela 5.2: Tempos medidos (em segundos) para operações de leitura de ficheiros de diferentes tamanhos nas várias *clouds* utilizadas

O gráfico da Figura 5.2, correspondente aos valores da Tabela 5.2, permite verificar

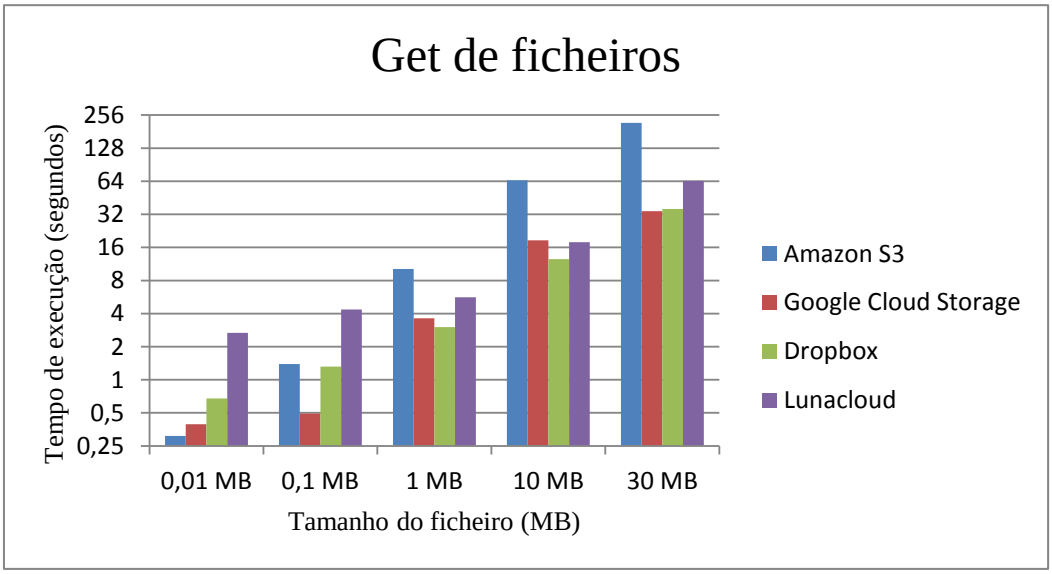



Figura 5.2: Gráfico correspondente aos valores da Tabela 5.2

que o comportamento das várias *clouds* difere em relação ao observado para a operação de `put`. Para tamanhos muito pequenos, a Amazon S3 continua a registar o menor tempo relativamente às restantes. Já para valores superiores ou iguais a 1 MB, verificam-se tempos consideravelmente superiores, onde se destaca claramente das restantes. A Google Cloud Storage e a Dropbox mantêm-se muito próximas para valores superiores ou iguais a 1 MB, sendo que, para valores inferiores, verifica-se o inverso do `put`, onde desta vez, a Dropbox tem clara superioridade sobre a Google Cloud Storage. Para tamanhos de 30 MB a Lunacloud já não domina como no `put`. Verifica-se, no geral, que para esta operação de leitura, a *cloud* que apresenta a melhore performance é a Google Cloud Storage. Aqui, a distância pode não representar um papel tão importante como no caso anterior, visto que a largura de banda disponível (*download*) é muito superior (em relação ao *upload*). Uma hipótese é que, embora uma ligação possa estar próxima em termos de distância, se os canais de comunicação não tiverem capacidade suficiente (inferior à capacidade máxima da rede), tornam-se um *bottleneck* no sistema. Assim, é possível que distâncias superiores apresentem, no final, tempos inferiores. Não obstante, os próprios servidores das *clouds* podem diferir na capacidade de resposta, bem como no tempo de processamento. Para a Lunacloud, por exemplo, verificou-se um tempo de resposta superior para tamanhos inferiores a 1 MB, inclusive. Uma hipótese é que este facto possa estar relacionado com a capacidade de resposta ou processamento por parte do próprio servidor. A partir de 1 MB, à medida que o volume de dados a receber aumenta, o tempo de transmissão já se sobrepõe à capacidade ou processamento do servidor.

### 5.2.3 `Remove` de ficheiros diretamente na *cloud*

Por fim, foram analisados os pedidos de remoção de um ficheiro. Os resultados obtidos (em segundos) encontram-se na Tabela 5.3 e no gráfico da Figura 5.3, sendo que, para este teste, também foram realizados 10 pedidos para cada tamanho e *cloud* considerado.

| Tamanho do Ficheiro | Amazon S3 | Google Cloud Storage | Dropbox | Lunacloud |
|---|---|---|---|---|
| **0,01 MB** | 0,263 | 0,486 | 0,51 | 9,143 |
| **0,1 MB** | 0,255 | 0,495 | 0,543 | 10,317 |
| **1 MB** | 0,263 | 0,453 | 0,503 | 9,532 |
| **10 MB** | 0,267 | 0,403 | 0,563 | 11,365 |
| **30 MB** | 0,253 | 0,389 | 0,632 | 11,668 |

Tabela 5.3: Tempos medidos (em segundos) para operações de remoção de ficheiros de diferentes tamanhos nas várias *clouds* utilizadas

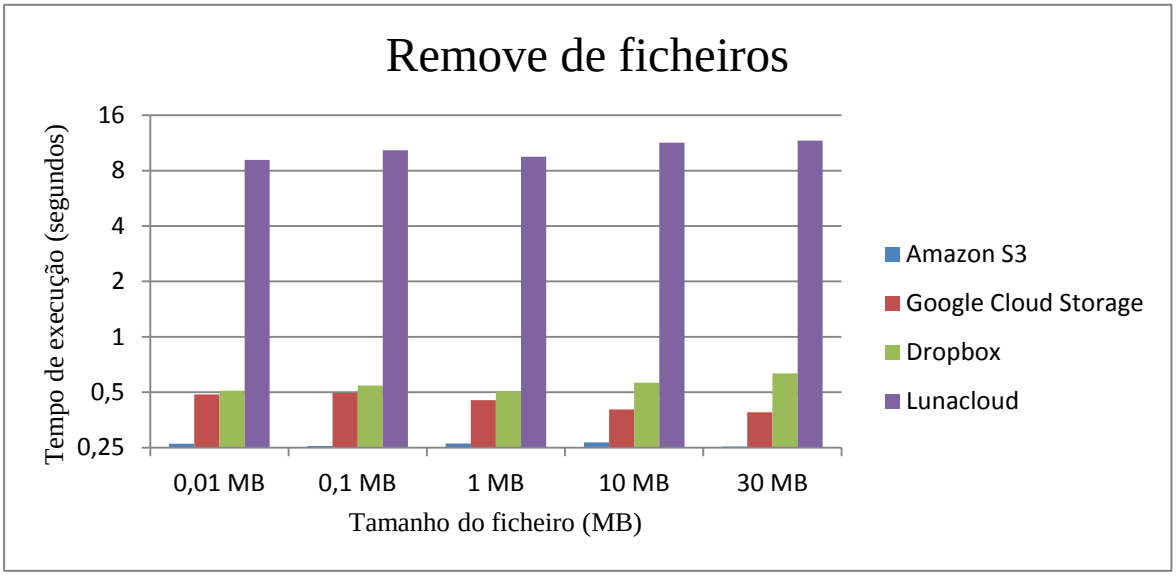



Figura 5.3: Gráfico correspondente aos valores da Tabela 5.3

É possível concluir que esta operação, do ponto de vista do utilizador, é independente do tamanho do ficheiro a considerar. O pedido é enviado para a *cloud*, e a mesma devolve a confirmação, ainda que, por hipótese, a remoção do ficheiro possa ficar a ser executada em *background*. Só assim é possível explicar o facto da discrepância ser praticamente nula entre a remoção de um ficheiro de $0,01$ MB e um de $30$ MB. É de notar a enorme rapidez que se verifica para a Amazon S3, em contraste com a discrepância verificada para a Lunacloud, que apresenta valores bastante superiores quando comparada com as restantes. Uma vez mais, pensa-se que esta diferença possa estar associada à própria capacidade de resposta, ou processamento, do servidor.

## 5.3 Integridade dos ficheiros armazenados

A escolha da função segura de *hash* para a geração das provas de integridade sobre os ficheiros foi justificada no Capítulo 4, implementação. Como mencionado, são conhecidos ataques com sucesso ao SHA-1, vindo o mesmo a ser substituído pela família de funções SHA-2, que inclui diferentes tamanhos para os *digest* gerados. Não obstante, o sucessor do SHA-2 já é conhecido atualmente (Keccak[1], selecionado como o novo standard SHA-3), apesar do SHA-2 não apresentar falhas a nível de segurança que justifiquem a sua descontinuidade. No teste seguinte, o objetivo foi o de comparar os vários algoritmos, nas suas versões Java, incluindo uma implementação existente do, mais recente, SHA-3[2]. Desta forma, podemos ter uma perceção melhor do tempo de processamento de cada um para diferentes tamanhos de ficheiros, de modo a avaliar a escolha do algoritmo efetuada. Na Tabela 5.4, apresentam-se os tempos medidos (em segundos) para a operação de geração de um *digest* para sete tamanhos diferentes e quatro funções de *hash*, duas das quais da família SHA-2.

| | **SHA1** | **SHA256** | **SHA512** | **SHA3 256** |
|---|---|---|---|---|
| **0,01 MB** | 0 | 0 | 0 | 0 |
| **0,1 MB** | 0,001 | 0,001 | 0,001 | 0,001 |
| **1 MB** | 0,006 | 0,01 | 0,007 | 0,008 |
| **5 MB** | 0,032 | 0,05 | 0,034 | 0,04 |
| **10 MB** | 0,063 | 0,098 | 0,067 | 0,078 |
| **15 MB** | 0,095 | 0,146 | 0,101 | 0,117 |
| **30 MB** | 0,19 | 0,293 | 0,203 | 0,235 |

Tabela 5.4: Tempos medidos (em segundos) para o calculo do *digest* para diferentes tamanhos de ficheiros, com diferentes funções de *hash*

Cada valor da Tabela 5.4 foi obtido através da média de 100 ensaios para cada combinação ficheiro-algoritmo. Através destes, foi gerado o gráfico da Figura 5.4. A análise aos resultados obtidos permite-nos concluir que o SHA-3 ou, pelo menos, a implementação do SHA-3 considerada, pouco difere em relação aos seus antecessores, no que diz respeito ao tempo de processamento. Não obstante, há que ter em mente a diferença dos 256 para os 512 bits do SHA512. Regra geral, com exceção do SHA1 que se descarta à partida devido a problemas de segurança, qualquer um dos restantes algoritmos seria uma boa opção, visto os tempos medidos diferirem pouco entre si. Visto a máquina onde os testes estão a ser realizados possuir uma arquitetura de 64 bits, a performance do SHA512 é, em todos os casos, superior à do SHA256 e do SHA3 256. Confirma-se a escolha da utilização do SHA512, no entanto, conclui-se que facilmente se substituiria a

[1] http://keccak.noekeon.org/ acedido e verificado a 14-03-2013

[2] https://jce.iaik.tugraz.at/crm/freeDownload.php?25000 verificado e acedido a 14-03-2013

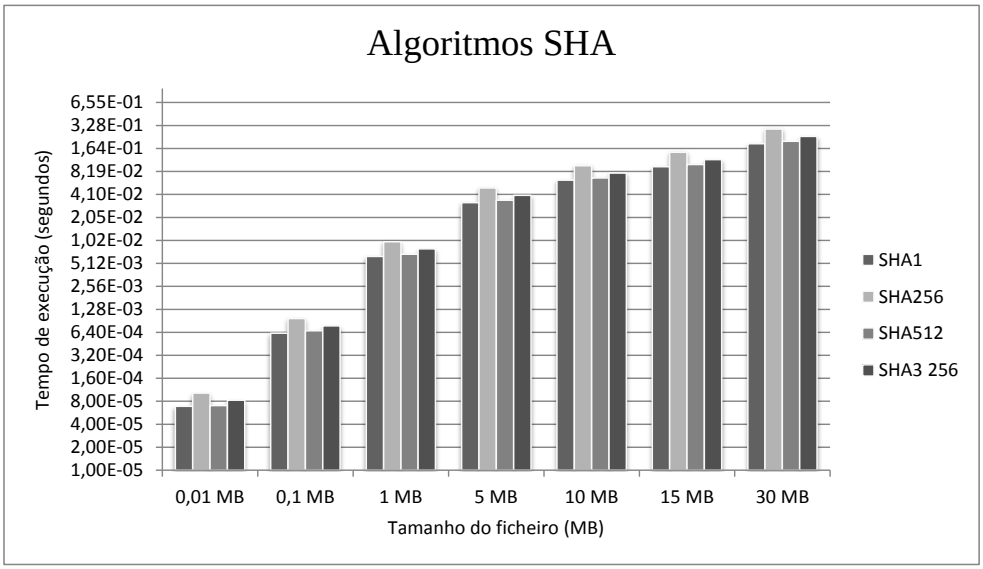



Figura 5.4: Gráfico correspondente aos valores da Tabela 5.4

função de *hash* utilizada pelo SHA3 256, sem que fosse notória uma quebra na performance do sistema. Apesar do *digest* gerado ser inferior, considera-se que um *digest* de 256 já é suficientemente grande para o efeito.

## 5.4 Algoritmo usado para a cifra/decifra dos dados

Visto a segurança e privacidade dos dados armazenados ser o principal foco deste sistema, foi feito um estudo sobre três dos principais candidatos (par algoritmo-tamanho de chave) a nível de algoritmos de cifra/decifra a usar. A escolha do AES com uma chave de 256 bits, face a uma de 128, tem a ver com o enorme ganho a nível da segurança quando comparada com o tempo de execução, que acaba por ser ligeiramente sacrificado, não assumindo variações relevantes. Com o objetivo de comprovar esta observação, foram testadas as implementações em Java dos algoritmos AES, bem como o Triple DES (ambos do provedor *SunJCE*). O DES, anterior standard, provou-se ser vulnerável nos dias de hoje. Não obstante, o Triple DES é considerado suficientemente seguro, embora com uma performance inferior à do AES. Os resultados dos testes efetuados (Tabela 5.5) para estes algoritmos em Java apresentam o comportamento esperado.

| | 3DES 192 | AES 128 | AES 256 |
|---|---|---|---|
| **0,01 MB** | 0,0011 | 0,0003 | 0,0004 |
| **0,1 MB** | 0,01 | 0,0014 | 0,0016 |
| **1 MB** | 0,1016 | 0,0136 | 0,0168 |
| **5 MB** | 0,5054 | 0,0685 | 0,0849 |
| **10 MB** | 1,0146 | 0,139 | 0,1716 |
| **15 MB** | 1,5236 | 0,2095 | 0,2584 |
| **30 MB** | 3,0598 | 0,4192 | 0,5167 |

Tabela 5.5: Tempos medidos (em segundos) para o calculo do *ciphertext* para diferentes tamanhos de ficheiros

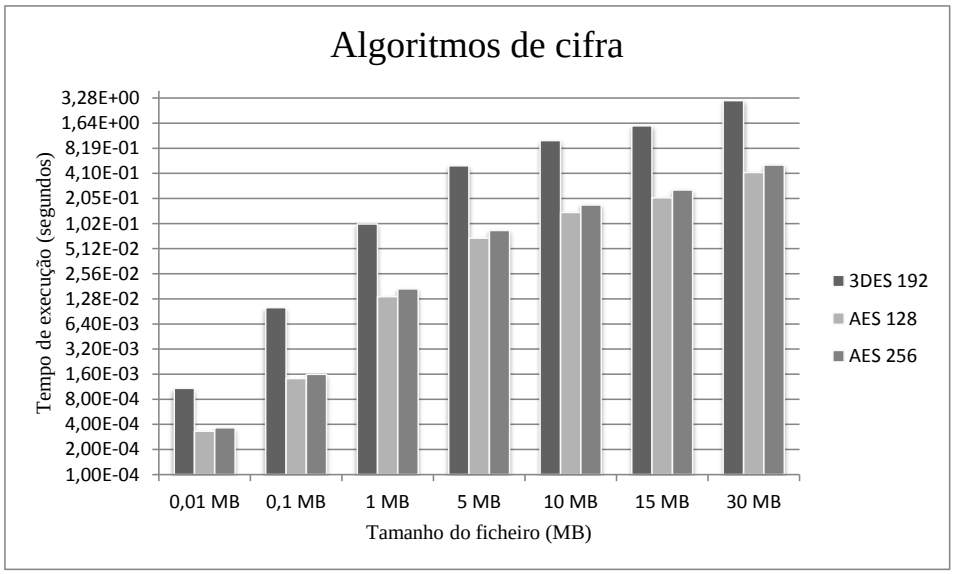



Figura 5.5: Gráfico correspondente aos valores da Tabela 5.5

Como é possível verificar através do gráfico da Figura 5.5, para qualquer tamanho de ficheiro testado, o Triple DES revela um tempo de processamento superior. Já o AES apresenta sempre valores muito próximos para as duas chaves de 128 e 256 bits. Chegam até a ser praticamente iguais para valores de $0,1$ MB. Verifica-se que o *tradeoff* é perfeitamente aceitável pois o uso de uma chave com o dobro do tamanho permite um nível de segurança bastante superior e tem como consequência um aumento do tempo de execução na ordem dos milissegundos.

## 5.5 Peso dos vários componentes do sistema

Na avaliação do sistema *middleware* implementado, e tendo em vista a comparação com a escrita e leitura direta na *cloud*, analisada na secção 5.2, é necessário verificar primeiro

o peso que os vários componentes representam nas três operações principais disponibilizadas. O objetivo dos testes na secção que se segue é verificar quais as operações predominantes, que correspondem ao maior tempo de processamento e têm maior influência no tempo total da operação em questão. Devido aos arredondamentos necessários, as percentagens medidas apresentam um erro de, no máximo, $0,02\%$. Para esta análise foram também realizadas 10 operações de `get`, `put` e `remove`, e foram medidos os tempos que cada componente consumia no total. Estes encontram-se na Tabela 5.6 onde Digest representa o tempo de processamento para o cálculo do *digest* através do SHA-512 e Metadata o tempo associado à extração e cifra dos metadados do ficheiro. As operações de Put/Get/Remove (inserção, obtenção e remoção dos dados na *cloud*), Compressão/Descompressão, Cifra/Decifra e Digest Frag são realizadas sobre os vários fragmentos sendo que, neste caso, existe apenas um. Nas várias tabelas que se seguem, as colunas Cache e Riak correspondem aos tempos medidos para a inserção/obtenção/remoção do ficheiro, respetivamente, na cache e no índice Riak.

### 5.5.1 Inserção (`put`)

| Tamanho | Digest | Metadata | Put | Compressão | Cifra | Digest Frag. | Cache | Riak |
|---|---|---|---|---|---|---|---|---|
| **0,01 MB** | 0 | 0,01 | 2,766 | 0 | 0,002 | 0,003 | 0,003 | 0,033 |
| **0,1 MB** | 0,004 | 0,018 | 3,717 | 0,006 | 0,003 | 0,003 | 0,024 | 0,068 |
| **1 MB** | 0,011 | 0,012 | 15,689 | 0,049 | 0,015 | 0,011 | 0,007 | 0,067 |
| **10 MB** | 0,097 | 0,03 | 141,47 | 0,612 | 0,153 | 0,079 | 0,023 | 0,109 |

Tabela 5.6: Tempos medidos (em segundos) para as várias operações envolvidas na escrita de ficheiros de diferentes tamanhos através do *middleware*

| Tamanho | Digest | MData | Put | Compr. | Cifra | Digest | Cache | Riak |
|---|---|---|---|---|---|---|---|---|
| **0,01 MB** | 0 | 0,35 | 98,19 | 0 | 0,07 | 0,11 | 0,11 | 1,17 |
| **0,1 MB** | 0,1 | 0,47 | 96,72 | 0,16 | 0,08 | 0,08 | 0,62 | 1,77 |
| **1 MB** | 0,07 | 0,08 | 98,92 | 0,31 | 0,09 | 0,07 | 0,04 | 0,42 |
| **10 MB** | 0,07 | 0,02 | 99,23 | 0,43 | 0,11 | 0,06 | 0,02 | 0,08 |

Tabela 5.7: Percentagem (%) de tempo ocupada pelas várias operações envolvidas na escrita de ficheiros de diferentes tamanhos através do *middleware*, com base nos valores da Tabela 5.6

Ao olhar para os valores da Tabela 5.6 é possível verificar que o grande peso e consequente *bottleneck* do sistema é a inserção dos dados nas várias *clouds*. A Tabela 5.7 permite concluir com maior clareza o peso que esta operação representa. Nesta, verificam-se as percentagens face ao tempo total despendido para a operação de `put`. Não só a operação de inserção nas *clouds* apresenta percentagens acima dos $96\%$, como a tendência que se verifica é que as restantes operações cada vez se tornam menos significativas quanto

ao tempo total despendido, à medida que o tamanho do ficheiro aumenta. O tempo despendido numa inserção na cache, bem como no índice, perde relevância, ao passo que operações mais complexas sobre conjuntos de dados, como a compressão ou a cifra, passam a ocupar uma percentagem superior com o aumento do volume dos dados. No entanto, apesar do aumento, este não é significativo face ao custo dominante da escrita na *cloud* através do protocolo bizantino.

### 5.5.2 Obtenção (`get`)

Na Tabela 5.8 são apresentados os tempos medidos, desta vez, para a obtenção de ficheiros. Aqui, uma vez mais, é verificada a grande discrepância da operação de obtenção dos fragmentos da *cloud*, face às restantes. Nestes testes, em particular, é possível verificar que a cache apresenta tempos demasiado baixos ou mesmo nulos. Neste caso, apesar da mesma ser de acesso bastante rápido, sendo esse o seu propósito, estes tempos nulos explicam-se devido à inexistência do ficheiro em questão na cache e consequente necessidade de obtenção do mesmo no índice. Ou seja, os testes realizados foram sempre sobre ficheiros que não se encontravam na cache. Não obstante, dado o peso bastante pequeno de uma leitura no índice (Tabela 5.9), ainda para mais com o aumento do tamanho do ficheiro, os valores obtidos, embora melhores, não seriam significativos face à enorme percentagem ocupada pela operação de obtenção dos dados nas *clouds* pelo protocolo bizantino.

| Tamanho | Digest | Get | Descompressão | Decifra | Cache | Riak |
|---|---|---|---|---|---|---|
| **0,01 MB** | 0 | 0,697 | 0,001 | 0,002 | 0 | 0,018 |
| **0,1 MB** | 0,001 | 1,418 | 0,001 | 0,005 | 0 | 0,015 |
| **1 MB** | 0,008 | 6,996 | 0,006 | 0,022 | 0 | 0,017 |
| **10 MB** | 0,078 | 38,798 | 0,076 | 0,178 | 0 | 0,041 |

Tabela 5.8: Tempos medidos (em segundos) para as várias operações envolvidas na obtenção de ficheiros de diferentes tamanhos através do *middleware*

| Tamanho | Digest (%) | Get (%) | Descompr. (%) | Decrifra (%) | Cache (%) | Riak (%) |
|---|---|---|---|---|---|---|
| **0,01 MB** | 0 | 97,1 | 0,1 | 0,3 | 0 | 2,5 |
| **0,1 MB** | 0,1 | 98,5 | 0,1 | 0,3 | 0 | 1 |
| **1 MB** | 0,1 | 99,3 | 0,1 | 0,3 | 0 | 0,2 |
| **10 MB** | 0,2 | 99 | 0,2 | 0,5 | 0 | 0,1 |

Tabela 5.9: Percentagem de tempo ocupada pelas várias operações envolvidas na obtenção de ficheiros de diferentes tamanhos através do *middleware*, com base nos valores da Tabela 5.8

É bem visível, novamente, o peso das operações sobre um conjunto de dados (compressão, cifra, integridade), face ao peso de operações constantes, como é o caso dos

acessos ao índice, que são diretos. No entanto, o aumento em relação ao tamanho do ficheiro não é significativo e a percentagem ocupada pela obtenção dos fragmentos é cada vez maior, representando cerca de 99% do tempo total da operação.

### 5.5.3 Remoção (`remove`)

Por fim, a remoção é aquela que menos variações regista. Tal como concluído no estudo inicial sobre as várias *clouds* de armazenamento de domínio público, esta operação é independente do tamanho do ficheiro a considerar. É possível verificar que o comportamento não se altera de forma significativa ao longo dos diferentes tamanhos, ao observar os tempos medidos na Tabela 5.10.

| Tamanho | Remove | Remove (%) | Cache | Cache (%) | Riak | Riak (%) |
|---|---|---|---|---|---|---|
| **0,01 MB** | 0,817 | **98,08** | 0,003 | **0,36** | 0,013 | **1,56** |
| **0,1 MB** | 0,603 | **96,17** | 0,011 | **1,75** | 0,013 | **2,07** |
| **1 MB** | 0,541 | **96,61** | 0,003 | **0,54** | 0,016 | **2,86** |
| **10 MB** | 0,651 | **97,46** | 0,003 | **0,45** | 0,014 | **2,1** |

Tabela 5.10: Tempos (em segundos) e percentagem medidos para as várias operações envolvidas na remoção de ficheiros de diferentes tamanhos através do *middleware*

Todas as operações que constituem um pedido `remove` são constantes. A remoção na *cloud* através do protocolo bizantino, bem como o acesso direto ao ficheiro em cache ou no índice, são sempre constantes. Por maior que seja o tamanho do ficheiro, numa remoção, há-de apresentar sempre características semelhantes, no que diz respeito aos tempos despendidos para as várias operações. Já o número de fragmentos irá certamente condicionar o tempo desta operação. Será algo a analisar e ter em conta em baixo, numa secção seguinte.

## 5.6 Operações com recurso ao *middleware* sem fragmentação

Comparativamente com os resultados anteriores, obtidos na secção 5.2, foram medidos os tempos para escritas de ficheiros, agora com a intermediação do *middleware*. No entanto, com os recursos existentes, não foi possível a realização de testes para ficheiros de 30 MB ou mais. O processamento dos vários servidores bizantinos na mesma máquina que o *middleware* constitui um entrave pois, para ficheiros de maior dimensão, este ultrapassa a capacidade de memória do computador utilizado. Dada a constatação de que a operação predominante é a escrita e leitura dos dados na *cloud* através do protocolo bizantino, a comparação com os resultados da secção 5.2 é bastante direta, dadas as mesmas condições. Este conjunto de testes é, provavelmente, o mais importante, pois tem como objetivo avaliar a viabilidade da solução proposta, com todas as propriedades de segurança inerentes, face ao uso não seguro e não confiável das *clouds* de armazenamento

de dados tal como são disponibilizadas. Só assim é possível verificar se o *tradeoff* envolvido justifica o uso do *middleware* como solução de armazenamento de dados segura e confiável.

### 5.6.1 Inserção (`put`)

Nesta subsecção, são apresentados os tempos medidos (em segundos) (Tabela 5.11) para a operação de escrita de ficheiros com a intermediação do *middleware*, até um máximo de 10 MB.

| Tamanho | Amazon S3 | Google Cloud Storage | Dropbox | Lunacloud | T-Stratus |
|---|---|---|---|---|---|
| **0,01 MB** | 0,578 | 2,667 | 1,011 | 2,303 | **2,818** |
| **0,1 MB** | 1,466 | 3,445 | 2,055 | 3,186 | **3,842** |
| **1 MB** | 12,631 | 12,432 | 14,593 | 5,44 | **15,861** |
| **10 MB** | 138,021 | 89,674 | 115,152 | 33,843 | **142,588** |

Tabela 5.11: Tempos medidos (em segundos) para operações de escrita de ficheiros de diferentes tamanhos para as quatro *clouds* e para o *middleware*

Os tempos obtidos encontram-se mapeados no gráfico da Figura 5.6, onde é possível ter uma ideia visual do impacto do uso do *middleware* face ao uso das *clouds* propriamente ditas, tal como existem atualmente.

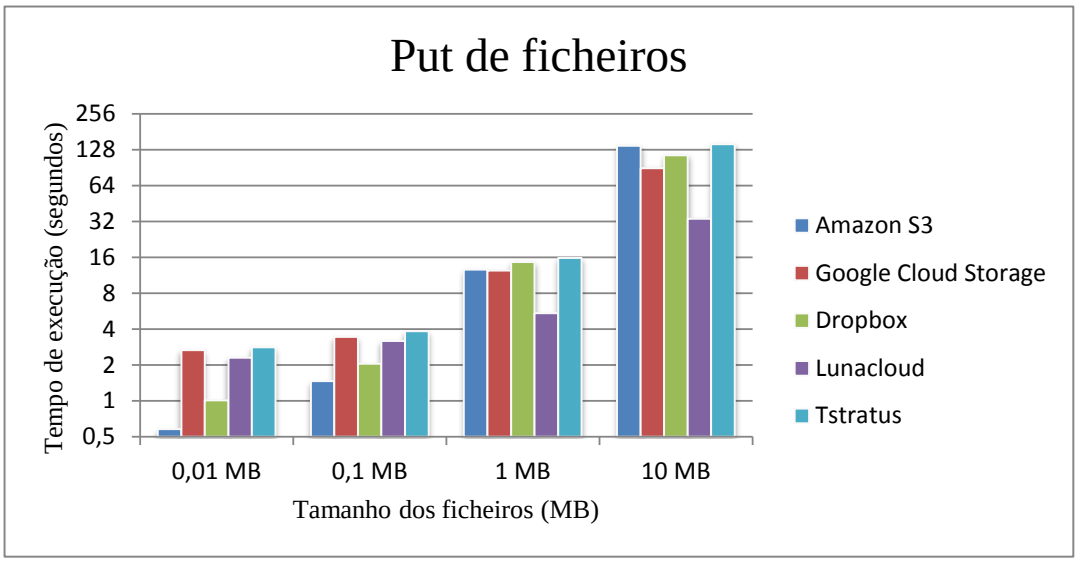



Figura 5.6: Gráfico correspondente aos valores da Tabela 5.11

Perante uma operação de escrita, a especificação da biblioteca usada (`BFTSMaRt`) garante a escrita num quórum de, pelo menos, três *clouds*. Seria expectável que o tempo de escrita com recurso ao *middleware* pudesse ser ligeiramente superior ao da escrita direta na terceira *cloud* mais lenta. Para tal, iríamos partir também da conclusão de que o grande *bottleneck* do sistema são as operações na *cloud*, tendo comprovado esse facto na

secção acima, sendo que as restantes representam uma percentagem mínima do tempo de execução do pedido. No gráfico da Figura 5.6 é possível verificar que os tempos de execução do *middleware* encontram-se bastante próximos daqueles registados para a *cloud* mais lenta. Não obstante, os tempos aqui registados também se encontram sob a influência da largura de banda disponível. Enquanto que nos testes realizados para as várias *clouds* individualmente, a largura de banda da rede é consumida somente pela *cloud* ativa naquele instante, aqui, as quatro executam em simultâneo, o que constitui uma limitação no que diz respeito à capacidade da rede disponível para cada uma. Posto isto, os valores obtidos são consideravelmente bons e diferem muito pouco em relação à pior *cloud* registada. Contudo, a capacidade de *upload* da rede de testes é demasiado limitada. Seria expectável a obtenção de tempos melhores, se a mesma fosse, por exemplo, idêntica à de *download*.

### 5.6.2 Obtenção (`get`)

Após o estudo do uso direto das quatro *clouds*, confrontámos novamente os resultados obtidos, com os tempos medidos para a obtenção de um ficheiro através do *middleware*. O resultado encontra-se na Tabela 5.12 e no gráfico da Figura 5.7.

| Tamanho | Amazon S3 | Google Cloud Storage | Dropbox | Lunacloud | Tstratus |
|---|---|---|---|---|---|
| **0,01 MB** | 0,308 | 0,394 | 0,676 | 2,674 | **0,722** |
| **0,1 MB** | 1,388 | 0,493 | 1,318 | 4,363 | **1,455** |
| **1 MB** | 10,194 | 3,625 | 3,018 | 5,638 | **7,064** |
| **10 MB** | 65,159 | 18,588 | 12,502 | 17,877 | **39,268** |

Tabela 5.12: Tempos medidos (em segundos) para operações de leitura de ficheiros de diferentes tamanhos para as quatro *clouds* e para o *middleware*

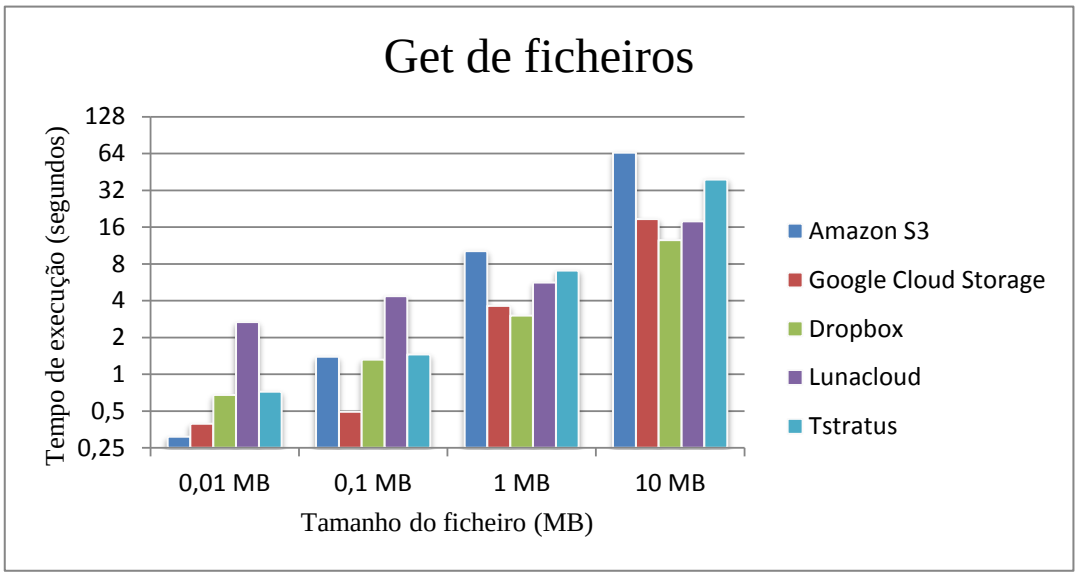



Figura 5.7: Gráfico correspondente aos valores da Tabela 5.12

Ao analisar os resultados obtidos, já se torna evidente a utilização do algoritmo de tolerância a falhas bizantinas, onde apenas um quórum de respostas determina a obtenção do ficheiro com sucesso. Como é possível verificar, ao olhar para o gráfico, a obtenção de um ficheiro por parte do *middleware* apresenta sempre valores entre o terceiro pior tempo registado para uma *cloud* e o pior. É evidente a espera de, pelo menos, três respostas com sucesso, por parte do cliente bizantino, de modo a dar como bem sucedida a operação. Um fator que também terá influenciado os resultados obtidos é a largura de banda disponível de *download* em relação à de *upload*. Apesar do uso de quatro *cloud* em simultâneo requerer uma distribuição da capacidade total da rede, visto haver uma largura de banda muito superior de *download*, a distribuição pelas quatro *clouds* tornasse menos limitativa. Já para o *upload*, a largura de banda disponível é tão baixa que acaba por condicionar mais o uso simultâneo das várias *clouds* no *middleware*. De futuro, o ideal seria ter um ambiente de testes onde a velocidade da rede fosse quatro vezes superior à atual. Desta forma, a distribuição da mesma pelos quatro conectores iria corresponder a uma velocidade mais próxima daquela disponível por cada uma das *clouds* nos testes individuais. De qualquer forma, os resultados obtidos estão dentro do intervalo expectável (entre a terceira pior e a pior *cloud*).

### 5.6.3 Remoção (`remove`)

Por fim, foi estabelecida uma comparação entre a remoção direta das *clouds* e a mesma com recurso ao *middleware*. Foram registados os tempos medidos, com o mesmo número de ensaios (10), para tamanhos de ficheiros até 10 MB. Na Tabela 5.13 encontram-se os valores medidos, mapeados no gráfico da Figura 5.8.

| Tamanho | Amazon S3 | Google Cloud Storage | Dropbox | Lunacloud | T-Stratus |
|---|---|---|---|---|---|
| **0,01 MB** | 0,263 | 0,486 | 0,51 | 9,143 | **0,812** |
| **0,1 MB** | 0,255 | 0,495 | 0,543 | 10,317 | **0,683** |
| **1 MB** | 0,263 | 0,453 | 0,503 | 9,532 | **0,588** |
| **10 MB** | 0,267 | 0,403 | 0,563 | 11,365 | **0,687** |

Tabela 5.13: Tempos medidos (em segundos) para operações de remoção de ficheiros de diferentes tamanhos para as quatro *clouds* e para o *middleware*

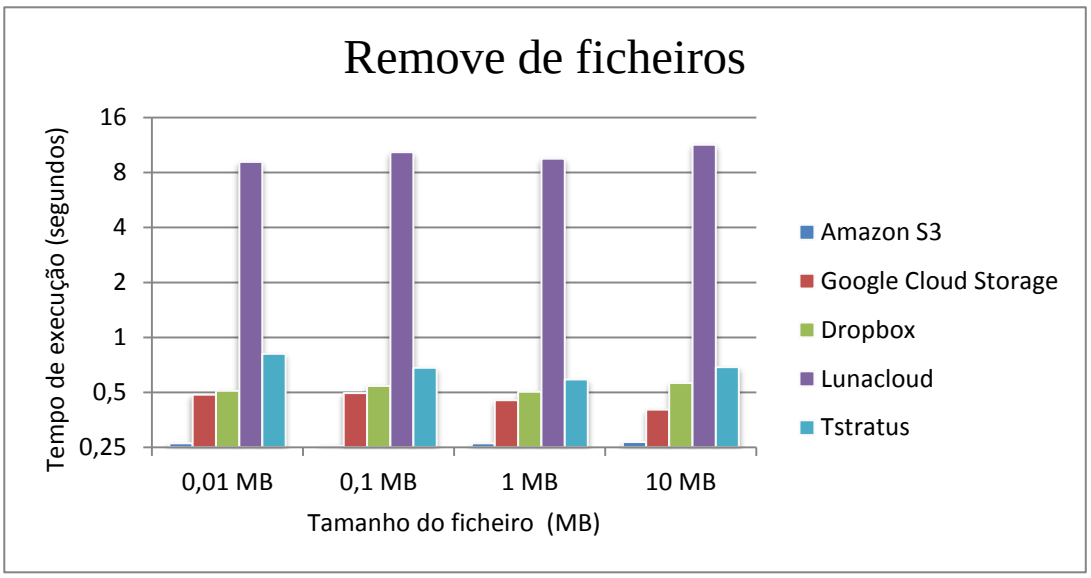



Figura 5.8: Gráfico correspondente aos valores da Tabela 5.13

Aqui o comportamento do *middleware* é semelhante ao `get`, onde os tempos medidos se encontram bastante próximos do terceiro pior tempo registado para as *clouds*. Como seria de esperar, verifica-se o mesmo comportamento que as *clouds*, onde a remoção de um ficheiro é completamente independente do seu tamanho. Derivado do estudo anterior, tanto às *clouds*, como ao peso dos vários componentes do *middleware* numa remoção, seria este o comportamento expectável. Como mencionado acima, iremos verificar um aumento consoante o número de fragmentos em que o ficheiro se divide, visto que é necessário um número de pedidos de `remove` equivalente. Essa será a análise a efetuar nos testes da secção seguinte, onde é tido em conta o número de fragmentos em que se divide o ficheiro.

## 5.7 Diferentes tamanhos de fragmentos ao nível do *middleware*

O resultado da variação do tamanho dos ficheiros escritos para o *middleware* na secção anterior é uma das métricas mais importantes a ser avaliada, pois é o melhor indicador de performance que se pode retirar, face à inserção direta dos ficheiros nas várias *clouds*

usadas. Verificou-se, até este ponto, que a diferença dos tempos medidos não é significativa, de forma a que possa comprometer a performance do sistema e utilização do mesmo como solução segura de armazenamento. No entanto, nesta secção, procurou-se analisar a forma como a alteração do tamanho dos fragmentos em que os ficheiros são repartidos afetada a performance das três operações principais do sistema, `get`, `put` e `remove`. O objetivo é tentar perceber até que ponto compensa dividir um ficheiro em fragmentos e não envia-lo na integra, como um único fragmento, de maior dimensão.

### 5.7.1 Inserção (`put`)

Inicialmente foi testada a variação da escrita de ficheiros de 1 MB para o *middleware*, para fragmentos de 100 KB, 300 KB, 600 KB e 1024 KB (este último é equivalente a não ocorrer fragmentação). Os resultados encontram-se na Tabela 5.14.

| Fragmento | Digest | Metadata | Put | Compressão | Cifra | Digest T | Cache | Riak |
|---|---|---|---|---|---|---|---|---|
| **100 KB** | 0,009 | 0,008 | 26,825 | 0,047 | 0,02 | 0,007 | 0,007 | 0,092 |
| **300 KB** | 0,009 | 0,006 | 17,734 | 0,049 | 0,018 | 0,007 | 0,006 | 0,067 |
| **600 KB** | 0,009 | 0,006 | 16,945 | 0,049 | 0,02 | 0,007 | 0,008 | 0,03 |
| **1024 KB** | 0,009 | 0,007 | 16,302 | 0,048 | 0,019 | 0,007 | 0,004 | 0,03 |

Tabela 5.14: Tempos medidos (em segundos) para operações de escrita de ficheiros de 1 MB para o *middleware*, para diferentes tamanhos de fragmentos

Aqui podemos verificar que para a geração da prova de integridade (Digest) e extração dos metadados do ficheiro (Metadata), a variação é praticamente inexistente, pois não tem qualquer efeito sobre o número de fragmentos e são duas operações que executam sobre todo o conjunto de dados do ficheiro. A Compressão, Cifra, Digest T e Cache (sendo as três primeiras o somatório para os vários fragmentos) acabam por não sofrer alterações significativas e mantêm-se bastante idênticas para fragmentos de 100 KB a 1024 KB. Já o índice Riak, para 100 KB e 300 KB apresenta um tempo superior, derivado da necessidade de armazenar uma maior quantidade de informação associada ao objeto, nomeadamente, os vários fragmentos que o compõem. No geral, ao analisar o tempo total gasto na escrita do ficheiro, conclui-se que, para ficheiros de 1 MB ou inferior, a fragmentação não é viável. Esta apenas irá piorar a performance geral do sistema, que se encontra diretamente condicionada pela operação na *cloud*, como visto anteriormente.

### 5.7.2 Obtenção (`get`)

De seguida foram efetuadas leituras sobre os ficheiros escritos no *middleware* no passo anterior. A Tabela 5.15 reúne os resultados obtidos, de onde podemos tirar ilações bastante semelhantes ao comportamento verificado para as escritas, na subsecção anterior.

| Fragmento | Digest | Get | Descompressão | Decifra | Cache | Riak |
|---|---|---|---|---|---|---|
| **100 KB** | 0,007 | 13,576 | 0,005 | 0,022 | 0 | 0 |
| **300 KB** | 0,007 | 7,849 | 0,005 | 0,024 | 0 | 0 |
| **600 KB** | 0,007 | 6,635 | 0,006 | 0,025 | 0 | 0 |
| **1024 KB** | 0,007 | 4,967 | 0,007 | 0,026 | 0 | 0 |

Tabela 5.15: Tempos medidos (em segundos) para operações de leitura de ficheiros de 1 MB a partir do *middleware*, para diferentes tamanhos de fragmentos

Aqui, verifica-se pouca ou nenhuma variação para a geração do Digest, bem como a Descompressão e a Decifra. O Riak apresenta o valor zero pois não chega a ser acedido em nenhuma das situações acima. O ficheiro em questão encontra-se na cache, e é diretamente obtido da mesma, sem necessidade de recorrer ao índice principal do sistema. O valor zero associado ao tempo de obtenção do objeto da cache, com a informação acerca do ficheiro armazenado não é, na verdade, nulo. Acontece que o valor é demasiado baixo (inferior a 1 milissegundo), não sendo portanto significativo. Finalmente, conclui-se que para ficheiros menores ou iguais a 1 MB a fragmentação não é viável, pois constitui um aumento do tempo total de execução da operação.

### 5.7.3 Remoção (`remove`)

Por fim, o mesmo conjunto de dados foi analisado face à remoção dos ficheiros do *middleware*. Os tempos medidos para esta operação (Tabela 5.16), como seria de esperar, demonstram que quanto maior o número de fragmentos, maior o tempo despendido para toda a operação.

| Tamanho | Remove Frag | Remove | Cache | Riak |
|---|---|---|---|---|
| **100 KB** | 0,491 | 5,4 | 0,005 | 0,014 |
| **300 KB** | 0,523 | 2,09 | 0,005 | 0,012 |
| **600 KB** | 0,5 | 1 | 0,004 | 0,015 |
| **1024 KB** | 0,588 | 0,588 | 0,009 | 0,012 |

Tabela 5.16: Tempos medidos (em segundos) para operações de remoção de ficheiros de 1 MB do *middleware*, para diferentes tamanhos de fragmentos

Esta conclusão era expectável, pois na subsecção 5.6.3 verificou-se que o tamanho do ficheiro não afeta a performance da operação de remoção do mesmo. No entanto, a performance degrada-se com o aumento do número de operações de remoção realizadas ao quórum bizantino. Consequentemente, quando maior o número de fragmentos em que um ficheiro se divide, maior será o tempo necessário para a remoção do mesmo.

Por fim, como conclusão desta secção, embora não sejam apresentados resultados para ficheiros superiores a 1 MB, foram realizados ensaios sobre ficheiros de 10 MB,

para fragmentos de 1 MB, 2 MB, 3 MB, 5 MB e 10 MB. Contudo, apenas foram efetuadas leitura e escritas, pois o teste à remoção de ficheiros foi conclusivo o suficiente para qualquer tamanho a considerar. A conclusão obtida para ficheiros de 10 MB foi idêntica, verificando-se que mesmo para estes, a fragmentação não compensa face à inserção direta de um único fragmento.

## 5.8 Pesquisas Seguras

### 5.8.1 Variante local VS variante *cloud*

Uma das funcionalidades mais importantes do sistema desenvolvido é a capacidade de pesquisa segura sobre um conjunto de metadados de um ficheiro armazenado. No entanto, esta pesquisa não apresenta características complexas de *ranking*, como foi mencionado ao longo do documento, devolvendo um resultado com base no número de ocorrências das palavras chave.

Para este estudo, foram efetuadas três tipos de pesquisas, cujos tempos se encontram na Tabela 5.17. A primeira (Cloud Server) corresponde a uma pesquisa em claro, diretamente no servidor onde é executado o Riak, sobre os ficheiros em disco e os metadados em *plaintext*. Aqui, a única latência existente corresponde ao envio do pedido e à receção da resposta. Todo o processamento é feito localmente, do lado do servidor onde o Riak se encontra a ser executado.

A coluna T-Stratus Cloud apresenta os resultados obtidos para uma pesquisa no Riak, onde o mesmo se encontra no servidor na Lunacloud, como mencionado no inicio deste capítulo. Esta envolve a obtenção dos vários ficheiros do índice e processamento de cada um deles do lado do *middleware*.

Por fim, a coluna T-Stratus Local diz respeito a uma pesquisa igual à anterior, mas onde o índice Riak executa na rede local, numa máquina virtual no próprio sistema onde se encontra o *middleware*, descrito no inicio deste capítulo. A cache, para os vários testes, corresponde sempre a 20% do número total de ficheiros no sistema naquele instante.

| Nº de ficheiros | Cloud Server | T-Stratus Cloud | T-Stratus Local |
|---|---|---|---|
| **100** | 0,138 | 1,43 | 0,336 |
| **200** | 0,15 | 2,165 | 0,603 |
| **300** | 0,163 | 3,197 | 0,862 |
| **400** | 0,169 | 4,082 | 1,119 |
| **500** | 0,17 | 5,505 | 1,359 |
| **600** | 0,181 | 6,13 | 1,624 |
| **700** | 0,181 | 6,976 | 1,903 |
| **800** | 0,188 | 8,435 | 2,199 |
| **900** | 0,193 | 9,144 | 2,359 |
| **1000** | 0,202 | 9,938 | 2,719 |

Tabela 5.17: Tempos medidos (em segundos) para operações de pesquisa de ficheiros

No gráfico da Figura 5.9, é possível ter uma ideia melhor da evolução à medida que o número de ficheiros no sistema aumenta. Nos dados da tabela, bem como no gráfico resultante, podemos observar que a pesquisa direta sobre os metadados dos ficheiros numa pasta local em disco é bastante mais rápida e pouco varia em relação ao aumento do número de ficheiros considerado. Esta discrepância é facilmente explicável pois o tempo de acesso ao disco é muito menor que a latência envolvida na obtenção de um ficheiro no índice. Neste último, é necessária uma operação de `get` para cada ficheiro do sistema, ao passo que na pesquisa local no Cloud Server o acesso é feito diretamente à memória. Posto isto, seria expectável um aumento linear do tempo de pesquisa no *middleware* com o aumento do número de ficheiros existentes no sistema, dada a quantidade de pedidos `get` a efetuar ao índice. Ao passo que na pesquisa local no Cloud Server, e para os valores testados, obtemos quase uma função constante, na pesquisa do *middleware* na *cloud* a complexidade já se situa na ordem de grandeza de $O(n)$, notando-se uma evolução de cerca de 1 segundo para cada 100 ficheiros no sistema. Já para a execução do índice na rede local (T-Stratus Local), verifica-se uma curva menos acentuada, pois os tempos de acesso (latências) são significativamente menores. Assumindo como modelo que um utilizador tem, em média, 1000 ficheiros, sobretudo multimédia, de cerca de 10 MB cada, o valor máximo apresentado corresponde a um volume de dados de 100 GB, o que se pode considerar um valor expectável para uma pasta de um utilizador comum nos dias de hoje.

No entanto, a comparação direta do uso do *middleware* face a uma pesquisa local em disco não é totalmente realista. Em ambos, é necessário aceder a todos os ficheiros individualmente, no entanto, o acesso a disco é bem mais rápido e escala melhor. O objetivo principal foi ver a forma como as três pesquisas evoluíam perante o aumento do número de ficheiros. A pesquisa no *middleware* (T-Stratus Cloud e T-Stratus Local) depende essencialmente da qualidade e capacidade dos canais de comunicação, que consequentemente contribuem para a diminuição ou aumento da latência entre o sistema T-Stratus e

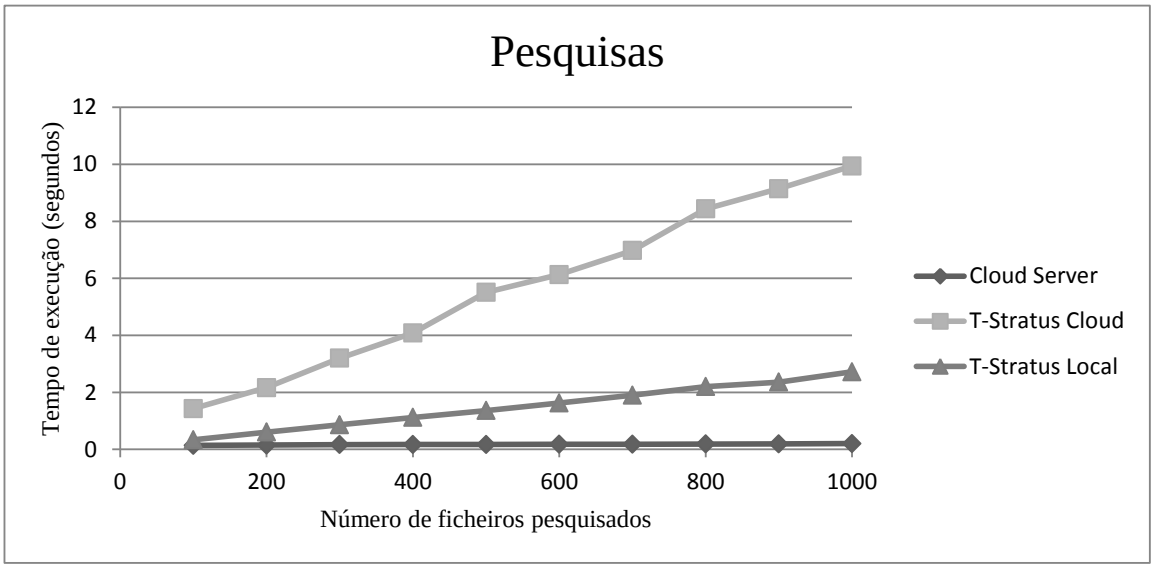



Figura 5.9: Gráfico correspondente aos valores da Tabela 5.17

o servidor onde se encontra o Riak.

### 5.8.2 Pesquisas para múltiplos argumentos

Por fim, os próximos testes tiveram como objetivo avaliar a variação dos tempos de execução da pesquisa (na sua vertente *cloud* e rede local) com o aumento do número de palavras chave a procurar. As Tabelas 5.18 e 5.19 apresentam os resultados para a pesquisa com o aumento do número de palavras chave, respetivamente, para a variante *cloud* e variante local. Não é realista pensar em pesquisas com um número de palavras superior a 20, no entanto, esses valores foram considerados dada a pequena variação que se verificou para valores menores, na tentativa de visualizar um padrão/tendência (gráfico da Figura 5.10).

| Nº palavras | Cache | Cache (%) | Riak | Riak (%) | Tempo total |
|---|---|---|---|---|---|
| **5** | 0,02 | **0,31** | 6,358 | **99,69** | 6,378 |
| **10** | 0,023 | **0,35** | 6,621 | **99,65** | 6,644 |
| **20** | 0,047 | **0,73** | 6,365 | **99,27** | 6,412 |
| **50** | 0,117 | **1,68** | 6,839 | **98,32** | 6,956 |
| **100** | 0,234 | **3,03** | 7,49 | **96,97** | 7,724 |
| **200** | 0,473 | **5,47** | 8,178 | **94,53** | 8,651 |

Tabela 5.18: Tempos medidos (em segundos) para a operação de pesquisa com aumento do número de argumentos, para um índice Riak numa instância *cloud*

| Nº palavras | Cache | Cache (%) | Riak | Riak (%) | Tempo total |
|---|---|---|---|---|---|
| **5** | 0,021 | **1,41** | 1,463 | **98,52** | 1,485 |
| **10** | 0,023 | **1,58** | 1,433 | **98,42** | 1,456 |
| **20** | 0,047 | **2,94** | 1,554 | **97,06** | 1,601 |
| **50** | 0,117 | **6,38** | 1,717 | **93,62** | 1,834 |
| **100** | 0,234 | **9,66** | 2,189 | **90,34** | 2,423 |
| **200** | 0,474 | **12,95** | 3,184 | **87,02** | 3,659 |

Tabela 5.19: Tempos medidos (em segundos) para a operação de pesquisa com aumento do número de argumentos, para um índice Riak numa máquina na rede local do utilizador

Ao analisar os dados das Tabelas 5.18 e 5.10 verifica-se que a pesquisa na cache, visto ser local, não apresenta praticamente alterações entre as duas variantes. Já quanto ao tempo total, era esperada uma diferença considerável devido às diferentes latências entre as duas variantes. Em ambas as variantes a cache passa a ter uma relevância maior à medida que o número de palavras a pesquisar aumenta, verificando-se que a percentagem do tempo ocupada pelo processamento da mesma aumenta com o aumento do número de palavras. No entanto a parametrização do tamanho da cache é um *tradeoff* que é preciso ter em conta, pois quanto maior, mais memória persistente irá ocupar na máquina onde se encontra a instância T-Stratus.

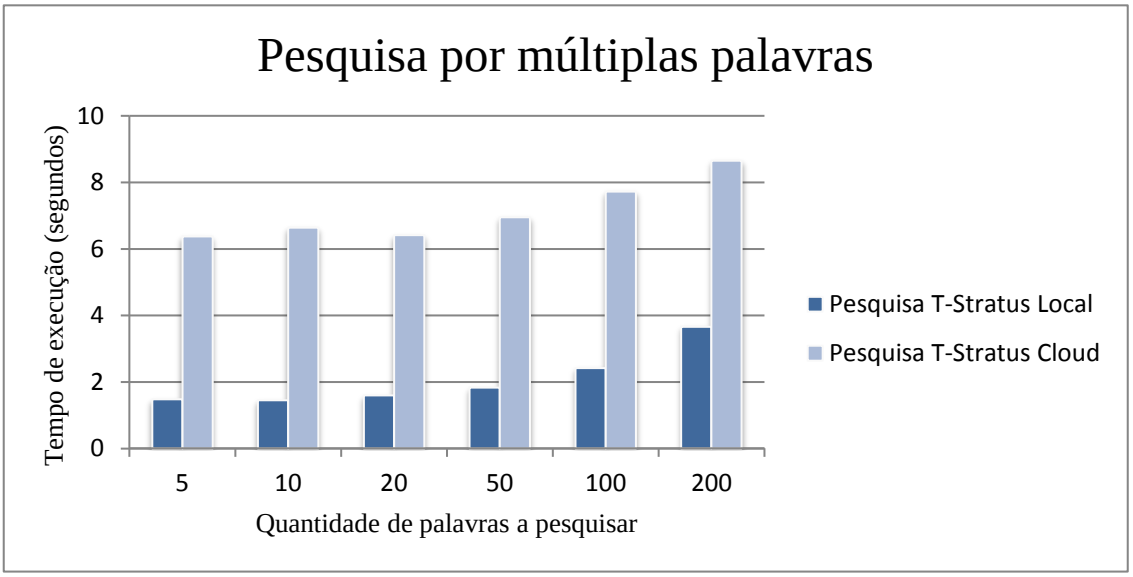



Figura 5.10: Gráfico correspondente aos valores das Tabelas 5.18 e 5.19

## 5.9 Impacto e considerações sobre economia de custos

À semelhança do estudo efetuado pelos autores do DepSky [BCQAS11], em relação à estimativa dos custos envolvidos no uso das quatro *clouds*, estabeleceu-se uma tabela dos mesmos para o sistema T-Stratus, assumindo a situação ótima para as várias operações.

Os montantes observados pelos autores do DepSky remontam a 25 de Setembro de 2010. Seria expectável que neste período de três anos houvessem alterações, no entanto, talvez não tão significativas. No caso da Amazon S3, que é a única *cloud* comum ao T-Stratus e ao DepSky, quando em 2010 apresentavam custos de $\$0,14$ para $10000$ leituras, atualmente apresentam um custo de $\$0,01$, montante significativamente menor. A Tabela 5.20 apresenta os custos para 10000 operações de leitura e escrita, para diferentes volumes de dados. Nesta, os custos associados ao armazenamento propriamente dito não são considerados. São sim, considerados, os custos por cada operação e transferência de dados, tanto de entrada, como saída.

| Operação | Dados | Amazon S3 | Google CS | Dropbox | Lunacloud | T-Stratus |
|---|---|---|---|---|---|---|
| 10k GET | **100 kB** | 0,01 | 0,13 | Grátis | Grátis | 0,01 |
| | **1 MB** | 1,21 | 1,21 | Grátis | Grátis | 1,21 |
| | **10 MB** | 11,77 | 11,77 | Grátis | 7,84 | 19,61 |
| 10k PUT | **100 kB** | 0,1 | 0,01 | Grátis | Grátis | 0,11 |
| | **1 MB** | 0,1 | 0,01 | Grátis | Grátis | 0,11 |
| | **10 MB** | 0,1 | 0,01 | Grátis | Grátis | 0,11 |

Tabela 5.20: Custos por operação e transferência de dados associados a $10000$ pedidos.

Quanto à Amazon S3, são cobrados[3] tanto os pedidos de leitura, como escrita. No entanto, a transferência dos dados de entrada não é cobrada, apenas a de saída (GET) e a partir de 1 GB. A Lunacloud apresenta um modelo de custos[4] em que não cobra nada por qualquer tipo de pedidos e, quanto à transferência de dados, apenas cobra os de saída a partir de 10 GB. A Google CS (Google Cloud Storage), à semelhança da Amazon S3, cobra pelos vários tipos de pedidos e, quanto à transferência de dados, apenas pelos de saída (no entanto, logo a partir dos 0 GB). Os pedidos de DELETE não são cobrados por nenhuma das 4 *clouds*. A Dropbox, embora incluída neste estudo, apresenta um modelo de custos por subscrição, com limites impostos, daí que, quanto ao armazenamento, não é facilmente comparável com as restantes.

Como é possível verificar através dos dados da Tabela 5.20, para 10000 operações e para o caso ótimo, uma operação de leitura no sistema T-Stratus corresponde ao preço das 3 *clouds* de menor custo, dada a obtenção de um quórum de $2f + 1$ para esta operação. Para 100 kB e 1 MB o custo mantém-se igual ao menor das 4 (tendo em conta as que cobram). Já para 10 MB, verifica-se um aumento de cerca de 60% face à *cloud* de menor

[3] `http://aws.amazon.com/s3/pricing/` acedido e verificado a 23-03-2013
[4] `http://www.lunacloud.com/pt/cloud-storage-precos` acedido e verificado a 23-03-2013

custo (Lunacloud) e de 40% em relação às seguintes (Amazon e Google).

Embora o custo total acabe por representar um valor cerca de 60% superior ao da melhor *cloud*, não optando por um quórum de $2f + 1$, teríamos um custo total superior em cerca de 93% (para 100 kB), 50% (para 1 MB) e 37 % (para 10 MB).

No que diz respeito à escrita, estando as quatro *clouds* disponíveis, esta é efetuada em todas. Logo, no melhor caso, corresponde ao somatório dos custos associados à escrita nas várias *clouds*. Atendendo à *cloud* de menor custo, representa um aumento de cerca de 90% em qualquer um dos casos (100 kB, 1 MB e 10 MB), sendo este o preço a pagar pela disponibilidade e replicação dos dados armazenados no sistema T-Stratus.

| Nº PUTs | Dados | Amazon S3 | Google CS | Dropbox | Lunacloud | T-Stratus |
|---|---|---|---|---|---|---|
| 10k | **100 kB** | 0,095 | 0,085 | n/a | 0,07 | 0,25 |
| | **1 MB** | 0,95 | 0,85 | n/a | 0,7 | 2,5 |
| | **10 MB** | 9,31 | 8,33 | n/a | 6,86 | 24,5 |

Tabela 5.21: Custos por operação e transferência de dados associados a $10000$ pedidos.

Por fim, na Tabela 5.21 constam os valores associados ao armazenamento do volume de dados equivalente a 10k PUTs de 100 kB, 1 MB e 10 MB. Obviamente, são também eles o somatório dos respetivos custos associados a cada *cloud*. Representam um aumento de cerca de 72% (para 100 kB, 1 MB e 10 MB), face à *cloud* de menor custo (Lunacloud). Este aumento de 72% é, uma vez mais, o preço a pagar pela replicação e disponibilidade dos dados, tendo em conta um modelo de resiliência de $3f + 1$ *clouds*, para $f$ falhadas.

## 5.10 Discussão

Neste capítulo foi feita uma avaliação do sistema desenvolvido, face ao uso das *clouds* de armazenamento públicas tal como são disponibilizadas atualmente. Verificou-se que a diferença de performance é mínima, quando comparada com as características de segurança acrescida que o *middleware* oferece, tornando a sua utilização prática e eficiente. Acima de tudo, todas as propriedades de segurança mencionadas anteriormente não impõem um processamento suficientemente elevado que torne a solução desenvolvida inviável para uso pessoal ou organizacional.

Inicialmente foram analisadas as escolhas ao nível dos algoritmos que permitem garantir a integridade dos dados, bem como a sua privacidade. O SHA-2, na variante de $512$ bits, demonstrou uma performance superior face à variante $256$ bits e face ao seu sucessor (SHA-3 variante $256$ bits), concluindo que a adoção do mesmo para o sistema permite garantir a integridade dos ficheiros armazenados com ótima performance. A proximidade dos resultados com o SHA-1, embora este último com tempos ligeiramente inferiores, não é vista como fator condicionante pois este não é, atualmente, considerado seguro, não apresentando portanto as características desejadas para o sistema. Não obstante, verifica-se que seria viável a utilização do SHA3, pelo menos na sua variante de

256 bits.

De seguida, o AES revelou ser a melhor opção no que diz respeito ao armazenamento seguro dos dados. Apesar da versão de 128 bits poder ser considerada suficiente nos dias de hoje, o uso de uma chave de 256 bits permite dotar o sistema de uma segurança que se assemelha ao nível de aplicações bancárias ou de gestão de outro tipo de dados sensíveis. Esta escolha permite que os utilizadores usem a T-Stratus de forma transparente para o armazenamento de qualquer tipo de dados, sejam eles mais ou menos importantes do ponto de vista da confidencialidade. Por outro lado, a diferença dos tempos medidos para as duas versões é mínima, quando comparada com a segurança acrescida.

Um fator bastante importante a ter em conta, para além do tempo total de cada operação (inserção, obtenção e remoção), foi o tempo despendido por cada componente em cada uma. Esta análise permitiu verificar qual o *bottleneck* do sistema, de forma a poder contribuir para uma melhor interpretação dos resultados. Dadas as observações anteriores, era expectável que o SHA-512 e o AES apresentassem um papel pouco significativo no tempo total de execução. Este facto comprovou-se após a análise dos vários pesos, verificando-se que não só o protocolo de tolerância a falhas bizantinas é o responsável por mais de 96% do tempo despendido (ou seja, as escritas e leituras para a *cloud*), como à medida que o tamanho do ficheiro aumenta, essa percentagem torna-se ainda mais significativa, face às restantes. No limite, para ficheiros de dimensões superiores a 10 MB, ou até menos, o tempo levado a cabo pelas operações de compressão, cifra e geração do *digest* são desprezáveis, pois o aumento do tempo das escritas e leituras, em relação ao tamanho dos ficheiros, é superior à mesma relação aplicada às três operações mencionadas anteriormente (compressão, cifra e geração do *digest*).

Após esta observação, os testes, permitiram verificar que, havendo largura de banda suficiente, o resultado obtido para a obtenção (operação predominante) no *middleware* é melhor que o obtido para a pior *cloud*, dada a composição de 3 *clouds* no quórum bizantino. O mesmo se verificou para a remoção de dados, que se mantém constante com o aumento do tamanho dos dados a remover. Já a escrita apresentou resultados muito próximos da pior *cloud* registada. Embora não tenha ficado entre a terceira pior e a pior, são na mesma valores aceitáveis, pois uma escrita, tendo disponíveis as quatro *clouds*, escreve para todas, embora a operação retorne sucesso após a escrita em pelo menos 3.

O aumento do número de fragmentos revelou-se pior que o envio dos dados como um todo. Não obstante, estes valores foram obtidos para a fragmentação de dados de 1 MB e 10 MB. Podemos concluir que até aos 10 MB esta constatação se mantém. A impossibilidade de realização de testes para valores superiores a 10 MB não permitiu que fossem tiradas novas ilações acerca de volumes maiores. Dada a característica constante da operação de remoção, esta aumenta consideravelmente apenas com o aumento do número de fragmentos, independentemente do seu tamanho. É possível concluir que, para qualquer uma das três operações principais e para os volumes de dados considerados, a fragmentação não compensa.

O mecanismo de pesquisas seguras, com características homomórficas, sobre os metadados dos ficheiros existentes no sistema aumenta linearmente com o número de ficheiros no sistema. Embora a complexidade seja na ordem de 1 segundo para cada 100 ficheiros existentes, tornasse impraticável para grandes volumes de dados. Já para a variante local, esse aumento é menos significativo. Numa vertente *cloud*, teria de ser considerada a existência de ligações dedicadas entre os componentes, de forma a melhorar este aspeto. Contudo, e através dos resultados obtidos, é dada como sugestão de trabalho futuro um melhoramento do mecanismo de pesquisa, de modo a otimizar a sua performance com o aumento do número de ficheiros no sistema, mantendo, no entanto, as propriedades de segurança da solução atual.

O aumento do conjunto de palavras a pesquisar pouco se reflete no processamento total desta operação. A grande diferença verifica-se, uma vez mais, na variante local face à variante *cloud*. Contudo, considera-se que, para efeitos de pesquisas realistas, com menos de 20 palavras chave, na implementação atual, este aumento do tempo de execução não é significativo. O grande *bottleneck* do sistema é a latência das comunicações entre o cliente e o servidor Riak, pois é feito um pedido GET para cada ficheiro existente no mesmo. A solução envolvia a utilização de canais dedicados, de grande capacidade.

Por fim, os modelos de custo das *clouds* adotadas permitem concluir, como seria de esperar, que há sempre um preço a pagar pelo uso do sistema confiável e seguro com as quatro *clouds* atuais, face ao uso não confiável e inseguro de uma única solução. O armazenamento total corresponde a um aumento de cerca de 72% face à *cloud* de menor custo e as leituras a cerca de 60% (valor possível devido ao quórum de apenas 3 *clouds*). Já para as escritas, bem como o armazenamento, é feito o somatório dos custos para as quatro *clouds*, pois só assim garantimos a disponibilidade dos dados face à indisponibilidade de uma *cloud*. No geral, envolve sempre custos superiores, tipicamente o somatório dos vários custos, sendo as leituras as que maior beneficio podem tirar. Não obstante, são valores ainda aceitáveis e é o preço que se tem a pagar face às propriedades fornecidas pelo sistema T-Stratus, descritas como contribuições e objetivos esperados para esta dissertação.

# 6

# Conclusão e Trabalho Futuro

O trabalho realizado nesta dissertação teve como objetivo modelar e desenvolver um sistema *middleware* responsável pela gestão confiável e segura de dados sensíveis armazenados na *cloud*, numa solução de *cloud* de *clouds*. Estes dados correspondem a ficheiros genéricos, colocados pelo utilizador, que detém todo o controle da base de confiança do sistema, que intermedeia uma aplicação de um utilizador e as várias *clouds* heterogéneas, não necessariamente seguras ou confiáveis, utilizadas como *outsourcing* para o armazenamento dos dados.

Com base na análise de diversos sistemas que tinham como objetivo o endereçamento desta problemática, foi proposto e implementado um sistema *middleware* genérico, com uma API de suporte idêntica aos serviços de armazenamento de dados na *cloud* existentes hoje em dia. Esta API externa permite que aplicações desenvolvidas em Java possam facilmente integrar a T-Stratus, como repositório seguro de armazenamento de dados. Não obstante, para além do armazenamento seguro, oferece capacidade para pesquisas seguras sobre um conjunto de metadados associados aos ficheiros armazenados. A motivação para o desenvolvimento de uma solução que pudesse satisfazer estas necessidades surgiu com o aparecimento da *cloud* e, consequentemente, de novas problemáticas associadas à segurança dos dados armazenados na sua infraestrutura, nomeadamente, o controlo da base de confiança.

A solução desenvolvida deveria ter modularidade suficiente de forma a não constituir um ponto único de falha, bem como a possibilidade de executar a mesma num ambiente do tipo *cloud* privada, confiável, segura e totalmente controlada pelo utilizador.

Assim, foi desenvolvido o sistema T-Stratus, uma *cloud* para controlo de intermediação e acesso transparente de múltiplas *clouds* de armazenamento de dados heterogéneas, onde os mesmo se encontram fragmentados, cifrados e distribuídos segundo um modelo

de replicação bizantina.

A solução desenvolvida e testada é organizada segundo um sistema *middleware* que intermedeia o acesso a múltiplas *clouds* de armazenamento (numa solução do tipo *cloud* de *clouds*), usando múltiplas *clouds* de diferentes provedores como solução de heterogeneidade e diversidade tecnológica, adotando apenas *clouds* de armazenamento como repositórios de dados. O sistema *middleware* proposto, implementado e testado pode ser concretizado e operacionalizado com flexibilidade, em diferentes tipos de solução:

- Como sistema *middleware* executando localmente num computador de um utilizador, para suportar um serviço local de intermediação com as múltiplas *clouds* de armazenamento utilizadas;
- Como serviço "proxy", por exemplo, na rede local de uma instituição, permitindo ser usado como servidor de intermediação entre os utilizadores, através de aplicações executadas nesses mesmos computadores e que são clientes daquele serviço;
- Como uma *cloud* de servidores confiáveis (ou como solução implementada numa *cloud* confiável), como uma solução mais escalável e para condições de acesso ubíquo.

Na sua arquitetura de software, o sistema disponibiliza um sistema confiável de acesso às múltiplas *clouds* de armazenamento, disponibilizando uma API para escritas, leituras e pesquisa de dados, com modelo de concorrência de semântica "one writer – multiple readers" garantindo propriedades de confiabilidade, segurança e privacidade de dados, com controlo dos utilizadores, desde que adote *clouds* que concretizem um modelo de concorrência de semântica regular.

O sistema apresenta imunidade a falhas ou ataques bizantinos desencadeados independentemente ao nível das infra-estruturas de múltiplos provedores de *clouds* de armazenamento Internet. Neste sentido, o sistema garante um modelo de falhas e tolerância a intrusões, nas seguintes condições:

- Suporte de escritas dos dados perante falhas acidentais por omissão (ou paragem) ou ataques à infra-estrutura computacional de um conjunto de *clouds* de armazenamento que tenha como consequência a paragem do seu serviço durante essas operações de escrita. Esta tolerância é garantida desde que um conjunto de $f + 1$ *clouds* esteja disponível para escrita replicada dos dados, no universo de $n$ *clouds* utilizadas, sendo $f$ o número de falhas por paragem que se verifiquem durante as operações de escrita;
- Suporte de leituras dos dados perante falhas bizantinas independentes que possam ocorrer em $f$ *clouds*, desde que $2f + 1$ *clouds* contendo os dados replicados estejam disponíveis.

## 6.1 Objetivos Revistos

Tendo em conta os objetivos inicialmente propostos, podemos considerar que estes foram atingidos. Na implementação e teste do sistema, enquanto arquitetura do tipo *middleware*, utilizável como servidor *proxy*, verificou-se que o custo de processamento acrescido associado ao uso da solução é aceitável, tendo em conta que é o preço a pagar pelas propriedades de confiabilidade e segurança garantidas, com base no controlo da base de confiança por parte dos utilizadores, independentemente de garantias complementares que sejam ou não suportadas ou aceites como acordos de qualidade de serviço (ou SLAs – *Service Level Agreements*) por parte dos provedores desses serviços.

No que diz respeito à avaliação de desempenho do sistema, as contribuições desenvolvidas estão em conformidade com os requisitos especificados e fundamentos da solução proposta revelando esta ser uma solução interessante, verificando-se que as várias operações são processadas em tempos muito idênticos, ou até menos, do que os tempos e latência observados para as piores *clouds*, quando usadas diretamente como repositórios para escrita ou leitura de dados.

A solução proposta garante confidencialidade e condições de privacidade dos dados armazenados, garantindo operações eficientes de pesquisa sobre dados mantidos cifrados, evitando que os mesmos fiquem expostos na infra-estrutura dos provedores, durante essas operações. A solução combina a adoção de criptografia simétrica forte com chaves de tamanho seguro, segundo um esquema de características homomórficas para suportar pesquisas lineares de meta-dados de forma eficiente.

A confidencialidade, autenticidade e integridade dos dados são salvaguardadas por processos criptográficos credíveis, sendo esse suporte ortogonal e transparente em relação à possível utilização de diferentes processos criptográficos incorporados ou a incorporar na solução.

O suporte para múltiplas *clouds* heterogéneas é garantido pela utilização de diferentes conectores que permitem acesso a diferentes soluções heterogéneas de diferentes provedores Internet, tendo sido avaliado o uso de quatro soluções diferentes. A adição de novos conectores é simples, pelo que qualquer *cloud* pode ser considerada desde que se adicione o respetivo conector.

A indexação dos dados (ou fragmentos de dados distribuídos e replicados nas múltiplas *clouds* de armazenamento) é garantida com um índice descentralizado, replicado, de processamento eficiente, onde são armazenados persistentemente todos os objetos correspondentes à indexação e acesso de ficheiros. O desempenho de acesso para leitura ou escrita é constante para qualquer número de ficheiros e depende, essencialmente, da capacidade da ligação entre o sistema e o local onde se encontra a ser executado o processamento do índice. O acesso rápido foi conseguido através de um mecanismo de cache, parametrizável, que existe localmente no servidor do *middleware* (servidor T-Stratus).

As operações de pesquisa segura revelam tempos de resposta idênticos quando realizadas de forma insegura. Na verdade, as pesquisas apenas podem ser mais ou menos

condicionadas pela latência dos canais de comunicação entre o sistema e o servidor onde se encontra alojado o serviço de replicação de índices (com base na solução Riak).

Por fim, a fiabilidade, tolerância a falhas bizantinas e disponibilidade permanente foram garantidas e testadas para um quórum de três *clouds*, de um conjunto total de quatro, revelando o bom funcionamento do sistema. A disponibilidade permanente foi demonstrada assumindo a existência de fragmentos replicados em três das quatro *clouds*, desde que garantidas $3f + 1$ réplicas para $f$ *clouds* indisponíveis.

## 6.2 Trabalho Futuro

Tendo como base o trabalho e reflexões de análise crítica que resultaram da dissertação, emergem diversas direções de investigação complementar que seriam interessantes de endereçar, de forma a estender e melhorar a solução proposta. Neste sentido destacamos as seguintes:

- Suporte para um maior número de *clouds* de armazenamento de dados, com o desenvolvimento de conectores específicos para as mesmas, com realização de testes mais abrangentes para um maior número de *clouds*. Esta abordagem que visaria dotar o sistema de uma maior resiliência face à falha de duas ou mais *clouds*, permitiria analisar a recuperação dos dados armazenados numa situação de maior escala;
- A análise do comportamento da solução observada a partir da execução do sistema T-Stratus em diferentes locais da Internet, forneceria indicações mais completas sobre a eficiência do sistema, independentemente da sua localização. Para este efeito, uma implementação da solução T-Stratus numa infra-estrutura como por exemplo a base experimental PlanetLab, poderá ser considerada para este estudo;
- Uma evolução do sistema para suporte de controlo de concorrência para múltiplos utilizadores (*multiple-writers*, *multiple-readers*), seria uma direção de investigação interessante, tendo em conta a implementação de uma solução de controlo de concorrência face à heterogeneidade de soluções de controlo de concorrência de diferentes *clouds* de armazenamento Internet;
  - A implementação de um mecanismo de controlo de acessos ao nível da utilização das APIs do sistema T-Stratus é também uma direção de trabalho interessante, podendo ser um problema aliciante se a solução for perspetivada para permitir partilha de objetos escritos, lidos e pesquisados por diferentes utilizadores e diferentes aplicações. Nesta direção, o sistema poderia passar a ser perspetivado como um serviço multi-utilizador.
  - O ensaio do sistema utilizando processos criptográficos alternativos é outra direção de trabalho. Nesta direção salientamos:
    * A utilização de provas de autenticação e integridade dos dados com soluções comparativas entre códigos de autenticação de fragmentos ou assinaturas digitais de chave pública desses mesmos fragmentos, avaliando-se

a repercussão do impacto de diferentes soluções (CMAC, HMAC) e diferentes esquemas de assinaturas digitais com métodos criptográficos assimétricos;
    - Utilização de outras formas de confidencialidade com diferentes métodos criptográficos simétricos;
    - Utilização de outras formas de integridade com diferentes métodos de síntese;
    - Implementação e ensaio de esquemas criptográficos homomórficos parciais, para suporte a diferentes tipos de pesquisa que não apenas pesquisas lineares por meta-dados.

- Desenvolvimento de uma aplicação com interface gráfica, através da API fornecida, semelhante à de sistemas atuais como a Dropbox, de modo a propor-se uma aplicação normalizada de referência para uso transparente do sistema T-Stratus;
- Fazer o particionamento dos ficheiros em fragmentos de diferentes dimensões, sendo as mesmas calculadas em função dos tempos observados para as diversas operações e em diferentes tamanhos de ficheiros. Observou-se, por exemplo, que *clouds* que apresentavam piores medições em tamanhos inferiores, acabavam por ser melhores (relativamente às restantes) para fragmentos de tamanhos superiores. Este melhoramento seria mais orientado à performance e, não tanto, ao custo, embora a análise de métricas de custo económico sejam elas próprias uma outra direção de investigação bastante interessante;
- Para efeitos de uma análise de impacto de custos económicos de exploração da solução (enquanto solução de *cloud* de *clouds*), seria uma boa abordagem a de dividir os ficheiros em fragmentos de acordo com os modelos de custo associados às várias *clouds* de armazenamento de dados. Estes modelos podem assumir diferentes custos para diferentes tamanhos dos dados enviados, bem como diferentes padrões de escritas e leituras. A otimização do processo de fragmentação e replicação de fragmentos de modo a minimizar os custos económicos é uma linha de investigação muito interessante.
    - Em complemento ao anterior, seria interessante a exploração da combinação de mecanismos com técnicas de *erasure coding* no processo de fragmentação dos dados, já que por essa via se podem obter também reduções de impacto de custo económico da solução;
    - A disponibilizarão futura de outras APIs de externalização dos serviços *middleware* da solução proposta é uma outra direção de trabalho futuro. Esta abordagem pode permitir processos de facilitação de transporte de aplicações com reutilização transparente da solução. Nesta direção, a implementação de APIs externas concretizando um modelo RAID (ex., RAID5), permitiria que a solução fosse vista como uma solução de "cluster virtual de *array* de discos", sendo os discos concretizados pelas *clouds* de armazenamento.

# Bibliografia


[Aba09] D. J. Abadi. *Data Management in the Cloud: Limitations and Opportunities*. IEEE Data Engineering Bulletin, 32(1). 2009.

[AEMGGRW05] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter e J. J. Wylie. "Fault-scalable Byzantine fault-tolerant services". Em: *Proceedings of the twentieth ACM symposium on Operating systems principles*. ACM SOSP '05. Brighton, United Kingdom, 2005, pp. 59–74. ISBN: 1-59593-079-5. DOI: http://doi.acm.org/10.1145/1095810.1095817. URL: http://doi.acm.org/10.1145/1095810.1095817.

[Ama] *Amazon S3*. http://aws.amazon.com/s3/ (verificado e acedido em 11/03/2013).

[ABCKPR09] J. Archer, A. Boehme, D. Cullinane, P. Kurtz, N. Puhlmann e J. Reavis. *Security Guidance for Critical Areas of Focus in Cloud Computing V2.1*. 2009.

[AFGJKKLPRSZ10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica e M. Zaharia. "A view of cloud computing". Em: *Commun. ACM* 53 (4 abr. de 2010), pp. 50–58. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672.

[BCHHHRV09] E. Baize, R. Cloutier, B. Hartman, D. S. Herrod, C. Hollis, U. Rivner e B. Verghese. *Identity & Data Protection in the Cloud: Best Practices for Establishing Environments of Trust*. 2009.

[BCQAS11] A. Bessani, M. Correia, B. Quaresma, F. André e P. Sousa. "DepSky: dependable and secure storage in a cloud-of-clouds". Em: *Proceedings of the sixth conference on Computer systems*. ACM EuroSys '11. Salzburg, Austria, 2011, pp. 31–46. ISBN: 978-1-4503-0634-8. DOI:

http://doi.acm.org/10.1145/1966445.1966449. URL: http://doi.acm.org/10.1145/1966445.1966449.

[Bft] *BFT-SMaRt*. http://code.google.com/p/bft-smart/ (verificado e acedido em 12/03/2013).

[BJO09] K. D. Bowers, A. Juels e A. Oprea. "HAIL: a high-availability and integrity layer for cloud storage". Em: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM CCS '09. Chicago, Illinois, USA, 2009, pp. 187–198. ISBN: 978-1-60558-894-0. DOI: http://doi.acm.org/10.1145/1653662.1653686. URL: http://doi.acm.org/10.1145/1653662.1653686.

[Bur03] W. E. Burr. "Selecting the Advanced Encryption Standard". Em: *IEEE Security and Privacy* 1.2 (mar. de 2003), pp. 43–52. ISSN: 1540-7993. DOI: 10.1109/MSECP.2003.1193210. URL: http://dx.doi.org/10.1109/MSECP.2003.1193210.

[Cal+11] B. Calder et al. "Windows Azure Storage: a highly available cloud storage service with strong consistency". Em: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM SOSP '11. Cascais, Portugal, 2011, pp. 143–157. ISBN: 978-1-4503-0977-6. DOI: http://doi.acm.org/10.1145/2043556.2043571. URL: http://doi.acm.org/10.1145/2043556.2043571.

[CL99] M. Castro e B. Liskov. "Practical Byzantine fault tolerance". Em: *Proceedings of the third symposium on Operating systems design and implementation*. USENIX Association OSDI '99. New Orleans, Louisiana, United States, 1999, pp. 173–186. ISBN: 1-880446-39-1. URL: http://dl.acm.org/citation.cfm?id=296806.296824.

[CDGHWBCFG08] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes e R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". Em: *ACM Trans. Comput. Syst.* 26 (2 2008), 4:1–4:26. ISSN: 0734-2071. DOI: http://doi.acm.org/10.1145/1365815.1365816. URL: http://doi.acm.org/10.1145/1365815.1365816.

[CLGKP93] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz e D. A. Patterson. *RAID: High-Performance, Reliable Secondary Storage*. Rel. téc. Berkeley, CA, USA, 1993.

[CPK10] Y. Chen, V. Paxson e R. H. Katz. *What's New About Cloud Computing Security?* Rel. téc. UCB/EECS-2010-5. EECS Department, University of California, Berkeley, 2010. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-5.html.

[CS11] Y. Chen e R. Sion. "To cloud or not to cloud?: musings on costs and viability". Em: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM SOCC '11. Cascais, Portugal, 2011, 29:1–29:7. ISBN: 978-1-4503-0976-9. DOI: http://doi.acm.org/10.1145/2038916.2038945. URL: http://doi.acm.org/10.1145/2038916.2038945.

[Cho10] K.-K. R. Choo. "Cloud computing: Challenges and future directions". Em: *Trends & Issues in Crime and Criminal Justice* 400 (2010). Australian Institute of Criminology, Australian Government, 2010, p. 6. URL: http://www.aic.gov.au/documents/C/4/D/\%7BC4D887F9-7D3B-4CFE-9D88-567C01AB8CA0\%7Dtandi400.pdf.

[CGJSSMM09] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka e J. Molina. "Controlling data in the cloud: outsourcing computation without outsourcing control". Em: *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM CCSW '09. Chicago, Illinois, USA, 2009, pp. 85–90. ISBN: 978-1-60558-784-4. DOI: http://doi.acm.org/10.1145/1655008.1655020. URL: http://doi.acm.org/10.1145/1655008.1655020.

[CKLWADR09] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin e T. Riche. "Upright cluster services". Em: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM SOSP '09. Big Sky, Montana, USA, 2009, pp. 277–290. ISBN: 978-1-60558-752-3. DOI: http://doi.acm.org/10.1145/1629575.1629602. URL: http://doi.acm.org/10.1145/1629575.1629602.

[CMLRS06] J. Cowling, D. Myers, B. Liskov, R. Rodrigues e L. Shrira. "HQ replication: a hybrid quorum protocol for byzantine fault tolerance". Em: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association OSDI '06. Seattle, Washington, 2006, pp. 177–190. ISBN: 1-931971-47-1. URL: http://dl.acm.org/citation.cfm?id=1298455.1298473.

[DHJKLPSVV07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall e W. Vogels. "Dynamo: amazon's highly available key-value store". Em: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. ACM SOSP '07. Stevenson, Washington, USA, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: http://doi.acm.org/10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[FD12] B. Ferreira e H. J. Domingos. "Gestão e Pesquisa de Dados Privados em Nuvens de Armazenamento". Em: *Inforum Simpósio de Informártica*. 2012.

[GJW11] S. Gueron, S. Johnson e J. Walker. "SHA-512/256". Em: *Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations*. ITNG '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 354–358. ISBN: 978-0-7695-4367-3. DOI: 10.1109/ITNG.2011.69. URL: http://dx.doi.org/10.1109/ITNG.2011.69.

[HPPT08] A. Heitzmann, B. Palazzi, C. Papamanthou e R. Tamassia. "Efficient integrity checking of untrusted network storage". Em: *Proceedings of the 4th ACM international workshop on Storage security and survivability*. ACM StorageSS '08. Alexandria, Virginia, USA, 2008, pp. 43–54. ISBN: 978-1-60558-299-3. DOI: http://doi.acm.org/10.1145/1456469.1456479. URL: http://doi.acm.org/10.1145/1456469.1456479.

[JGMSV08] R. C. Jammalamadaka, R. Gamboni, S. Mehrotra, K. E. Seamons e N. Venkatasubramanian. "iDataGuard: middleware providing a secure network drive interface to untrusted internet data storage". Em: *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. ACM EDBT '08. Nantes, France, 2008, pp. 710–714. ISBN: 978-1-59593-926-5. DOI: http://doi.acm.org/10.1145/1353343.1353432. URL: http://doi.acm.org/10.1145/1353343.1353432.

[JG11] W. Jansen e T. Grance. *Guidelines on Security and Privacy in Public Cloud Computing*. United States of America: National Institute of Standards and Technology Special Publication. 2011.

[JSBGI11] M. Jensen, J. Schwenk, J.-M. Bohli, N. Gruschka e L. L. Iacono. "Security Prospects through Cloud Computing by Adopting Multiple Clouds". Em: *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*. IEEE Computer Society CLOUD '11. Washington, DC, USA, 2011, pp. 565–572. ISBN: 978-0-7695-4460-1. DOI: http://dx.doi.org/10.1109/CLOUD.2011.85. URL: http://dx.doi.org/10.1109/CLOUD.2011.85.

[KSSZWS11] J. Konczak, N. F. de Sousa Santos, T. Zurkowski, P. T. Wojciechowski e A. Schiper. *JPaxos: State machine replication based on the Paxos protocol*. Rel. téc. 2011.

[KADCW07] R. Kotla, L. Alvisi, M. Dahlin, A. Clement e E. Wong. "Zyzzyva: speculative byzantine fault tolerance". Em: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. ACM SOSP '07. Stevenson, Washington, USA, 2007, pp. 45–58. ISBN: 978-1-59593-591-5. DOI: http://doi.acm.org/10.1145/1294261.1294267. URL: http://doi.acm.org/10.1145/1294261.1294267.

[LM10] A. Lakshman e P. Malik. "Cassandra: a decentralized structured storage system". Em: *ACM SIGOPS Oper. Syst. Rev.* 44 (2 2010), pp. 35–40. ISSN: 0163-5980. DOI: http://doi.acm.org/10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[Lam98] L. Lamport. "The part-time parliament". Em: *ACM Trans. Comput. Syst.* 16 (2 1998), pp. 133–169. ISSN: 0734-2071. DOI: http://doi.acm.org/10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[LM04] L. Lamport e M. Massa. "Cheap Paxos". Em: *Proceedings of the 2004 International Conference on Dependable Systems and Networks*. IEEE Computer Society Washington, DC, USA, 2004, pp. 307–. ISBN: 0-7695-2052-9. URL: http://dl.acm.org/citation.cfm?id=1009382.1009745.

[LSP82] L. Lamport, R. Shostak e M. Pease. "The Byzantine Generals Problem". Em: *ACM Trans. Program. Lang. Syst.* 4 (3 1982), pp. 382–401. ISSN: 0164-0925. DOI: http://doi.acm.org/10.1145/357172.357176. URL: http://doi.acm.org/10.1145/357172.357176.

[MHS03] M. C. Mont, K. Harrison e M. Sadler. "The HP time vault service: exploiting IBE for timed release of confidential information". Em: *Proceedings of the 12th international conference on World Wide Web*. ACM WWW '03. Budapest, Hungary, 2003, pp. 160–169. ISBN: 1-58113-680-3. DOI: http://doi.acm.org/10.1145/775152.775175. URL: http://doi.acm.org/10.1145/775152.775175.

[NLV11] M. Naehrig, K. Lauter e V. Vaikuntanathan. "Can homomorphic encryption be practical?" Em: *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM CCSW '11. Chicago, Illinois, USA, 2011, pp. 113–124. ISBN: 978-1-4503-1004-8. DOI: http://doi.acm.org/10.1145/2046660.2046682. URL: http://doi.acm.org/10.1145/2046660.2046682.

[NGSN10] S. Narayan, M. Gagné e R. Safavi-Naini. "Privacy preserving EHR system using attribute-based infrastructure". Em: *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*. ACM CCSW '10. Chicago, Illinois, USA, 2010, pp. 47–52. ISBN: 978-1-4503-0089-6. DOI: http://doi.acm.org/10.1145/1866835.1866845. URL: http://doi.acm.org/10.1145/1866835.1866845.

[Pai] *Paillier Cryptosystem - Implementação Java*. http://www.csee.umbc.edu/~kunliu1/research/Paillier.html (verificado e acedido em 28/01/2012).

[PRZB11] R. A. Popa, C. M. S. Redfield, N. Zeldovich e H. Balakrishnan. "CryptDB: protecting confidentiality with encrypted query processing". Em: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM SOSP '11. Cascais, Portugal, 2011, pp. 85–100. ISBN: 978-1-4503-0977-6. DOI: http://doi.acm.org/10.1145/2043556.2043566. URL: http://doi.acm.org/10.1145/2043556.2043566.

[PKZ11] K. P. N. Puttaswamy, C. Kruegel e B. Y. Zhao. "Silverline: toward data confidentiality in storage-intensive cloud applications". Em: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM SOCC '11. Cascais, Portugal, 2011, 10:1–10:13. ISBN: 978-1-4503-0976-9. DOI: http://doi.acm.org/10.1145/2038916.2038926. URL: http://doi.acm.org/10.1145/2038916.2038926.

[Ree] *Reed–Solomon Codes*. http://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html (verificado e acedido em 28/01/2012).

[Ria] *Riak*. http://basho.com/products/riak-overview/ (verificado e acedido em 11/03/2013).

[SSBT10] J. Sheehy, D. Smith, E. Brewer e B. Technologies. *Bitcask: A Log-inspired Hash Table for Fast Key/value Data*. 2010. URL: http://books.google.pt/books?id=HydStwAACAAJ.