Hylson Vescovi Netto

ARMAZENAMENTO DE DADOS EFICIENTE TOLERANTE A FALTAS BIZANTINAS EM MÚLTIPLAS NUVENS COM COORDENAÇÃO DE METADADOS INTEGRADA A UM GERENCIADOR DE *CONTAINERS*

Tese submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, como requisito para a obtenção do Grau de Doutor em Ciência da Computação.
Orientador: Prof. Lau Cheuk Lung, Dr. Eng.

Florianópolis

2017

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Netto, Hylson Vescovi
Armazenamento de dados eficiente tolerante a faltas bizantinas em múltiplas nuvens com coordenação de metadados integrada a um gerenciador de containers / Hylson Vescovi Netto ; orientador, Lau Cheuk Lung, 2017.
176 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós Graduação em Ciência da Computação, Florianópolis, 2017.

Inclui referências.

1. Ciência da Computação. 2. Armazenamento de dados. 3. Computação em nuvem. 4. Tolerância a faltas bizantinas. 5. Containers. I. Lung, Lau Cheuk. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Hylson Vescovi Netto

Armazenamento de Dados Eficiente Tolerante a Faltas Bizantinas em Múltiplas Nuvens com Coordenação de Metadados Integrada a um Gerenciador de *Containers*

Esta tese foi julgada adequada para obtenção do título de doutor e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 19 de maio de 2017.

JOSÉ LUÍS ALMADA GÜNTZEL
Subcoordenador do PPGCC/UFSC
Portaria Nº 1882/2015/GR

Banca Examinadora:

Profª. Luciana de Oliveira Rech, Drª.
Universidade Federal de Santa Catarina
Presidente

Prof. Elias Procópio Duarte Junior, Dr.
Universidade Federal do Paraná

Prof. Luis Eduardo Teixeira Rodrigues, Dr.
Universidade de Lisboa, Portugal

Prof. Joni da Silva Fraga, Dr.
Universidade Federal de Santa Catarina

Prof. Ricardo Alexandre Reinaldo de Moraes, Dr.
Universidade Federal de Santa Catarina

Prof. Frank Augusto Siqueira, Dr.
Universidade Federal de Santa Catarina

Este trabalho é dedicado à memória de Lau Cheuk Lung, meu orientador.

AGRADECIMENTOS

Agradeço a Deus por esta boa passagem na Terra, com muitas alegrias, dificuldades e consequentes aprendizagens.

À minha mãe Maria, pelo constante incentivo à realização desse sonho. Ao meu pai Jorge, pelo constante apoio e ajuda nas minhas andanças pelo Brasil, em busca da realização. À minha esposa Léia, por me acompanhar firmemente nesse desafio. Ao amigo José Wayne (Zé) e sua família, pelas lições de vida, pelos cafés, almoços e jantares.

Ao Koller, pelo incentivo e pela oportunidade de mudança de lotação no IFC. Ao amigo Aldelir, por me apresentar a oportunidade de ser orientado pelo Lau e pela parceria nos artigos. Aos professores Crediné e Pessoa (UFES), pela experiência propulsora do mestrado, que permanece até hoje. Ao professor Elias de Oliveira (UFES), pela preparação que me forneceu para a vida de doutorando e pela sua orientação que me levou a encontrar um ponto de contato na UFSC. Ao Juliano Brignoli, pelo incentivo de iniciar as jornadas pelas estradas da minha residência até a UFSC, na modalidade de aluno especial. Ao Instituto Federal Catarinense (IFC), pela concessão dos horários especiais de estudante, bem como a concessão do afastamento integral para a realização do doutorado.

Aos professores Dalton e Bornia, pela introdução e apaixonamento pelo mundo da Estatística. Ao professor Marcelo, pelas empolgantes aulas de Estatística e as boas conversas sobre resultados em artigos. À professora Vânia, a cujo *template* fornecido para a escrita de artigos ainda recorro eventualmente. À professora Carina, pela revelação da estrutura que compõe um parágrafo. Aos professores Joni, Frank e Mário Dantas, pelas aulas que me localizaram no domínio de Sistemas Distribuídos, bem como pelas bancas intermediárias nas quais esses dois últimos participaram. Ao Uriarte por me advertir de que "doutorado é organização".

Ao Tulio pela introdução ao laboratório, amizade e parceria nos artigos. Ao Sergio, pela animação no laboratório. Ao Paulo Pinho, pela companhia e suas lições de sobriedade. Ao Ricardo e ao Paulo Moraes, pelas hospedagens (incluindo café da manhã e jantar!) e convivências. Ao mestre Tigre, pelos momentos de convivência na capoeiragem. Ao Lúcio, pela introdução no mundo do Tai Chi.

Ao professor Miguel, pelo exemplo e valiosas co-autorias. À Luciana Souza, pela parceria nos artigos e consequentes aprendizagens envolvidas (escrita de artigos, inglês). Aos parceiros de laboratório com os quais a convivência possibilitou muitos momentos: André, Eliza.

Ao amigos de corredores e sala de espera da limpeza, pelas conversas inspiradoras: Camilo, Ivan.

Ao Telmo, pelo convite para a meditação. Ao Jeovani, pelas conversas animadas em memoráveis terças-feiras. Aos professores e colegas do inglês sem fronteiras presencial, por ajudar na construção da proficiência em inglês. Ao mestre Polegar, por compartilhar de forma tão generosa sua capoeira. Aos camaradas da capoeira, pela companhia! Ao Oliveiros, pelas conversas esclarecedoras e suas lições de movimento e contemplação da vida.

Aos parceiros que fizeram revisões preliminares nesta tese (ainda não citado: o Leandro Loss). Ao Jim Lau, pelas revisões na escrita desta tese. À Luciana Rech, pela revisão desta tese e por ter continuado em frente. Ao professor Elias Duarte, pela específica revisão desta tese.

Por último, e de forma mais expressiva, agradeço ao professor Lau. Agradeço a ele pela aceitação desse desafio e pela constante orientação, mesmo nos momentos mais difíceis, tanto para mim quanto para ele. Ao professor Lau agradeço também por disponibilizar o espaço de trabalho LaPeSD, palco de muitas ações e emoções.

Existem mais pessoas que desistem do que pessoas que fracassam.

Henry Ford

RESUMO

O armazenamento de dados em nuvens tem se tornado uma opção para permitir a geo-replicação. No contexto de aplicações críticas, é preciso garantir tolerância a faltas para que essas aplicações possam depender dos dados armazenados nas nuvens. Esta tese apresenta soluções para tornar mais simples e eficiente a operação de sistemas críticos que armazenam dados em múltiplos provedores de nuvem (*Intercloud*). Três contribuições integram esta tese. A primeira contribuição consiste em uma técnica denominada Antecipação de Pedidos (AdP) cujo objetivo é melhorar a eficiência de protocolos baseados em quóruns. Essa técnica é otimista e considera a latência dos provedores envolvidos na operação para paralelizar parcialmente fases de um protocolo. A aplicação da AdP no contexto desta tese resulta no RafeStore, um sistema confiável e eficiente de armazenamento de dados em múltiplas nuvens. O RafeStore considera uma categoria específica de dados multiversão, denominada Dado de Conteúdo Independente, cuja formação de novos valores não depende de valores anteriores. A avaliação do RafeStore usando provedores de nuvem comerciais demonstra que o mesmo requer menores latência e custo, quando comparado a outros sistemas baseados em quóruns bizantinos. A segunda contribuição desta tese refere-se à organização da execução de requisições, necessária quando múltiplos clientes atualizam simultaneamente um mesmo dado. Para tal fim, é proposto o sistema DORADO, que coordena metadados segundo a estratégia de replicação de máquinas de estado. O DORADO é projetado para funcionar no gerenciador de *containers* Kubernetes, visando o uso dessa emergente tecnologia de virtualização. A incorporação do DORADO ao Kubernetes por meio de integração torna a coordenação de metadados transparente sob a perspectiva do usuário. A avaliação de uma integração parcial demonstra a viabilidade dessa proposta. A contribuição final desta tese é o sistema denominado FITS, cuja função é orquestrar os sistemas RafeStore e DORADO. Dessa maneira, obtém-se um sistema de armazenamento de dados eficiente e tolerante a faltas que opera na *Intercloud*.

Palavras-chave: Armazenamento de Dados. Computação em Nuvem. Tolerância a Faltas Bizantinas. *Containers*.

ABSTRACT

Storing data in clouds has become an option in enabling geo-replication. In the context of critical systems, fault tolerance is required in order for the applications to be able to rely on the data stored in the cloud. This thesis presents solutions to simplify and make the operation of critical systems which store data in multiple cloud providers (Intercloud) more efficient. The contributions in this Thesis are threefold. The first contribution of this thesis consists in a technique to improve the efficiency of quorum-based protocols. We named it Requests Anticipation. This technique is optimistic and considers latency of providers to partially parallelize phases in a protocol. Applying request anticipation in the storage context results in RafeStore, a dependable and efficient system which stores data in multiple clouds. RafeStore considers a specific multiversion data type, named Data with Independent Content. With this type of data, new values are not necessarily related to previous ones. Our evaluation of RafeStore in commercial providers demonstrates that it requires lower latency and costs, when compared to other Byzantine quorum-based systems. The second contribution of this thesis refers to the organization of requests when multiple users simultaneously update the same data (i.e., race condition). To accomplish that, we propose DORADO: a system that manages metadata according to the strategy of state machine replication. DORADO was designed to work inside the container management system called Kubernetes, aiming at taking advantage of its emerging virtualization technology. Incorporating DORADO in Kubernetes via integration makes the metadata coordination transparent to the user. Our preliminary evaluation of the proposed approach demonstrates its viability.The third and final contribution of this thesis is FITS, a system that orchestrates RafeStore and DORADO. FITS enables the operation of an efficient and fault tolerant storage data system in the Intercloud.

Keywords: Storage. Cloud Computing. Byzantine Fault Tolerance. Containers.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

2PC	*Two-Phase Commit*
AdP	Antecipação de Pedidos
API	*Application Programming Interface*
BD	Banco de Dados
BFT	*Byzantine Fault Tolerance*
CA	*Consistency Anchor*
CAP	*Consistency, Availability and Partition Tolerance*
CIRC	*Confidentiality, Integrity, Reliability and Consistency*
CNCF	*Cloud Native Computing Foundation*
DORADO	*or**D**ering **O**ve**R** sh**A**re**D** mem**O**ry*
EC	*Erasure codes*
FITS	*e**F**fect**I**ve s**T**orage for the ma**S**ses*
FLP	Iniciais dos sobrenomes Fisher, Lynch e Patterson
FRS	*Fragmentation, replication and scattering*
GST	*Global Stabilization Time*
k8s	Kubernetes
LAN	*Local Area Network*
LXC	Linux *containers*
MWMR	*Multiple Writer, Multiple Readers*
MC	Memória Compartilhada
MV	Máquina Virtual
MVCC	*Multi-Version Concurrency Control*
PBFT	*Practical Byzantine Fault Tolerance*
RACS	*Redundant Array of Cloud Storage*
RafeStore	***R**eliable, **a**vailable, **f**ast and **e**conomic **Stor**ag**e***
RAID	*Redundant Array of Inexpansive Disks*
RCD	Registrador Compartilhado Distribuído
RMDS	*Reliable MetaData Service*
RME	Replicação de Máquinas de Estado
RTT	*Round-Trip Time*
SCFS	*Shared Cloud-backed File System*
SGBD	Sistema Gerenciador de Banco de Dados

SS	*Storage Service*
SEL	Solicitação de Eleição de Líder
SWMR	*Single Writer, Multiple Readers*
UUID	*Universally Unique IDentifier*
WAN	*Wide Area Network*

SUMÁRIO

1 INTRODUÇÃO

Dance primeiro, pense depois. Essa é a ordem natural.
Samuel Beckett

O armazenamento de dados em provedores de computação em nuvem (*Cloud Computing*) tem se tornado uma opção economicamente viável, especialmente para fins de replicação geográfica de dados (MELL; GRANCE, 2010). Replicar dados em um provedor de nuvem favorece a sobrevivência do dado mediante eventos como problemas em computadores e erros humanos (por exemplo, exclusão acidental de arquivos), fatores predominantes entre as causas de perda de dados (SMITH, 2003). Os preços praticados pelos provedores são eventualmente reduzidos por questões de concorrência (Apêndice A, itens 1 a 5). A alta disponibilidade também é uma característica propulsora na adoção de provedores de nuvens como locais para armazenar dados. A replicação de dados em diversos *data centers* de um provedor torna os dados mais próximos a clientes distribuídos geograficamente.

Apesar das vantagens do cenário descrito, aplicações críticas podem ser afetadas por problemas sutis existentes nos domínios da computação em nuvem. A pequena indisponibilidade de 0,01% mencionada por provedores (Apêndice A, item 62) pode se manifestar de maneira mais expressiva quando o sistema opera em ambientes altamente conectados como a Internet. Outra indisponibilidade menos frequente, porém possível, é causada pela desativação completa e abrupta de um provedor. Relatos de tais ocorrências apontam causas diversas, como questões financeiras, falta de oferta de outros serviços relacionados a armazenamento de dados ou mesmo à mudança de foco da empresa provedora (exemplos no Apêndice A, itens 50 a 52).

Problemas relacionados ao armazenamento podem também ocorrer quando o dado é acessível mas não está correto. A tolerância a faltas bizantinas (*Byzantine Fault Tolerance* - BFT) permite ao cliente assegurar-se de que uma operação sobre o dado (leitura ou escrita) está correta, respeitando-se um limite máximo f de faltas toleradas (CASTRO; LISKOV, 2002). Isso é alcançado obtendo-se respostas iguais de pelo menos $f+1$ provedores. Dessa maneira, o cliente certifica-se de que a atuação do sistema é correta quanto à operação sobre o dado.

Uma solução para mitigar as limitações individuais de provedores de nuvem é o uso coordenado de múltiplos provedores. A *Intercloud* (nuvem-de-nuvens) significa uma nuvem virtual composta de diversos (e

diferentes) provedores de nuvem (BERNSTEIN et al., 2009). Por meio da *Intercloud*, aumenta-se a disponibilidade, evita-se a dependência de um provedor único e obtém-se tolerância a desastres (Seção 2.2.5). A literatura dispõe de trabalhos que atuam na *Intercloud*, fornecendo também BFT. O DepSky (BESSANI et al., 2013) armazena dados em múltiplos provedores de maneira confiável. Porém, este e a maioria dos protocolos BFT são notoriamente custosos, em comparação a sistemas tolerantes a faltas de parada. Trabalhos como o MDStore (CACHIN; DOBRE; VUKOLIĆ, 2014) e o Hybris (DOBRE; VIOTTI; VUKOLIĆ, 2014) propõem reduções para o número de réplicas nas quais o dado é armazenado. Com menos réplicas, os custos de implementar o BFT podem ser reduzidos.

Diversas organizações que operam sistemas críticos não possuem interesse em manter infraestrutura computacional (BESSANI et al., 2013). Para essas organizações, armazenamento em nuvens é uma opção atraente, devido ao modelo de custos sob demanda. Entretanto, aplicações críticas não podem experimentar problemas de indisponibilidade de dados. Por exemplo, sistemas de energia possuem bases de dados históricos que contém informações sobre eventos coletados na rede de energia e em outros subsistemas. Essas bases de dados devem estar sempre disponíveis para consulta. Nesse contexto, considera-se importantes os esforços na redução de custos em sistemas de armazenamento BFT que operem na *Intercloud*.

Esta tese busca tornar mais eficientes protocolos de armazenamento de dados em múltiplos provedores de nuvem. Como primeira contribuição, a técnica de Antecipação de Pedidos (AdP) utiliza a latência dos provedores como critério para paralelizar parcialmente as fases de protocolos baseados em quóruns. Essa técnica foi aplicada no sistema de armazenamento denominado RafeStore, em um cenário *Intercloud*. O RafeStore considera uma categoria específica de dados multiversão, denominada dado de conteúdo independente (Seção 4.1), cuja formação de novos valores não depende de valores anteriores. A avaliação do RafeStore demonstra redução de latência percebida pelo cliente, bem como a redução de custos que envolvem os provedores, quando comparado a outros sistemas de armazenamento.

A segunda contribuição desta tese refere-se à organização da execução de requisições. Quando múltiplos clientes atualizam simultaneamente um mesmo dado, é preciso coordenar o acesso ao dado, para evitar sobreposições. Para essa finalidade, esta tese propõe o sistema DORADO, que coordena os metadados segundo a estratégia replicação de máquinas de estado (RME). O DORADO é projetado para fun-

cionar no gerenciador de *containers* Kubernetes, visando o uso dessa emergente tecnologia de virtualização. A incorporação do DORADO ao Kubernetes por meio de integração torna a coordenação de metadados transparente sob a perspectiva do usuário. A avaliação de uma integração parcial demonstra a viabilidade dessa proposta.

A terceira contribuição consiste no sistema denominado FITS, cuja função é a de orquestrar a execução dos sistemas RafeStore e DORADO. A arquitetura do FITS permite, em um único protocolo, a aplicação da técnica AdP no armazenamento de dados e a utilização da coordenação transparente de metadados do DORADO. Dessa maneira, obtém-se um sistema completo de armazenamento de dados eficiente, confiável e apropriado para o uso de múltiplos provedores de computação em nuvem.

1.1 OBJETIVOS

O objetivo geral desta tese é demonstrar que o armazenamento de dados multiversão de conteúdo independente em provedores de nuvens é mais eficiente com antecipação de pedidos e coordenação de metadados integrada.

Os objetivos específicos são relacionados a seguir:

- Especificar uma maneira de melhorar a eficiência do armazenamento de dados em múltiplos provedores em nuvem. Justificativa: tolerar faltas bizantinas é notoriamente custoso, em relação à latência observada pelo cliente e consumo de recursos. A melhoria desse processo torna-se, portanto, relevante.
- Especificar uma maneira de incorporar coordenação de metadados em um ambiente de gerenciamento de *containers*. Justificativa: incorporar coordenação de metadados a um ambiente torna o controle de concorrência transparente para o usuário, simplificando a operação do sistema.
- Especificar uma maneira de compor as soluções de armazenamento eficiente de dados e coordenação de metadados integrada em uma única solução. Justificativa: clientes que utilizam a solução composta obtêm os benefícios fornecidos pelas soluções individuais.

1.2 DELIMITAÇÕES

Aguns aspectos no contexto de armazenamento de dados não são considerados nesta tese. Esses aspectos são descritos a seguir.

Esta tese não discorre sobre confidencialidade ou privacidade de dados. Apesar do provimento de confidencialidade a dados armazenados em nuvens poder ser feito de maneira simples, como cifrar o dado e armazenar as chaves no provedor, essa abordagem requer confiança no provedor. O presente trabalho não considera possível confiar no provedor, pois julga que os provedores podem apresentar comportamento malicioso ao armazenar dados (Seção 4.1.2). Outra possibilidade ainda mais simples é manter as chaves junto ao cliente. Contudo, essa abordagem é considerada inapropriada (CACHIN; HAAS; VUKOLIC, 2010), porque a perda das chaves implica na perda dos dados.

Uma solução conhecida para prover confidencialidade consiste em particionar a chave e distribuir as partes entre as réplicas (SHAMIR, 1979). No contexto desta tese, a distribuição poderia ser feita entre diferentes provedores de nuvem (BESSANI et al., 2013). O particionamento requer a divisão da chave em, no mínimo, **três** partes (Seção 2.2.4). A recuperação da chave pode ser feita com pelo menos duas partes. Essa solução não é aplicável neste estudo, pois busca-se reduzir o armazenamento de dados para apenas **duas** partes (Seção 4.1). Manter a chave junto ao provedor de metadados também não é viável, porque a solução apresentada neste documento considera que os metadados são mantidos em uma única nuvem (*cluster* do Kubernetes).

As soluções de armazenamento de dados consideradas neste trabalho não foram projetadas para uso de dados volumosos (*Big data*). Não foram realizados experimentos com grandes volumes de dados.

As técnicas de replicação consideradas neste documento não incluem estratégias *peer-to-peer* (p2p). Aplicações p2p fornecem consistência fraca (CHO; AGUILERA, 2012), enquanto as soluções aqui tratadas utilizam consistência forte (Seção 5.2.1). As réplicas consideradas nesta tese são definidas por conjuntos fixos. Reconfiguração é abordada (Seção 5.2.1), mas questões relativas à frequência de atualização do conjunto de réplicas do sistema não são tratadas neste estudo.

Os algoritmos apresentados nesta tese não tratam sobre particionamento de dados. Particionar significa armazenar um dado em réplicas específicas, segundo critério determinado (exemplos: *round-robin*, *hashing*, faixa de valores e campos específicos em servidores diferentes). As contribuições descritas nesta tese assumem que os dados são armazenados em todas as réplicas.

Aspectos legais de armazenamento não são tratados. Por motivos de lei, dados podem ser transferidos de uma nuvem para outra. Por exemplo, no Airbnb, a partir de 01 de novembro de 2016, os dados sobre residências chinesas em aluguel foram transferidos para o *data center* da Airbnb sob jurisdição da China (Apêndice A, item 40). Em outro caso ocorrido em 17 de novembro de 2016, a Rússia bloqueou o uso do LinkedIn por 24hs, devido à inconformidade com uma lei que requer o armazenamento de dados de cidadão russos em servidores russos (Apêndice A, item 63). O protocolo apresentado neste estudo (Seção 4.1) usa apenas a latência para melhorar a eficiência de armazenamento, sem considerar aspectos legais.

As soluções propostas nesta tese foram projetadas para banco de dado NoSQL chave-valor (Seção 2.2.1). É possível que as soluções apresentadas sejam aplicáveis em bancos de dados SQL ou sistemas de arquivos. Porém, as soluções propostas não se destinam a esses tipos de armazenamento (ainda que as arquiteturas de armazenamento descritas na Seção 3.3 implementem sistemas de arquivos).

1.3 ORGANIZAÇÃO DO TEXTO

O presente capítulo introduz o contexto de armazenamento de dados em provedores de nuvem, bem como a motivação para tratar dos problemas elencados. São descritos os objetivos da tese e são apresentadas delimitações sobre assuntos abordados no presente trabalho. Deste ponto em diante, o texto está organizado conforme descrito a seguir.

- **Capítulo 2:** contém os fundamentos necessários ao entendimento dos trabalhos relacionados ao tema desta tese.
- **Capítulo 3:** descreve a literatura pertinente, por meio de alguns dos principais trabalhos relacionados ao tema. São identificadas oportunidades de pesquisa, levando em conta os estudos relacionados e o contexto apresentado no Capítulo 1.
- **Capítulo 4:** descreve o sistema *RafeStore*, que melhora a eficiência do armazenamento de dados em múltiplos provedores de nuvem. Essa solução utiliza a técnica de Antecipação de Pedidos, também desenvolvida nesse Capítulo.
- **Capítulo 5:** contém o sistema *DORADO*, que incorpora coordenação de metadados ao Kubernetes, por meio de integração.
- **Capítulo 6:** define o sistema *FITS*, que permite o uso do RafeStore em conjunto com o DORADO, a fim de permitir a obtenção dos benefícios providos por estas duas soluções.

- **Capítulo 7:** descreve a avaliação das soluções *RafeStore* e *DORADO*, propostas nesta tese.
- **Capítulo 8:** contém as considerações finais, as contribuições e trabalhos futuros.

2 FUNDAMENTAÇÃO

Encontre coisas que brilham, e mova-se em direção a elas.
Mia Farrow

Neste capítulo são apresentados os conceitos fundamentais para a compreensão da literatura relacionada a esta tese (Capítulo 3) e das propostas (Capítulos 4, 5 e 6). São abordados os conceitos sobre sistemas distribuídos, armazenamento de dados e virtualização em nível de sistema.

2.1 SISTEMAS DISTRIBUÍDOS

Nesta seção são descritos conceitos básicos de sistemas distribuídos: replicação, modelos de sincronismo, tolerância a faltas, consenso e sistemas de quórum.

2.1.1 Replicação

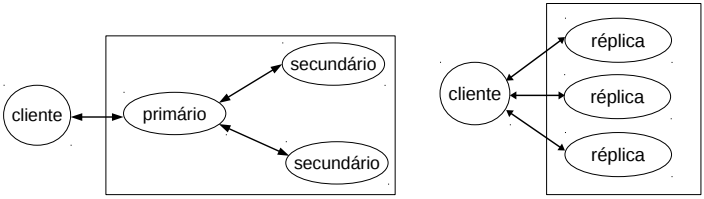
Replicação é uma estratégia usada para aumentar a disponibilidade e o desempenho de sistemas distribuídos. A disponibilidade é favorecida pela capacidade que o sistema replicado tem de continuar funcionando mesmo quando uma réplica para de funcionar (COULOURIS; DOLLIMORE; KINDBERG, 2012). No caso de uma réplica se tornar indisponível, outras podem atender às requisições dos clientes. A leitura de dados em sistemas de informação pode obter melhor desempenho replicando-se esses dados próximo aos clientes (CACHIN; GUERRAOUI; RODRIGUES, 2011).

A replicação de dados imutáveis é simples: basta aumentar o número de réplicas. Nesse caso, pode ser usado um balanceador de carga para distribuir requisições entre as réplicas. A replicação de dados mutáveis requer ações adicionais para manter as réplicas atualizadas. A considerações a partir deste ponto referem-se a réplicas que contém dados mutáveis (ou réplicas que possuem estado mutável).

Duas abordagens de replicação são descritas a seguir: replicação passiva e replicação ativa (Figura 1). A replicação **passiva** possui outras denominações, como replicação primário-secundário, *primary-backup* e mestre-escravo (BUDHIRAJA et al., 1993). Nesse modelo

(Figura 1(a)), uma réplica é a principal (primária) e as demais são secundárias. O cliente comunica-se somente com a réplica primária para usar o serviço provido. A réplica primária executa as operações e envia cópias do dado atualizado para as réplicas secundárias. Se o primário falhar, uma das réplicas secundárias é promovida e atua como primária. O processo de troca de primário é conhecido como **eleição**. O cliente recebe a resposta após o primário executar o pedido e obter a confirmação de que as réplicas secundárias atualizaram o estado.

Figura 1 – Replicação (a) passiva e (b) ativa.



Fonte: próprio autor (2017).

Na replicação **ativa** (Figura 1(b)), não existe distinção de papéis e todas as réplicas atuam de maneira equivalente. O cliente envia o pedido a todas as réplicas e cada uma processa o pedido. Se houver falha em algumas delas, não há impacto no desempenho porque cada réplica envia a resposta do pedido para o cliente. Quando muitos clientes acessam simultaneamente uma réplica primária (Figura 1(a)), essa réplica realiza o controle de concorrência localmente. Na replicação ativa (Figura 1(b)), o acesso simultâneo de clientes precisa ser coordenado.

Outras formas de replicação são tratadas no capítulo de trabalhos relacionados. No modelo de replicação em **cadeia**, o cliente comunica-se apenas com uma réplica (semelhante à replicação passiva), e cada réplica comunica-se apenas com uma única outra réplica (RENESSE; SCHNEIDER, 2004). A transmissão de pedidos entre as réplicas ocorre formando uma espécie de *pipeline*. Esse tipo de replicação pode ser utilizado para propagar dados entre réplicas (exemplo: Gnothi, Seção 3.1.5). Em alguns cenários não existe a necessidade de armazenar o dado em todas as réplicas (exemplo: SPANStore, Seção 3.1.6). A replicação **parcial** consiste em replicar apenas um subconjunto de dados nas réplicas (TANENBAUM; STEEN, 2007). Nesse caso, nem todas as réplicas são idênticas.

2.1.2 Modelos de sincronismo

Processos distribuídos (por exemplo, cliente e réplica, ou duas réplicas) comunicam-se a fim de trocar informações. Em sistemas distribuídos é difícil definir limites de tempo para a execução de processos, entrega de mensagens em rede ou diferença entre relógios físicos dos processos (COULOURIS; DOLLIMORE; KINDBERG, 2012). Os dois principais modelos de tempo que fazem considerações sobre esses limites temporais são os modelos síncrono e assíncrono.

Em sistemas distribuídos **síncronos** são conhecidos diversos limites temporais (SIMONS, 1990; HADZILACOS; TOUEG, 1994):

- Existem limites inferior e superior para o tempo de execução nos processos, e são conhecidos.
- Existe um limite superior (Δ) conhecido para o tempo de transmissão de mensagens na rede.
- Existe desvio do relógio de cada processo, em relação a uma referência padrão.

O modelo síncrono permite a obtenção *a priori* dos limites e desvios mencionados. Dessa maneira, serviços como detecção de atrasos no envio de mensagens e coordenação de estados baseada em tempo podem ser implementados em sistemas distribuídos síncronos (CACHIN; GUERRAOUI; RODRIGUES, 2011). É comum o uso de **temporizadores** em sistemas síncronos. Um temporizador indica quando um tempo máximo foi alcançado (expirou). Comunicações que excedem o tempo máximo de transmissão (*timeout*) podem ser consideradas problemáticas, e ações de correção podem ser tomadas (por exemplo, reenvio de mensagens).

Em sistemas distribuídos **assíncronos**, não é possível estabelecer valores para tempos de execução de processos, tempos de transmissão de mensagens ou desvio de relógios. O modelo assíncrono de tempo não faz qualquer suposição sobre intervalos de tempo envolvidos na execução da comunicação[1]. Uma relação considerável é a de que qualquer sistema desenvolvido como um sistema distribuído assíncrono opera corretamente também em um ambiente síncrono. Soluções desenvolvidas sob o modelo assíncrono são, desse modo, mais portáveis (funcionam no sistema síncrono).

Sistemas que operam em ambientes assíncronos não consideram as relações temporais mencionadas, mas são expostos a outras restrições. Por exemplo, em um sistema assíncrono é difícil descobrir o mo-

[1] Considera-se como execução da comunicação o envio de um pedido a um destinatário, a execução do pedido pelo destinatário e a entrega da resposta ao emissor.

tivo pelo qual uma mensagem não alcança seu destino. A rede pode ser motivo da instabilidade, causando lentidão de transmissão. Outra possível causa é o processo destinatário estar lento, provocando uma demora no processamento. Nesse cenário de dúvidas surgiu a **impossibilidade FLP**[2] (FISCHER; LYNCH; PATERSON, 1985). A impossibilidade FLP é um resultado que surge quando as restrições temporais do sistema síncrono não são atendidas. Segundo esse resultado, não há garantias de que seja possível alcançar consenso em ambientes assíncronos se houver pelo menos um processo que possa apresentar comportamento incorreto. O consenso será discutido adiante (Seção 2.1.4).

Outro modelo relevante no contexto desta tese é o ambiente **parcialmente síncrono** (DWORK; LYNCH; STOCKMEYER, 1988). Esse ambiente tem as mesmas premissas de um ambiente síncrono, exceto o fato de que a transmissão de mensagens em rede (Δ) pode variar conforme dois cenários descritos a seguir.

No primeiro cenário, o limite superior para o tempo de transmissão de mensagens na rede (Δ) existe, mas é desconhecido *a priori*. Isso cria dificuldade para verificar o funcionamento dos sistemas. O uso de um Δ pequeno pode caracterizar erro. A escolha de um Δ grande pode prolongar a descoberta de uma situação de erro.

O segundo cenário consiste na existência de Δ, no seu conhecimento *a priori* (condição idêntica a dos sistemas síncronos), mas na instabilidade dos componentes da comunicação. Um canal de comunicação instável pode apresentar comportamento idêntico ao de uma rede assíncrona (na qual se aplica a impossibilidade FLP). Para lidar com tal situação, define-se uma constante denominada tempo de estabilização global (*global stabilization time* - GST). Leva-se em conta também um instante de tempo L, no qual GST já transcorreu. Assume-se, desse modo, que qualquer mensagem enviada após L será recebida respeitando o limite Δ. Considera-se esse cenário, na prática, em ambientes assíncronos que apresentam intervalos de comportamento síncrono.

Sistemas parcialmente síncronos possuem informação sobre duas da três características de um sistema síncrono (tempo de execução dos processos, tempo de comunicação de rede e diferença entre relógios dos processos) (HADZILACOS; TOUEG, 1994).

[2]FLP - sigla adotada pela comunidade científica que se refere às iniciais dos autores Fisher, Lynch e Patterson.

2.1.3 Tolerância a faltas

A replicação de dados é utilizada para aumentar a disponibilidade (Seção 2.1.1). O serviço provido pelo sistema replicado é **correto** quando implementa adequadamente a funcionalidade para a qual o sistema foi projetado. Um sistema replicado, entretanto, pode manifestar comportamentos incorretos. Uma **falha** no serviço ocorre quando o resultado do serviço é diferente do serviço correto (AVIZIENIS et al., 2004). O funcionamento incorreto do serviço é chamado **erro**. A causa do erro é denominada **falta**. Faltas podem causar erros, e erros podem gerar falhas.

Para aumentar a confiabilidade de sistemas, pode-se atuar no âmbito das faltas, ou seja, prevenir, tolerar, remover e prever faltas. A presente tese atua no sentido de *tolerar faltas*. Isso significa criar mecanismos para que a ocorrência de uma falta não gere um erro e, consequentemente, não venha a provocar falhas em um sistema. Nesse contexto, admite-se a ocorrência de faltas. Estas, porém, são mascaradas (toleradas), no sentido de não causar um erro. Dessa maneira, evita-se a falha.

Faltas podem ser originadas por diversos fatores. É possível categorizar faltas em função do comportamento incorreto apresentado pelos processos (MELLIAR-SMITH; MOSER; AGRAWALA, 1990; DÉFAGO; SCHIPER; URBÁN, 2004):

- Falta de **parada** (*crash*): o processo para definitivamente de funcionar, deixa de realizar qualquer processamento ou comunicação.
- Falta de **omissão**: o processo deixa de realizar alguma ação (como enviar ou receber mensagens).
- Falta **temporal** (*timing*): o processo viola restrições temporais. Esse tipo de falta não se aplica em sistemas assíncronos (Seção 2.1.2).
- Falta **bizantina**: o processo realiza um comportamento diferente do esperado. Por exemplo: duplica mensagens, altera conteúdo de mensagens ou tenta atuar para causar uma falha no sistema.

Existem classificações que separam as faltas bizantinas em faltas de **resposta** e faltas **arbitrárias** (TANENBAUM; STEEN, 2007). Considera-se falta de resposta[3] quando a resposta fornecida pelo serviço é incorreta. Faltas arbitrárias estão associadas a qualquer situação ou comportamento que possa provocar um erro.

Faltas bizantinas (sejam de resposta ou arbitrárias) podem ter

[3] Esse tipo de falta é associado à resposta fornecida pelo servidor.

origem maliciosa. Valores incorretos podem ser resultantes de um envio intencionalmente alterado no processo respondente. O próprio processamento pode ter sido alterado a fim de apresentar resultados errôneos. Um fato relevante é que para tolerar faltas bizantinas é preciso utilizar replicação ativa (Seção 2.1.1). Até o momento não é possível tolerar faltas bizantinas usando-se replicação passiva. O problema dos generais Bizantinos é um problema que assume faltas maliciosas, e deu origem ao nome da categoria, sendo exemplo clássico de tolerância a faltas maliciosas (LAMPORT; SHOSTAK; PEASE, 1982). Uma relação importante expressa no contexto dos generais Bizantinos leva em consideração que para tolerar faltas maliciosas de até f membros, é necessário atender a relação $n = 3f + 1$, tal que n é o número de membros do sistema. Nesse contexto, a comunicação entre as réplicas é assinada para garantir a autenticidade das mensagens (DOLEV; STRONG, 1983). Consequentemente, o gerenciamento das chaves de segurança torna-se um ônus para o sistema.

2.1.4 Consenso

Quando processos atuam conjuntamente, há situações em que é preciso tomar decisões. Por exemplo, no problema dos generais Bizantinos, os generais precisam decidir se atacam ou aguardam. Em uma transação de transferência de valores, os computadores envolvidos devem concordar se realizarão o crédito e o débito respectivos (COULOURIS; DOLLIMORE; KINDBERG, 2012, p.659). Nessas tomadas de decisão, os processos devem chegar a um *consenso*. Pode-se definir consenso informalmente do seguinte modo: "cada processo *propõe* um valor. Todos os processos precisam, em algum momento, *decidir* pelo mesmo valor. O valor decidido precisa ser um dos valores propostos" (DÉFAGO; SCHIPER; URBÁN, 2004).

De maneira formal, o consenso pode ser especificado em termos de dois eventos denominados *propose* e *decide* (CACHIN; GUERRAOUI; RODRIGUES, 2011, p.204). Cada processo participante do consenso tem um valor inicial v que é proposto com uma mensagem *propose*. Todos os processos corretos propõem um valor inicial. Os processos comunicam-se a fim de trocar entre si os valores propostos e todos os processos corretos precisam decidir por um mesmo valor. Em seguida, manifestam essa decisão informando o valor escolhido no envio de uma mensagem *decide*. O consenso satisfaz quatro propriedades, descritas a seguir:

- Terminação: cada processo correto decide, em algum momento, por algum valor.
- Validade: se um processo decide um valor v, então v foi proposto por algum processo.
- Integridade: nenhum processo decide duas vezes.
- Acordo: nenhum processo correto decide um valor diferente de outro processo correto.

Quando um sistema possui réplicas com dados mutáveis e a replicação é ativa, torna-se necessário *coordenar* a execução das requisições (i.e., chegar a um consenso sobre a ordem de execução dos pedidos nas réplicas). Uma das maneiras mais utilizadas para coordenar a execução de requisições em sistemas replicados é a **replicação de máquinas de estado (RME)** (SCHNEIDER, 1990). A regra de coordenação da RME declara que *todas as réplicas recebem e processam os pedidos na mesma sequência*. Nesse caso, protocolos de difusão de ordem total podem ser utilizados para implementar tal comportamento. Duas propriedades devem ser atendidas pelas réplicas participantes da RME:

- Acordo: cada réplica correta recebe cada requisição.
- Ordem: cada réplica correta processa cada requisição recebida na mesma ordem que as demais réplicas.

As propriedades mencionam réplicas *corretas* porque em geral os protocolos RME são tolerantes a faltas (Seção 2.1.3).

Dentre os protocolos que implementam RME tolerantes a faltas, o **Paxos** é um dos algoritmos mais conhecidos (LAMPORT, 1998). De maneira simplificada e resumida, o algoritmo do Paxos compõe-se de três fases (CHANDRA; GRIESEMER; REDSTONE, 2007):

- Uma réplica é eleita para atuar como coordenador[4].
- O coordenador seleciona um valor e difunde-o às demais réplicas por meio de uma mensagem *Accept*. As réplicas aceitam ou rejeitam essa mensagem.
- Quando a maioria de réplicas aceita o valor proposto, o coordenador envia uma mensagem *Commit* às demais réplicas.

A implementação do Paxos não é uma tarefa trivial, fato que motivou o desenvolvimento do algoritmo Raft, mencionado mais adiante (Seção 3.2.3). O Paxos tolera faltas de parada.

No contexto de faltas bizantinas, o PBFT (CASTRO; LISKOV et al., 1999) foi o primeiro trabalho baseado no Paxos a viabilizar a execução de RME tolerante a faltas bizantinas. O PBFT utiliza um sequenciador fixo (DÉFAGO; SCHIPER; URBÁN, 2004) para decidir

[4]Outras nomenclaturas conhecidas para este papel são *sequenciador* (DÉFAGO; SCHIPER; URBÁN, 2004) e *líder*.

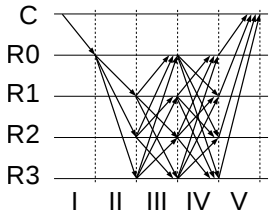


Figura 2 – PBFT: protocolo.

a ordem das requisições. A execução do PBFT (Figura 2) inicia com o cliente enviando o pedido à réplica líder (fase I, denominada *REQUEST*). A seguir, o líder ordena o pedido e envia às demais réplicas seguidoras (fase II, denominada *PRE-PREPARE*). As réplicas seguidoras trocam informações entre si, a fim de assegurar a veracidade da ordenação proposta pelo líder (fase III, denominada *PREPARE*), e o líder também recebe essa troca de informações. Cada réplica procura acumular a quantidade mínima (2f+1) de confirmações necessárias para ratificar a ordem apresentada pelo líder (confirmações iguais).

O passo seguinte consiste em nova troca de informações (fase IV, denominada *COMMIT*), que tem por objetivo garantir que as réplicas aceitaram a ordenação do líder e efetivarão o pedido. A confirmação de aceitação é necessária devido à possibilidade de mudança de visão (mudança de líder). Dessa maneira, é garantido que réplicas corretas vão executar pedidos na ordem correta, mesmo que esses pedidos tenham sido efetivados em visões anteriores. A outra razão que motiva a troca de aceitações (fase IV) é a garantia de que um pedido efetivado localmente será também efetivado em pelo menos $2f+1$ réplicas. Após acumular aceitações de uma maioria de réplicas, cada réplica pode executar o pedido e responder ao cliente (fase V, denominada *REPLY*).

O PBFT requer sincronia parcial para que o protocolo chegue até o fim. Protocolos que usam um sequenciador fixo (líder) precisam de temporizadores para monitorar o funcionamento do líder. Em caso de ausência de atuação do líder, uma nova eleição é necessária. Na versão original do PBFT, o cliente envia o pedido apenas para a réplica líder. Se o cliente não recebe resposta do sistema após um determinado tempo, o cliente reenvia o pedido a todas as réplicas. Em versão posterior do PBFT, o cliente envia o pedido para todas as réplicas logo na primeira vez (CASTRO; LISKOV, 2002).

Uma implementação estável e disponível de RME é o BFT-SMaRt (BESSANI; SOUSA; ALCHIERI, 2014). O BFT-SMaRt im-

plementa transferência de estado e reconfiguração. A reconfiguração permite a saída e a entrada de réplicas no sistema e a transferência de estado sincroniza novas réplicas com o estado global replicado.

2.1.5 Sistemas de quórum

Há várias técnicas de replicação de dados. Dentre elas, os *quóruns* (GIFFORD, 1979) destacam-se por manter a disponibilidade do sistema mesmo na presença de faltas. No contexto de armazenamento de dados, quóruns atuam basicamente dispensando a necessidade de todas as réplicas responderem a uma requisição. As operações de leitura e escrita de dados são enviadas a todas as réplicas, mas são necessárias apenas respostas de um quórum de réplicas para a conclusão da operação. Quando essa quantidade consiste de uma maioria de réplicas, garante-se que existe uma *interseção* de réplicas que executam as operações. O dado é multiversão (Seção 2.2.2), permitindo assim verificar qual é o dado mais recente. Por exemplo, se um grupo tem 5 participantes, basta que 3 respondam às operações de leitura e escrita para que o sistema como um todo permaneça consistente.

Para tolerar faltas bizantinas, mais réplicas devem ser adicionadas (MALKHI; REITER, 1998). Quóruns contam com $3f + 1$ réplicas para tolerar faltas bizantinas. Clientes escritores precisam compartilhar uma chave criptográfica para garantir a autenticidade do valor (MARTIN; ALVISI; DAHLIN, 2002). Isso é necessário para evitar que uma réplica maliciosa crie um valor espúrio, que poderia ser considerado correto. A versão do dado também precisa ser protegida dessa maneira. Em geral não se consideram clientes escritores maliciosos em sistemas de quóruns, porque é difícil evitar que o cliente escreva valores diferentes, associados à mesma versão, em réplicas diversas.

Tolerar clientes escritores Bizantinos é possível, mas esforço extra é requerido (LISKOV; RODRIGUES, 2006). Nesse caso, etapas de comunicação adicionais podem ser requeridas, como operações de *write-back*[5], nas quais leitores têm permissão de escrita e podem reparar metadados incorretos (ATTIYA; BAR-NOY; DOLEV, 1995).

[5] *Write-back* é utilizado no Vivace, Seção 3.1.4.

2.2 ARMAZENAMENTO DE DADOS

Esta seção contém conceitos básicos relacionados ao armazenamento de dados: bancos de dados chave-valor, consistência, integridade e fragmentação de dados. Aborda-se, também, o conceito de provedor de armazenamento em nuvem, devido ao contexto desta tese. Adicionalmente, são descritas ferramentas de comunicação de grupo, que usam armazenamento para persistir a configuração que define os membros de um grupo.

2.2.1 Bancos de dados chave-valor

Um banco de dados (BD) pode ser definido como uma coleção de dados relacionados, e os dados são fatos conhecidos que podem ser registrados (ELMASRI; NAVATHE, 2011, p. 3). Um Sistema Gerenciador de Banco de Dados (SGBD) opera sobre o BD. O SGDB controla a concorrência de modo a garantir o acesso simultâneo de usuários a um mesmo dado. Os BD *relacionais* (CODD, 1970) surgiram com o objetivo de oferecer uma visão mais natural da descrição dos dados, em comparação a modelos de grafos ou de rede. O usuário não precisaria mais saber o formato dos dados para então acessá-los.

Apesar do predomínio de uso de BD relacionais, os BD não-relacionais existem desde 1960 (LEAVITT, 2010). Esses BD são também conhecidos como NoSQL, possível abreviação de "not only SQL". O armazenamento é realizado de diversas formas, a exemplo do hierárquico, dos grafos e dos orientados a objetos. Isso decorre do fato de que alguns tipos de dados são melhores organizados quando armazenados em estruturas que não sejam tabelas de BD relacionais.

Instâncias volumosas de BD NoSQL têm demonstrando o seu potencial de escalabilidade. Em 2007 a Amazon tornou pública a especificação do sistema Dynamo, que armazena muitas informações importantes da Amazon (DECANDIA et al., 2007). O Facebook apresentou, em 2009, detalhes de seu banco de dados nomeado Cassandra, desenvolvido para atender à demanda de buscas nas caixas de mensagens (LAKSHMAN; MALIK, 2010).

2.2.2 Consistência

Tradicionalmente, consistência de um dado é discutida no contexto de operações de leitura e escrita sobre um dado compartilhado (TANENBAUM; STEEN, 2007). O compartilhamento se refere ao fato de o leitor e o escritor de um dado não serem necessariamente o mesmo ator. Diferentes atores, com diferentes papéis, podem atuar simultaneamente sobre um mesmo dado.

A primeira premissa a ser considerada é a de que apenas um escritor pode alterar o dado por vez. Uma designação comum para esse caso é *Single Writer, Multiple Readers* (SWMR). A atuação simultânea de leitores não gera problemas quanto à consistência do dado. Entretanto, a atuação simultânea de leitores e um escritor pode apresentar comportamentos diferentes. Pode-se estabelecer, então, uma analogia entre o dado compartilhado e um registrador (LAMPORT, 1986). Assim, o comportamento observado a partir da interação com um registrador pode ser classificado como seguro, regular ou atômico. Esses comportamentos são descritos a seguir.

Um registrador **seguro** garante que uma leitura não concorrente com uma escrita observará o último valor escrito. Caso a leitura ocorra de forma concorrente a uma escrita não é possível definir com exatidão qual será o valor retornado. Apenas pode-se afirmar que o valor será um valor válido, isto é, um valor existente no conjunto de possíveis valores que podem ser atribuídos ao registrador.

A próxima consistência é denominada **regular**, a qual mantém as propriedades da consistência segura. Além disso, estabelece que, durante uma leitura e uma escrita concorrentes no registrador, o valor lido retornará o valor existente antes da escrita, ou o valor depois da escrita. Por exemplo, considere um registrador regular que contém o valor 1. Durante a escrita do valor 3, uma leitura simultânea a essa alteração somente poderá retornar os valores 1 ou 3. Com essa consistência, diferentes leitores simultâneos podem observar diferentes valores (1 ou 3, no exemplo mencionado).

Por fim, a consistência **atômica** inclui as propriedades da consistência segura e adiciona a noção de ordem total (Seção 2.1.4) às operações. As operações de leitura e escrita são tratadas de maneira a corresponder a uma execução sequencial única e definida, sem sobreposição. Nesse cenário, diferentes leitores simultâneos observam os mesmos valores, de acordo com uma sequência temporal, exemplificada a seguir. Situe-se uma operação de escrita que atualiza um registrador atômico k do valor x para o valor y. Leitores simultâneos de k obser-

vam o valor x ou y (assim como na consistência regular). Entretanto, a partir do momento que um leitor qualquer observa o valor y, todos os demais leitores também observam apenas o valor y.

Registradores com consistência atômica permitem a construção de sequências lineares das operações (HERLIHY; WING, 1990). Portanto, um sistema de armazenamento oferece semântica **linearizável** quando a consistência de seus dados é atômica (exemplos: Gnothi, Seção 3.1.5; SPANStore, Seção 3.1.6, e Hybris, Seção 3.1.8).

Uma forma de manter leitores atualizados sobre escritas é usar o padrão *listener* (MARTIN; ALVISI; DAHLIN, 2002). Após uma solicitação de leitura de um dado, a réplica mantém o cliente em uma lista. Em caso de atualizações do dado (escritas), a réplica avisa aos clientes existentes na lista sobre essa atualização. Ao concluir a leitura, o cliente envia uma mensagem à réplica, informando que não precisa mais acompanhar a atualização do dado (mensagem *read_ complete*).

Pode-se distribuir um registrador para aumentar sua resiliência. Registradores compartilhados distribuídos (RCD) são implementados utilizando-se tecnologia de virtualização (SILVA et al., 2013a).

Os modelos de consistência para dados compartilhados são difíceis de implementar de maneira eficiente em sistemas distribuídos de larga escala (TANENBAUM; STEEN, 2007). Replicação em sistemas de larga escala é útil para, por exemplo, melhorar o desempenho de acesso a dados (CHO; AGUILERA, 2012). No caso de dados georeplicados, clientes leitores obtêm o dado da réplica mais próxima. No cenário de sistemas largamente distribuídos, a consistência é classificada como: forte, fraca e eventual (VOGELS, 2009), descritas a seguir.

Na consistência **forte**, depois que uma atualização é concluída, qualquer acesso subsequente retornará o valor atual, até que uma nova atualização ocorra. É difícil implementar consistência forte em sistemas geo-distribuídos. A visão linear de operações (linearização) é apropriada apenas para sistemas cujo tempo de resposta é crítico (HERLIHY; WING, 1990). Um exemplo de aplicação que requer consistência forte é a edição colaborativa de documentos (WU et al., 2013).

Em uma consistência **fraca**, não há qualquer garantia de que acessos subsequentes à atualização irão retornar o valor atual. O período entre a atualização do dado e o momento a partir do qual pode-se garantir que qualquer cliente irá observar o novo valor é conhecido como janela de inconsistência.

A consistência **eventual** é um tipo de consistência fraca na qual existe a garantia de que, após uma atualização sem subsequentes atualizações, todos os leitores irão observar o valor atualizado. A janela

de inconsistência pode ser determinada com base em fatores como a latência de rede, carga de processamento e número de réplicas. Por exemplo, plicações de rede social são satisfeitas, de modo geral, com consistência eventual (WU et al., 2013).

É comum "relaxar" a consistência em sistemas geo-distribuídos. O Teorema CAP (***C**onsistency, **A**vailability and **P**artition tolerance*) (GILBERT; LYNCH, 2002) afirma a impossibilidade de um sistema geograficamente distribuído prover, simultaneamente, garantias de consistência, disponibilidade e tolerância a particionamento de rede, cujas características são descritas a seguir. A consistência considerada forte (ou atômica) é aquela cujo serviço provido pelas réplicas ocorre como se, do ponto de vista do cliente, existisse apenas um único sistema (GILBERT; LYNCH, 2012). Disponibilidade refere-se ao fato de que cada requisição enviada pelo cliente será respondida em algum momento. Particionamento de rede significa a incapacidade de réplicas comunicarem-se entre si. Rupturas na rede, lentidão ou perda de mensagens podem caracterizar um particionamento de rede.

Os sistemas, na prática, priorizam duas dessas características em detrimento da terceira. Algoritmos como o Paxos (Seção 2.1.4) e o Raft (Seção 3.2.3) fornecem consistência forte, mas não são escaláveis em ambientes geograficamente distribuídos. Há esforços para reduzir a latência na operação de sistemas que provêm consistência forte em ambientes geo-distribuídos por meio de configurações que aumentam a representatividade de algumas réplicas (SOUSA; BESSANI, 2015).

Existe uma semântica denominada *fork* na qual escritas simultâneas geram ramificações na sequência de valores de um dado (MAZIERES; SHASHA, 2002). Cada cliente observará o dado gravado segundo a sua própria sequência (criada devido à concorrência de escrita) e pode-se resolver essa divisão com uma operação denominada *join* (MAHAJAN et al., 2011). Nesse caso, os clientes podem comunicar-se fora do sistema replicado, ou podem usar algum critério determinístico para resolver o conflito. É possível detectar conflitos de forma antecipada, ao invés de aguardar leituras subsequentes. Há ferramentas que avisam os clientes sobre alterações em dados (Seção 2.2.6), permitindo-lhes identificar operações de escrita concorrentes.

O controle de concorrência no contexto de transações originou o surgimento do conceito de dado **multiversão** (BERNSTEIN; GOODMAN, 1983). A cada nova escrita do dado, uma nova cópia é criada, ao invés de sobrescrever o dado. Apenas a nova versão do dado é sobrescrita, resultando num valor seguinte (incremental) ao valor anterior. Diversos trabalhos (Seção 3.1) realizam controle de concorrência so-

bre dados multiversão (*Multi-version concurrency Control* - MVCC). É comum denominar o número de versão como *timestamp*. Em geral, esse número é um inteiro não-negativo monotônico[6] sequencial. O cenário com múltiplos escritores (*Multiple Writer, Multiple Readers* - MWMR) pode utilizar *timestamps* para ordenar escritas, adicionando um identificador do cliente. O *timestamp* multi-escritor permite fornecer a semântica MWMR (ATTIYA; WELCH, 2004, p. 220)(CACHIN; GUERRAOUI; RODRIGUES, 2011, p. 162).

As operações de escrita e leitura estão sujeitas à semântica de operação oferecida pelas réplicas. Um algoritmo pode oferecer liberdade de obstrução (*obstruction-freedom*) quando garante que, se um cliente correto invoca uma operação *op* e não existem outros clientes realizando atividades relacionadas, *op* será executada em algum momento (HERLIHY; LUCHANGCO; MOIR, 2003). Por exemplo, um leitor só termina se não houver escritores paralelos acessando o mesmo dado. O cliente tem garantia de execução sempre que executar uma ação em modo "isolado".

A semântica que provê liberdade de espera (*wait-freedom*) oferece uma "garantia mais independente" (HERLIHY, 1991). Se um cliente invoca *op*, então *op* será executada em algum momento, independente da operação de outros clientes. Por exemplo, um leitor pode concluir sua operação independentemente da existência de múltiplos escritores atuando simultaneamente à leitura.

2.2.3 Integridade

O algoritmo RSA (RIVEST; SHAMIR; ADLEMAN, 1978) de criptografia de chave pública é frequentemente usado para garantir a autenticidade de dados armazenados em provedores de nuvem (exemplos: ICStore, Seção 3.1.2; DepSky, Seção 3.1.3). É possível verificar também a integridade dos dados com os algoritmos RSA, uma vez que para decifrar o dado é preciso que o mesmo esteja íntegro.

A forma usual de se verificar a integridade de dados é o uso de um resumo criptográfico (também denominado *hash*), sendo os mais conhecidos o MD5 (RIVEST, 1992) e o SHA1 (EASTLAKE 3RD; JONES, 2001). Importante situar que o MD5 já não é mais considerado seguro para uso de autenticação de dados (WANG; YU, 2005). Entretanto, o mesmo ainda é útil para fins de verificação de integridade.

[6] Uma sequência a_n é incremental monotônica se $a_{n+1} \geq a_n \forall n \in N$.

2.2.4 Fragmentação de dados

Em sistemas de armazenamento, a replicação integral de um dado tem custo diretamente proporcional à quantidade de réplicas e uma corrupção no dado compromete o dado replicado. Por esse motivo, foram criadas técnicas de fragmentação de dados, permitindo o isolamento de partes do dado. A replicação dos fragmentos provê redundância, tolerando assim a corrupção de certo número de fragmentos sem comprometer a recuperação do dado. Observa-se que a fragmentação de dados pode aumentar o tempo médio entre falhas, quando comparado à replicação integral (WEATHERSPOON; KUBIATOWICZ, 2002). Na sequência do texto, duas estratégias são apresentadas: fragmentação com replicação e espalhamento, e fragmentação com redundância.

2.2.4.1 Fragmentação, replicação e espalhamento

O trabalho seminal que estabeleceu o termo "tolerância a intrusão" também introduziu uma técnica de fragmentação e compartilhamento de dados (FRAGA; POWELL, 1985). Tal técnica foi utilizada juntamente com estratégias de gerenciamento de acesso, estabelecendo assim a técnica FRS (*fragmentation, replication and scattering*) (DESWARTE; BLAIN; FABRE, 1991). Os dados considerados pelo FRS são arquivos, e as informações são cifradas e distribuídas entre réplicas. Nesta condição, é preciso obter acesso a uma quantidade mínima de réplicas para reconstruir o dado original. A técnica FRS ocupa espaço igual ao dobro do dado original, o que equivale, portanto, a uma replicação integral (em termos de espaço ocupado). Contudo, provê tolerância a faltas e preserva a confidencialidade do dado. Caso um invasor obtenha acesso a uma réplica, o dado é ilegível.

A próxima técnica (Seção 2.2.4.2) cumpre objetivos semelhantes aos do FRS. Ela não provê confidencialidade, mas ocupa menos espaço.

2.2.4.2 Fragmentação com redundância

Os códigos de correção de erro foram criados para tolerar pequenas corrupções em dados (HAMMING, 1950). Algumas informações são agregadas ao dado original, tornando possível a recuperação de perdas de dados. Esta técnica foi inicialmente usada para corrigir perdas em sinais de comunicação e, posteriormente, adaptada para uso em

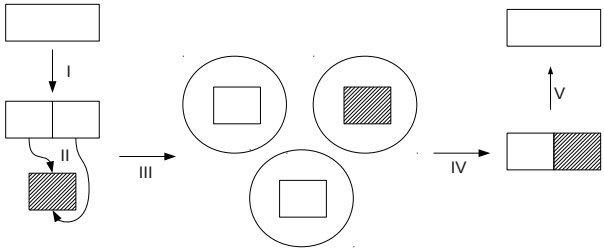
armazenamento de dados, permitindo a correção de dados em repouso.

O uso da correção de erros em dados ocorreu na técnica denominada RAID (*Redundant Array of Inexpensive Disks*) (PATTERSON; GIBSON; KATZ, 1988). Essa técnica tem por objetivo permitir a realização de armazenamento confiável utilizando dispositivos econômicos de baixa confiabilidade. Ressalta-se que os primeiros discos rígidos não eram dispositivos confiáveis. O RAID, então, permite o uso de vários discos, de modo que a substituição de discos pode ser feita sem interromper o funcionamento do sistema nem resultar em perda de dados.

A partir da fragmentação de dados realizada pelo RAID, surgiu a ideia de distribuir fragmentos em nós de uma rede de computadores. Nesse contexto, uma das primeiras técnicas de fragmentação com redundância foi a técnica de dispersão de dados (RABIN, 1989). O procedimento consiste em dividir o dado em k fragmentos e criar m fragmentos adicionais. Qualquer conjunto de k fragmentos possibilita a reconstrução do dado. Os parâmetros são definidos como (k, m).

O uso de fragmentação com redundância adiciona tolerância a faltas ao armazenamento de dados. Entretanto, esta técnica apenas tem sentido quando são utilizadas no mínimo três réplicas. Considere o caso de uma configuração (k,m)=(2,1). Inicialmente (Figura 3), o dado é dividido em dois fragmentos (passo I). A seguir, um fragmento redundante é criado (passo II), a partir dos outros dois fragmentos. Por fim, os fragmentos são distribuídos entre as réplicas (passo III). Restaura-se o dado mediante quaisquer dois fragmentos (passo IV). Basta processar esses fragmentos para obter o dado original (passo V).

Figura 3 – Fragmentação com redundância: exemplo, (k, m)=(2,1).



Fonte: próprio autor (2017).

Quando um fragmento é comprometido, diversas estratégias podem ser usadas para recuperará-lo (ABU-LIBDEH; PRINCEHOUSE; WEATHERSPOON, 2010). Pode-se fazer um monitoramento de fragmentos, e iniciar a restauração assim que forem detectadas inconsis-

tências (fragmentos corrompidos ou indisponíveis). É possível também fazer restauração dos fragmentos na medida em que os dados são lidos pelo cliente, e detecta-se que houve fragmentos não obtidos ou com falhas. Importante lembrar que a dispensa da reconstrução do fragmento pode inviabilizar a capacidade do sistema de tolerar novas faltas.

A técnica de fragmentação com redundância faz uso de *erasure codes* (EC) para implementar a criação de fragmentos redundantes, bem como para restaurar fragmentos (RABIN, 1989). Alguns trabalhos (RACS, Seção 3.1.1; DepSky, Seção 3.1.3) utilizam EC para armazenar dados em provedores de nuvem. Mais adiante, nesta tese (Seção 7.2.6), há ponderações sobre o espaço ocupado pela replicação integral e a replicação com fragmentos.

2.2.5 Provedores de armazenamento em nuvens

O armazenamento de dados remoto pode ser realizado em provedores de computação em nuvem. Computação em Nuvem pode ser definida como:

> "[...] um modelo para permitir acesso ubíquo, conveniente e sob demanda para recursos de computação compartilhados e configuráveis [...] que podem ser rapidamente provisionados e liberados com esforço de gerenciamento mínimo ou por um serviço de interação com o provedor." (MELL; GRANCE, 2010).

Provedores de armazenamento em nuvem podem ser classificados como ativos ou passivos (BESSANI et al., 2013). Provedores ativos são instanciados principalmente sob forma de máquinas virtuais (MV), e permitem a execução de código do usuário, bem como a comunicação com outros provedores. Provedores como a Amazon EC2 (Apêndice A, item 20) e o Linode (Apêndice A, item 21) fornecem MV.

Provedores passivos não possuem a capacidade de executar código do usuário, e contêm apenas primitivas que realizam apenas leitura e escrita de dados. O acesso ocorre por interfaces padronizadas (por exemplo, API's REST). Exemplos de provedores passivos são os serviços Amazon S3 (Apêndice A, item 22) e o Azure Object Storage (Apêndice A, item 23).

A consistência oferecida por serviços de armazenamento em nuvens passivas varia de acordo com o provedor e com a operação. O Amazon S3 oferece consistência eventual na atualização de dados (Apêndice A, item 24) e na listagem de objetos (Apêndice A, item 25). O

Microsoft Azure, por sua vez, oferece consistência forte em suas operações (CALDER et al., 2011).

O uso de múltiplos provedores de nuvem tem se tornado uma alternativa para aumentar a confiabilidade dos sistemas (CORREIA, 2013). O termo *Intercloud*, ou nuvem de nuvens, significa uma nuvem virtual composta de diversos (e diferentes) provedores de nuvem (BERNSTEIN et al., 2009). É possível que serviços de um provedor fiquem indisponíveis em todos os *data centers* (Apêndice A, item 12). Isso pode ocorrer porque a forma como um provedor administra seus sistemas tende a ser a mesma em todos os locais onde atua. O gerenciamento de um provedor em nuvem por uma única companhia torna o provedor um ponto único de falha (ARMBRUST et al., 2010). Os principais benefícios de se utilizar múltiplos provedores de nuvem são aumentar disponibilidade, prover tolerância a desastres e evitar dependência de provedor (GUERRAOUI; YABANDEH, 2010).

O uso de diferentes provedores de nuvem aumenta a disponibilidade de informações. Existem aplicações críticas que requerem informações sempre disponíveis. Por exemplo, sistemas de energia têm bases de dado históricos que armazenam eventos coletados de *power grids* e essas informações precisam estar sempre acessíveis para consulta (BESSANI et al., 2013).

O uso de diferentes provedores em nuvem fornece diversidade, o que favorece a independência de falhas (CACHIN; HAAS; VUKOLIC, 2010). A diversidade existente em cada provedor diminui a possibilidade de indisponibilidade simultânea de diferentes provedores. É possível elencar várias características de um provedor em nuvem: localização geográfica, modo de fornecimento de energia, arquitetura interna, enlaces de rede (rede do provedor até o cliente), conexões internas (ao provedor) de rede, gerência de administração, *middleware* e aplicações (AVIZIENIS; KELLY, 1984). Cada provedor possui uma configuração única quanto à implementação dessas características.

Diversos provedores de nuvem possuem *data centers* dispersos pelo mundo (Apêndice A, itens 54 e 55). Diferentes provedores possuem *data centers* em diferentes locais. Desse modo, o uso de vários provedores favorece a continuidade de acesso aos dados georeplicados, mesmo na ocorrência de desastres como tsunamis ou terremotos.

A diversidade advinda do uso de uma *Intercloud* proporciona independência de provedor e são diversos os motivos que podem levar um cliente a contratar ou cancelar o contrato com um provedor:

- Preços de provedores variam (Apêndice A, itens 1 a 5). Apesar de provedores apresentarem preços semelhantes, pequenas dife-

renças tornam-se significativas quando praticadas em escala. Há sistemas específicos para gerar recomendações ao usuário sobre qual provedor usar (Seção 3.1.6) com base em valores de armazenamento de dados, tráfego de rede e quantidade de operações.

- Alterações na rede dos provedores (exemplos: modificações de topologia, concorrência de rede por MVs) podem modificar as latências percebidas pelos clientes (ARMBRUST et al., 2010).
- Provedores de nuvem deixam de fornecer serviços (Apêndice A, itens 50 e 51). Por exemplo, a nuvem HP Helion, utilizada em experimentos dessa tese (Seção 7.1), anunciou a oferta de serviços de armazenamento em 7 de setembro de 2011 (Apêndice A, item 52). Todavia, encerrou a atuação nessa área em 31 de Janeiro de 2016.

Provedores de nuvem podem ser públicos ou privados. Provedores públicos oferecem recursos como elasticidade de recursos e esquemas de pagamento flexíveis (pague somente pelo que usar). Provedores privados possuem infraestrutura cujo objetivo primário não é fornecer recursos a clientes pela Internet, mas sim atender demandas de usuários específicos, em seus respectivos domínios administrativos (SOTOMAYOR et al., 2009). Pode-se fazer uso de ambos os tipos de provedores simultaneamente, configurando o que se chamada de nuvem híbrida. Existem trabalhos especificamente projetados para funcionar em nuvens híbridas (exemplo: Hybris, Seção 3.1.8), na tentativa de reunir vantagens de cada tipo de nuvem.

2.2.6 Ferramentas de comunicação de grupo

Serviços de coordenação de estados podem usar ferramentas que oferecem primitivas básicas de comunicação em grupo. Nesta seção são mencionadas duas ferramentas: o *ZooKeeper* (HUNT et al., 2010) e o *etcd* (Apêndice A, item 8). Essas ferramentas armazenam informações sob a referência de chaves (semelhante à chave-valor). As chaves seguem também uma hierarquia semelhante a um sistema de arquivos.

ZooKeeper e *etcd* possuem características comuns (AILIJIANG; CHARAPKO; DEMIRBAS, 2016) e podem ser usados para realizar tarefas como eleição de líder e gerência de configuração, entre outras. Ambas ferramentas oferecem a semântica livre de espera (Seção 2.2.2). Além disso, fornecem *interfaces* de armazenamento de dados com mecanismos de notificação de alterações (MARTIN; ALVISI; DAHLIN, 2002). Isso significa que os clientes podem observar valores e serem

informados sobre alterações (padrão *listener*, Seção 2.2.2).

Os dados armazenados pelo ZooKeeper e pelo *etcd* podem ser replicados a fim de aumentar a resiliência. Para tal fim, o ZooKeeper usa o protocolo Zab (REED; JUNQUEIRA, 2008), enquanto o *etcd* utiliza o Raft (Seção 3.2.3). Os protocolos Zab e Raft fazem uso de um líder (Seção 2.1.4) para sequenciar pedidos às demais réplicas. O Zab original foi estendido pelo ZooKeeper e pode atribuir pesos diferenciados a réplicas (GIFFORD, 1979). Dessa maneira, algumas réplicas podem se tornar mais representativas (comportamento semelhante a *super nodes*), enquanto outras réplicas podem ter sua participação em quóruns descartada. O *etcd* é uma aplicação *stateless*, no sentido de que não mantém informações sobre clientes que o acessam. Assim, oferece uma *interface* de acesso REST para a utilização de seus serviços.

2.3 VIRTUALIZAÇÃO EM *CONTAINERS*

O conceito de virtualização surgiu da necessidade de simular a execução de instruções computacionais (GOLDBERG, 1974). Virtualização pode ser utilizada, por exemplo, para executar um *software* projetado para um *hardware* antes que o mesmo esteja disponível. As máquinas simuladas, denominadas máquinas virtuais (MV), têm sido utilizadas em produção e podem executar programas com velocidade similar à execução no ambiente físico que hospeda as MV (XAVIER et al., 2013; FELTER et al., 2015).

Uma arquitetura baseada em *containers* possui um sistema operacional compartilhado que conta com um sistema de arquivos, um conjunto de bibliotecas e aplicações (SOLTESZ et al., 2007). Os *containers* são MV que, nessa arquitetura, podem ser instanciadas, desligadas ou reiniciadas do mesmo modo como ocorre em arquiteturas tradicionais. Recursos como espaço em disco, memória e processamento são associados a uma MV no momento em que a mesma é criada.

As imagens dos *containers* são pequenas, em comparação a imagens de MV tradicionais, pois apenas os arquivos que não existem no *host* são armazenados na imagem. Isso é possível devido à utilização de um sistema de arquivos em camadas (por exemplo, o sistema *AuFS* (MERKEL, 2014)). Assim, a criação de *containers* é mais rápida e o provisionamento de recursos torna-se mais eficiente, quando comparados às tradicionais MVs (XAVIER et al., 2013; FELTER et al., 2015).

Os *containers* são instanciados a partir de imagens estáticas.

Quando um *container* é desligado, seu estado[7] é perdido. Em geral, *containers* não mantém dados de sessão sobre clientes, de tal maneira que são conhecidos como máquinas virtuais *stateless*.

Os recursos que permitem a virtualização a nível de sistema, no *kernel* do Linux, são conhecidos como LXC (Apêndice A, item 26). **Docker** é um *software* que estendeu o LXC e tornou-se uma popular implementação de *containers* (BERNSTEIN, 2014; PEINL; HOLZSCHUHER; PFITZER, 2016). Outras implementações apresentam diferentes características, a exemplo da empresa CoreOS, que desenvolve o *container* chamado *rkt* (Apêndice A, item 6), cujas premissas são fornecer uma implementação de *container* mais independente e segura.

Docker e *rkt* são implementações de *containers*, mas um *cluster* precisa de ferramentas que ofereçam outros recursos de gerenciamento. Monitoramento, composição de serviços e rede de *containers* entre máquinas físicas são algumas dessas necessidades. O Docker oferece diversas ferramentas que podem implementar essas necessidades (*Docker swarm* para composição, *Docker machine* para hospedar *containers*, etc). Entretanto, essas ferramentas são restritas a *containers* Docker.

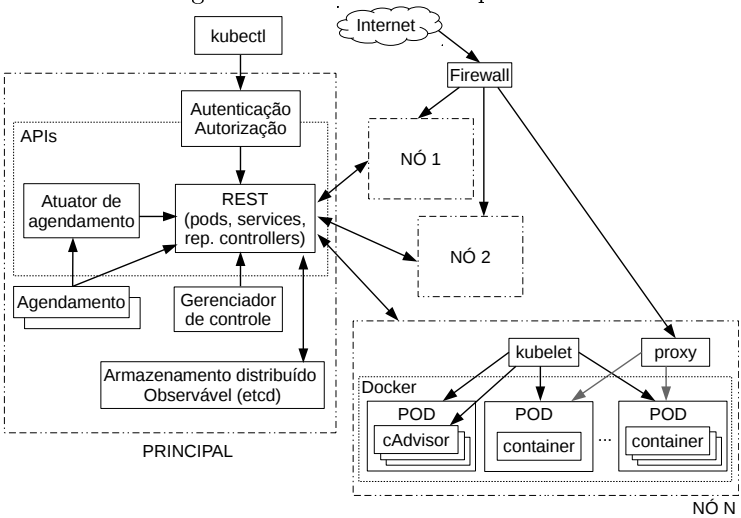
Há dois principais gerenciadores que integram soluções para *containers*: Apache Mesos e **Kubernetes**. O Mesos é um gerenciador de *cluster* que cria abstrações de recursos em máquinas físicas. *Software* de terceiros realizam tarefas como o agendamento da criação de *containers* Docker (Marathon, Apêndice A, item 7) e eleição de líder (em caso de replicação da máquina principal) do *cluster* (ZooKeeper, Seção 2.2.6).

Esta tese utiliza o **Kubernetes** como ambiente de gerenciamento de *containers*. A empresa Google criou o Kubernetes (k8s) para ser um sistema de gerenciamento de *containers* completo (BERNSTEIN, 2014). O Kubernetes herda alguns conceitos do Borg, que é o sistema de gerenciamento de *containers* da Google (VERMA et al., 2015). Por exemplo, considere um servidor web que produz logs, e um analisador de log que os lê. Estas aplicações podem ser criadas em *containers* diferentes, o que favorece o isolamento das aplicações. Contudo, *containers* altamente acoplados devem ficar na mesma máquina, a fim de reduzir comunicação em rede. O Borg possui uma funcionalidade denominada *Alloc*, que mantém *containers* na mesma máquina. No Kubernetes, o componente *Pod* possui a mesma função do *Alloc*.

Um *cluster* Kubernetes é composto de máquinas, virtuais ou físicas (Figura 4). Cada máquina é um nó. Um *Pod* é a menor unidade de gerenciamento, e pode conter um ou mais *containers*. O *Pod* foi criado

[7] Estado pode ser definido como todos os dados armazenados no *container* desde o instante em que o mesmo foi instanciado a partir da imagem.

Figura 4 – Kubernetes: arquitetura.



Fonte: adaptado de Apêndice A, item 60 (2016).

para manter juntas (na mesma máquina) aplicações que possuem alto acoplamento. *Pods* recebem um endereço de rede e são alocados em nós. *Containers* que estão dentro de um mesmo *Pod* dividem recursos, como dispositivos nos quais eles podem escrever e ler informações. Clientes contatam o *cluster* por meio de um firewall, que distribui os pedidos aos nós de acordo com as regras de balanceamento de carga.

Pedidos enviados por clientes são recebidos por nós. O *proxy* recebe os pedidos do *firewall* e encaminha-os aos *Pods*. Cada nó possui um *proxy* instalado. Se um *Pod* está replicado, o *proxy* distribui a carga, enviando o pedido a uma das réplicas. O *kubelet* gerencia os *Pods*, imagens, *containers* e outros elementos do nó e, também, envia dados sobre o monitoramento dos *containers* ao nó principal.

O nó principal do Kubernetes contém componentes de gerenciamento. O *kubectl* é uma *interface* de comando na qual o operador humano pode interagir com o *cluster*, situando-se como exemplos de ações possíveis a criação de *Pods* e a verificação do estado do *cluster*. Para acessar o cluster, usuários devem autenticar-se e possuir autorização de acesso. Componentes do Kubernetes interagem entre si via APIs *REST* (*Application Programming Interface* - API). Um componente de armazenamento mantém o estado de configuração do *cluster* e é capaz de enviar notificações a outros componentes quando eventos

acontecem, como a criação ou a alteração de dados. A ferramenta *etcd* (Seção 2.2.6) implementa o componente de armazenamento. O gerenciador de controle atua em nível de *cluster*, monitorando nós, enquanto o componente *Agendamento* realiza, por meio de um *Atuador de agendamento*, a criação e a destruição de *containers*. Nesta tese, optamos por usar o termo *container* em vez de *Pod*. Essa consideração é coerente quando assume-se que cada *Pod* contém apenas um *container*.

Containers podem fazer uso de volumes externos para persistir dados. Um exemplo de volume externo é um diretório no sistema de arquivos do nó que hospeda o *container*. Um exemplo de volume de armazenamento mais robusto é o serviço *Amazon Elastic File System* (Amazon EFS) (Apêndice A, item 56). É possível, portanto, persistir dados de *containers* no Amazon EFS (Apêndice A, item 57).

2.4 CONCLUSÕES DO CAPÍTULO

Este capítulo fez uma breve descrição de conceitos relacionados às três grandes áreas que integram esta tese: sistemas distribuídos, armazenamento de dados e virtualização em nível de sistema. Destaca-se o conceito de replicação, elemento básico para prover redundância necessária à tolerância a faltas, e consenso, presente na coordenação de estados de réplicas que devem manter as cópias de dados sincronizadas. Foram descritos aspectos de consistência de dados, essenciais no ambiente geograficamente distribuído, bem como as principais características de provedores de armazenamento em nuvem, utilizados na avaliação deste trabalho (Capítulo 7). Por fim, a caracterização de *containers* descreveu tecnologias usadas na construção de soluções para os problemas elencados nesta tese. O próximo capítulo contém alguns dos principais trabalhos da literatura que tratam sobre os problemas relacionados à contribuição desta tese.

3 REVISÃO DA LITERATURA

Se eu vi mais longe, foi por estar de pé sobre ombros de gigantes.
Isaac Newton

Este capítulo apresenta uma revisão da literatura pertinente a esta tese, descrevendo alguns dos principais trabalhos relacionados a armazenamento de dados geo-distribuído, coordenação de estados replicados e arquiteturas de armazenamento. Ao final do capítulo, são identificadas oportunidades de pesquisa, levando em conta a literatura e o contexto desta tese.

3.1 GEO-REPLICAÇÃO DE DADOS

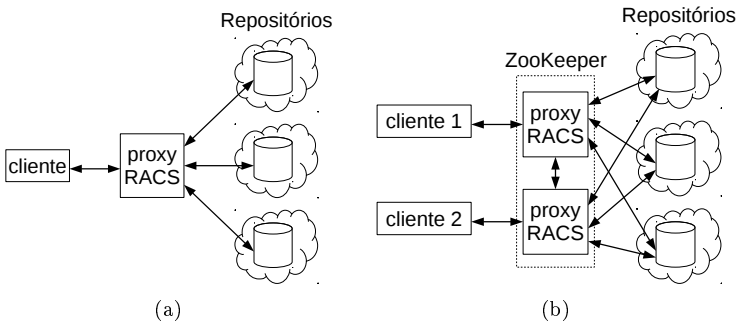
Nesta seção são apresentados alguns dos principais estudos relacionados a armazenamento geo-distribuído de dados em provedores de nuvens. São descritos (em ordem temporal de publicação) o RACS, ICStore, DepSky, Vivace, Gnothi, SPANStore, MDStore e Hybris. Ao término da seção são resumidas as principais características desses trabalhos e apontadas direções de pesquisa nas quais esta tese prossegue.

3.1.1 RACS

RACS *(Redundant Array of Cloud Storage)* é um sistema de armazenamento chave-valor que utiliza múltiplos provedores de nuvens (ABU-LIBDEH; PRINCEHOUSE; WEATHERSPOON, 2010). Seu principal objetivo é reduzir custos em cenários nos quais é necessária a migração de provedor. A técnica RAID (Seção 2.2.4) é utilizada, em nível de provedores de nuvem, para reduzir o tamanho dos dados replicados. O cliente interage com um *proxy* que contacta provedores para realizar o armazenamento (Figura 5(a)). A *interface* de acesso aos dados é idêntica à *interface* do Amazon S3, de tal forma que um cliente do Amazon S3 pode utilizar o RACS de forma transparente (via *proxy*). As operações suportadas são *put*, *create*, *delete*, *get* e *list*.

Pode-se usar mais de um *proxy* (Figura 5(b)), coordenando-os via ZooKeeper. Os *proxies* compartilham informações de autenticação do usuário, localização e credenciais de acesso para cada repositório. RACS utiliza a semântica SWMR.

Figura 5 – RACS: arquitetura com (a) um *proxy* (b) vários *proxies*.



Fonte: adaptado de (ABU-LIBDEH; PRINCEHOUSE; WEATHERSPOON, 2010).

Ao receber um dado para ser armazenado, o *proxy* cria os fragmentos usando *erasure codes* (Seção 2.2.4.2). O dado é particionado, originando k *data shares* (cada *data share* possui o tamanho $1/k$ do dado original). A seguir, são criados m *shares* redundantes, e qualquer conjunto de k *shares* é suficiente para reconstruir o dado. Os metadados contém informações como nomes dos *buckets*, data de modificação, tamanho original do dado e *hash* do dado. Os metadados são replicados em todos os servidores.

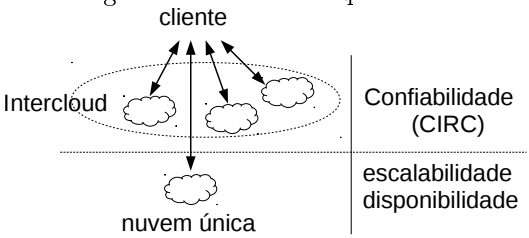
O objetivo do RACS é fornecer uma solução que reduza o custo do armazenamento de dados em provedores de nuvens. Menciona-se tolerância ao tipo de falta "econômica", no sentido de permitir a livre migração de dados entre provedores. Provedores em nuvem eventualmente alteram preços cobrados por seus serviços (Seção 2.2.5). RACS apresenta benefícios em cenário de migração de dados entre diferentes provedores, porém a recuperação de *shares* perdidos é custosa, e essa avaliação não é realizada (há uma breve discussão sobre replicar o dado integralmente ou fragmentado, na Seção 7.2.6).

3.1.2 ICStore

ICStore (*InterCloud Store*) é um serviço de armazenamento chave-valor composto por serviços disponíveis em nuvens comerciais (CACHIN; HAAS; VUKOLIC, 2010). Seus principais objetivos são prover confidencialidade, integridade, confiabilidade e consistência (*Confiden-*

tiality, Integrity, Reliability and Consistency - CIRC) ao dado. O conceito de *Intercloud* (nuvem de nuvens) é apresentado como uma camada superior às nuvens individuais (Figura 6). O uso de um provedor em nuvem fornece maior escalabilidade e disponibilidade à aplicação. No nível Intercloud, o cliente orquestra o armazenamento em múltiplos provedores de nuvem, e os objetivos nessa camada são prover o CIRC. Clientes usam uma biblioteca para acessar as múltiplas nuvens. ICStore manipula dados com as *interfaces put*, *get* e *delete*.

Figura 6 – ICStore: arquitetura.



Fonte: adaptado de (CACHIN; HAAS; VUKOLIC, 2010).

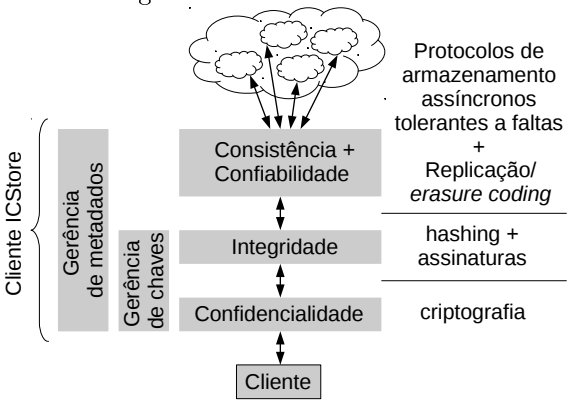
O ICStore possui três camadas principais (Figura 7) para gerenciar o CIRC[1]. Cada camada pode ser ativada ou desativada individualmente, para prover a funcionalidade desejada. A camada de integridade fornece aos dados proteção criptográfica contra modificações indevidas. Para um leitor e escritor simples (proprietário do dado), o uso de um *hash* é suficiente. No entanto, quando há múltiplos leitores e escritores, chaves públicas e privadas (criptografia assimétrica) são utilizadas para garantir a integridade do *hash* (Seção 2.2.3). A camada de consistência e confiabilidade distribui o dado em múltiplos provedores de nuvem, com o uso de protocolos de armazenamento tolerantes a faltas (por exemplo, RACS (Seção 3.1.1)). O dado pode ser replicado integralmente ou com o uso de *erasure codes* (Seção 2.2.4).

O ICStore fornece consistência regular (Seção 2.2.2), considerada suficiente para aplicações com baixa contenção. Em cenários de alta contenção, admite-se a coexistência de dois valores sob o mesmo número de versão (consistência *fork*, Seção 2.2.2).

O ICStore apresenta-se como solução em camadas, o que flexibiliza seu funcionamento, podendo incluir outras soluções em cada camada. Um protótipo é mencionado, mas não há implementação dis-

[1] Confidencialidade não é comentada, pois não está no escopo desta tese (Seção 1.2).

Figura 7 – ICStore: camadas.



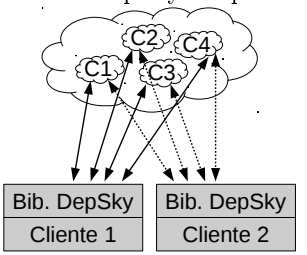
Fonte: adaptado de (CACHIN; HAAS; VUKOLIC, 2010).

ponível. Entretanto, em dezembro de 2013, a IBM manifestou intenção em utilizar o ICStore na prática (Apêndice A, item 13).

3.1.3 DepSky

O DepSky é um sistema de armazenamento chave-valor que utiliza diversos provedores em nuvem (Figura 8) para melhorar a confiabilidade do dado, quando comparado ao armazenamento em uma única nuvem (BESSANI et al., 2013). As principais características da nuvem virtual de armazenamento (*Intercloud*) são alta disponibilidade, integridade, privacidade e independência de provedor.

Figura 8 – DepSky: arquitetura.



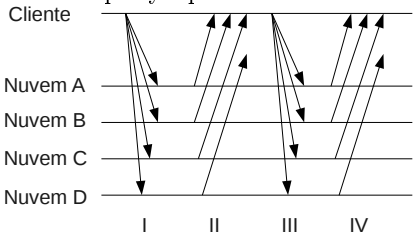
Fonte: adaptado de (BESSANI et al., 2013).

DepSky apresenta custos variáveis entre 1.2 a 2 vezes o custo de usar uma única nuvem. As operações realizadas sobre os dados são *put, create, delete, get* e *list*. Dois protocolos são responsáveis por realizar o armazenamento: o DepSky-A e o DepSkyCA. O DepSky-A realiza a replicação integral do dado, enquanto o DepSky-CA aplica criptografia e *erasure codes* aos dados. O código do DepSky está disponível sob forma de biblioteca para ser utilizada junto ao cliente, em Java.

Os dados são armazenados e recuperados dos provedores com o uso de quóruns bizantinos (Seção 2.1.5) e a leitura e escrita são realizadas com quóruns de duas fases (Figura 9). Para escrever um dado no DepSky-A, inicialmente o cliente envia o dado para todos os provedores (etapa I), considerando que a quantidade de provedores (n) é definida por $n = 3f + 1$ (Seção 2.1.3). Após a confirmação de $2f + 1$ respostas dos provedores (etapa II), relativas à gravação do dado, o cliente solicita aos provedores que armazenem os metadados (etapa III) correspondentes ao dado que foi armazenado. Após $2f + 1$ confirmações de gravação do metadado pelos provedores (etapa IV), a operação de escrita é concluída. Criptografia assimétrica é utilizada nos metadados, de maneira que clientes escritores compartilham uma chave privada, e clientes leitores têm acesso à respectiva chave pública.

Para ler os dados, o cliente do DepSky-A solicita a todos os provedores os metadados relativos ao dado desejado. Ao obter respostas de $2f + 1$ provedores, pode solicitar o dado a qualquer provedor. Nos protocolos do DepSky, a escrita de dados é feita antes da escrita de metadados para garantir que os metadados estarão disponíveis para consulta somente após a confirmação da gravação dos dados.

Figura 9 – DepSky: quórum com dois *rounds*.



Fonte: adaptado de (BESSANI et al., 2013).

O protocolo DepSky-CA aplica *erasure code* (Seção 2.2.4.2) para melhorar a eficiência do armazenamento, reduzindo os custos de replicação. Especificamente, utiliza-se a biblioteca Jerasure (PLANK; SIMMERMAN; SCHUMAN, 2008). Além disso, garante a confidenci-

alidade do dado por meio de cifragem, utilizando uma chave simétrica gerada aleatoriamente. Esta chave é distribuída entre os servidores, utilizando-se a técnica de compartilhamento secreto (SHAMIR, 1979). A fragmentação e o compartilhamento secreto são combinados seguindo uma estratégia já estabelecida (KRAWCZYK, 1993), de forma que *blocos* são compostos por um fragmento do dado e um compartilhamento secreto da chave, e os blocos são distribuídos entre os provedores.

A leitura de dados no DepSky-CA é realizada consultando-se metadados de forma semelhante ao DepSky-A. A seguir, por meio de um quórum de leitura de dados, obtém-se blocos de k provedores, tal que k é o número mínimo de fragmentos necessários para reconstruir o dado e a chave usada na cifragem. Na prática, utiliza-se $k = f + 1$.

A leitura de dados no DepSky pode ser aprimorada com acesso aos provedores de forma ordenada e sequencial, sendo feita de forma unitária para o DepSky-A e simultaneamente em $f + 1$ provedores, para o DepSky-CA. No caso de um tempo máximo ser excedido, mais provedores seriam contactados para a obtenção do dado.

O DepSky oferece a semântica de um escritor e múltiplos leitores (SWMR). Há um controle de concorrência que considera baixa contenção, permitindo assim a semântica MWMR. Porém, esse controle contém limitações, apresentadas pelos próprios autores desse sistema. Quando dois escritores tentam simultaneamente fazer os bloqueios nos provedores, é possível que nenhum deles tenha sucesso, e ainda, que a liberação de um bloqueio possa ser *atrasada* por um provedor malicioso. Contudo, essa característica não impede o bloqueio infinitamente, pois todo bloqueio carrega consigo uma marca de tempo (*timestamp*). Dessa maneira, todo bloqueio acaba por expirar em determinado momento.

Outro aspecto limitante nesse controle de concorrência é o fato de que um arquivo de bloqueio pode não surgir logo após seu comando de criação, visto que a consistência de gravação de dados em provedores de nuvem é, de modo geral, eventual (Seção 2.2.5). O algoritmo de bloqueio pode ser alterado para aguardar antes de verificar se o bloqueio foi efetivado, mas a obtenção desse tempo de espera envolve dificuldades similares à definição de temporizadores (Seção 2.1.2). A inclusão de identificadores únicos de clientes escritores (Seção 2.2.2) também é mencionada como medida para aumentar a segurança (*safety*) do protocolo de bloqueio. No caso, é sugerido o *hash* de parte da chave privada do cliente. Porém, se os clientes compartilharem uma chave única para escrita, essa estratégia fica comprometida. Essas sugestões objetivam garantir a segurança (*safety*) e manter os nomes de arquivos de dados diferentes, quando criados por escritores concorrentes.

3.1.4 Vivace

Vivace é um sistema de armazenamento chave-valor que provê consistência forte e geo-replicação (CHO; AGUILERA, 2012). Seu objetivo é manter o desempenho estável durante congestionamentos de rede, tolerando faltas de parada. Para isso, em situações de congestionamento os metadados são reduzidos e priorizados no tráfego. No Vivace, a técnica da redução dos metadados é aplicada em dois algoritmos: o primeiro implementa leitura e escrita (registradores) e o segundo implementa replicação de máquinas de estado. Nesta tese, será detalhado apenas o algoritmo que realiza leitura e escrita, por apresentar as características de interesse do estudo desenvolvido.

A implementação do Vivace sobre registradores é baseada no algoritmo ABD (ATTIYA; BAR-NOY; DOLEV, 1995), que funciona usando quóruns. Para escrever um valor no ABD, primeiramente o cliente obtém um *timestamp*[2]. A seguir, solicita às replicas que armazenem o valor e aguarda por uma maioria de respostas. Desse modo, a leitura do dado inicia com o cliente enviando um pedido de leitura para todas as réplicas, e estas informam o valor e o *timestamp* que possuem. Assim que houver resposta da maioria, o cliente seleciona o par (valor, *timestamp*) com o *timestamp* mais recente. Se perceber alguma réplica com *timestamp* desatualizado, o cliente executa uma fase de *write-back*.

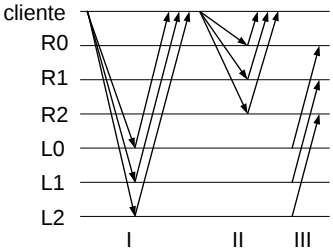
Em cenários de congestionamento, as mensagens são divididas em duas partes: campos críticos e outros campos[3], sendo necessários novos *rounds* de comunicação para operacionalizar as mensagens divididas. Outra modificação é a criação de réplicas locais. Para cada réplica remota existe uma réplica local, que desempenha uma espécie de *cache* parcial da réplica remota.

Quando há congestionamento de rede, uma escrita no Vivace (Figura 10) procede conforme segue. Inicialmente, o cliente obtém um *timestamp* das réplicas e envia os dados completos (campos críticos e outros) às réplicas locais (L0...L2, fase I). Após a confirmação da escrita local, o cliente envia às réplicas remotas (R0...R2) apenas os dados críticos (atualização de metadados). Essas mensagens têm prioridade na rede, e, após obter as respostas das réplicas remotas, a escrita é considerada concluída. Em segundo plano (fase III), as réplicas locais enviam os dados às réplicas remotas, em mensagens não priorizadas.

[2]Os processos possuem relógios sincronizados que são usados como contadores. Estes relógios podem ser implementados com GPS, rádio ou protocolos como NTP.

[3]Na prática, configura-se aqui uma separação entre dados e metadados, sendo que os metadados contém os campos críticos.

Figura 10 – Vivace: operação de escrita em registrador.



Fonte: próprio autor (2017).

A leitura de um registrador envolve apenas as réplicas remotas e requer três fases de operações com quóruns. Primeiro, o cliente solicita às réplicas remotas o *timestamp* (metadado) mais atual (fase I), sendo que essas mensagens são pequenas e priorizadas na rede. Após obter respostas de uma maioria de réplicas, o cliente seleciona o *timestamp* mais recente (o maior deles) e então solicita às réplicas locais o dado com o *timestamp* selecionado (fase II). Quanto à obtenção do dado, é suficiente a resposta de apenas uma das réplicas (a mais rápida). Por fim, se o cliente perceber *timestamps* desatualizados, ele atualiza as réplicas com a escrita do *timestamp* mais atual (fase III, *write-back*).

Para melhorar a eficiência da operação de leitura de dados no Vivace, após obter o *timestamp* mais recente (fase I da leitura) pode-se obter os dados apenas da réplica mais próxima. Para isso, o cliente deve manter um histórico de latências dos provedores.

No Vivace existe uma janela de vulnerabilidade devido à replicação assíncrona das réplicas locais para as réplicas remotas. A escrita é confirmada para o cliente após a gravação dos dados em réplicas locais. Contudo, se houver falha nas réplicas locais (por exemplo, um desastre no *data center* local), o dado não terá sido enviado às réplicas remotas, o que ocasionará perda dos dados.

3.1.5 Gnothi

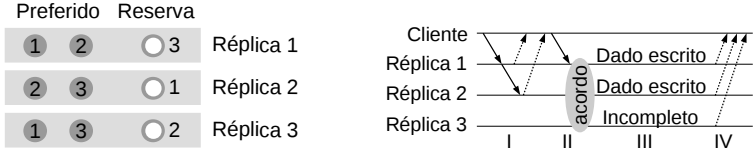
Gnothi é um sistema de armazenamento em blocos (WANG; ALVISI; DAHLIN, 2012). A *interface* é similar a um disco rígido: há um número fixo de blocos de mesmo tamanho, e aplicações podem ler e gravar blocos. O Gnothi provê tolerância a faltas de parada, com semântica linearizável. Metadados são armazenados em $2f + 1$ réplicas

(via RME, Seção 2.1.4) e dados são replicados em $f + 1$ réplicas.

Nesse sistema, é necessária a descrição das características do bloco, para a compreensão do funcionamento do protocolo. Um bloco é *completo* se sua versão corresponde à versão que está registrada no metadado. É *incompleto* se o metadado contém uma versão mais recente que o dado gravado na réplica. Blocos podem ser também *fresh* ou *stale*. Um bloco *fresh* contém o dado da última atualização realizada, enquanto um bloco *stale* contém uma versão anterior à última atualização, havendo possibilidade de existir, por exemplo, um bloco completo e *stale*. Uma réplica marca cada bloco de dados que possui como completo ou incompleto e isso garante a leitura correta da última versão de dados, pois quando a réplica está incompleta, o cliente é informado e pode solicitar o dado à outra réplica. O tamanho do bloco é configurável (4KB a 1MB foram avaliados).

Os dados são gravados em um conjunto de réplicas denominado réplicas preferidas e os metadados são gravados em todas as réplicas. Considere-se um exemplo no qual o dado 1 é gravado nas réplicas 1 e 3, o dado 2 é gravado nas réplicas 1 e 2, e o dado 3 é gravado nas réplicas 2 e 3 (Figura 11(a)). Esses dados são gravados juntamente com os metadados. Por outro lado, ocorre também uma gravação exclusiva de metadados: o metadado do dado 1 é gravado na réplica 2, o metadado do dado 2 está na réplica 3 e o metadado do dado 3 está na réplica 1. O nó (réplica) 1 é um nó de armazenamento reserva para o dado 3 e os nós 2 e 3 são nós de reserva para os dados 1 e 2, respectivamente. Assim, quando faltas ocorrem (ou temporizadores expiram), o bloco de dados pode ser gravado em uma réplica reserva.

Figura 11 – Gnothi: exemplos de replicação.



Fonte: adaptado de (WANG; ALVISI; DAHLIN, 2012).

Para escrever um dado (Figura 11(b)), o cliente armazena os dados nas $f + 1$ réplicas preferenciais (fase I, chamada *PrepareData*). O dado possui um identificador único composto pela tupla (*clientID, clientSeqNo*). O primeiro campo é um identificador do cliente, e o segundo

é um número sequencial controlado pelo cliente. Por meio de replicação em cadeia (Seção 2.1.1) propaga-se o dado nas réplicas e o cliente aguarda a confirmação do armazenamento do dado por $f + 1$ réplicas. Se um temporizador expirar, réplicas adicionais são contactadas.

Após gravar os dados, o cliente envia metadados para todas as réplicas (fase II, denominada *WriteData*) e um protocolo de acordo[4] é aplicado para ordenar a escrita. Ao receber os metadados (fase III), três situações podem ocorrer:

- A réplica contém a mesma versão de dado e metadado, e a réplica é preferida. Nesse caso, o bloco torna-se completo na réplica (representado por "Dado escrito" na Figura 11(b)).
- A réplica contém dados de versão anterior aos metadados recebidos. Nesse caso, o bloco é marcado como incompleto na réplica. Réplicas incompletas afetam a leitura de dados, conforme descrito mais adiante.
- A réplica possui dados e metadados de mesma versão, mas não é preferida (nesse caso, é uma réplica *reservada*, na nomenclatura adotada). Nesse caso, o dado é gravado na seção reservada e o bloco torna-se completo na réplica. Destaca-se que esse cenário ocorre quando há réplicas lentas ou indisponíveis.

Por fim (fase IV), o cliente aguarda $f+1$ confirmações de escrita de metadado, e em caso de *timeout*, repete-se a gravação do metadado.

A leitura de dados é realizada usando o protocolo Gaios (BOLOSKY et al., 2011), modificado para operar com réplicas incompletas. Com isso, para ler dados, o cliente envia uma solicitação de leitura de dado ao líder do protocolo de acordo. Em geral, a primeira réplica do quórum preferido é o alvo dessa solicitação. O líder associa o número de versão do dado mais recente ao comando de leitura. Esse número é igual ao número de escritas já realizadas (ou a serem realizadas) sobre aquele dado. A réplica, então, aguarda até que a escrita com o número de versão especificado tenha sido realizada para depois executar a leitura do dado. Dois casos são possíveis nesse processo:

- se o bloco é completo, a réplica envia o dado ao cliente e o processo de leitura está completo.
- se o bloco é incompleto, a réplica envia a mensagem "incompleto" ao cliente. Assim, ele pode solicitar a leitura do dado a outra réplica, em vez de aguardar um temporizador expirar.

O Gnothi realiza instanciação sob demanda, armazenando dados apenas em réplicas preferidas, o que permite reduzir custos de escrita. Utiliza, também, propagação em cadeia para enviar os dados às réplicas,

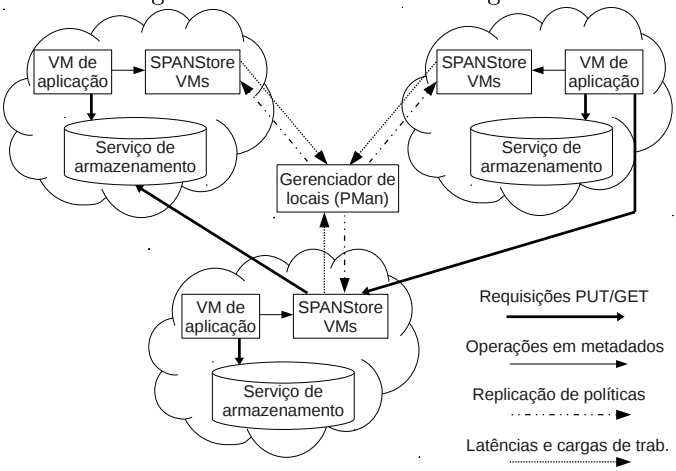
[4] Foi usada uma versão modificada do ZooKeeper (Seção 2.2.6).

a fim de evitar gargalos de rede. No entanto, essa abordagem não é factível em provedores de armazenamento passivos (Seção 2.2.5). O Gnothi mantém informações de metadados diferentes em cada réplica: um bloco pode estar *stale* ou *fresh* em uma réplica. Se a réplica é quem marca o bloco recebido, de acordo com os metadados que possui, ela precisa ser um elemento ativo. Lembra-se que provedores de dados de menor custo são passivos.

3.1.6 SPANStore

SPANStore é um sistema de armazenamento chave-valor que permite armazenar um dado em múltiplos provedores em nuvem (WU et al., 2013). O objetivo principal é reduzir custos explorando as diferenças de preços praticados por diferentes provedores, tolerando faltas de parada. Esse sistema localiza-se no *data center* do provedor de nuvem (Figura 12), considerando que uma máquina virtual (MV) contém o SPANStore e outra MV hospeda a aplicação. Desse modo, um serviço de armazenamento do *data center* está disponível para uso pelo SPANStore.

Figura 12 – SPANStore: visão geral.



Fonte: adaptado de (WU et al., 2013).

A redução de custos do SPANStore é alcançada com uma recomendação para a aplicação sobre quais provedores devem ser utilizados

para realizar o armazenamento, ação para a qual são utilizadas informações mantidas pelo SPANStore, custos dos provedores e informações de necessidades específicas da aplicação. Uma das informações que o SPANStore precisa são os valores praticados pelos provedores de serviço (tabela de preços), que pode ser obtida diretamente dos sites de provedores. Custos dos provedores influenciam a decisão de onde armazenar dados (custo de armazenamento), bem como por onde os dados devem ser enviados (custos de transmissão, ou custos de rede).

Outro grupo de informações mantidas pelo SPANStore é uma tabela interna que contém as latências de comunicação entre os *data centers* e a quantidade de acessos *(workload)* dos objetos armazenados em cada *data center*. Essas informações são coletadas pelo componente *Placement Manager* (PMan), ou Gerenciador de Locais (Figura 12). O PMan obtém latências de comunicação entre o *data center* no qual se encontra e os outros *data centers* com os quais possui comunicação. Além disso, é coletada a frequência de acesso dos objetos na MV por diferentes clientes. Essas informações são atualizadas periodicamente (a cada hora, conforme mencionado pelos autores).

O cliente também fornece informações específicas que influenciam na recomendação gerada pelo SPANStore. Pode-se escolher o nível de consistência desejado: eventual ou forte (Seção 2.2.2), descritas a seguir. A consistência eventual permite que o cliente confirme a gravação quando o dado for gravado em apenas uma réplica, e a gravação do dado em outras réplicas ocorrerá em segundo plano. Ao utilizar consistência forte, o sistema provê linearização, mas requer que o cliente aguarde a confirmação da gravação do dado em todas as réplicas. Para isso, o cliente deve informar o número máximo de faltas (f) a tolerar, e o dado será gravado em $f + 1$ provedores. Além dessas informações, o cliente fornece a latência máxima desejada e uma fração de *requests* que devem possuir latência menor do que a latência máxima especificada.

Nesse sistema, os metadados são replicados em todas as MVs, e mantidos em memória. As informações desses metadados mapeiam quais provedores armazenam quais dados, e o dado é multiversão.

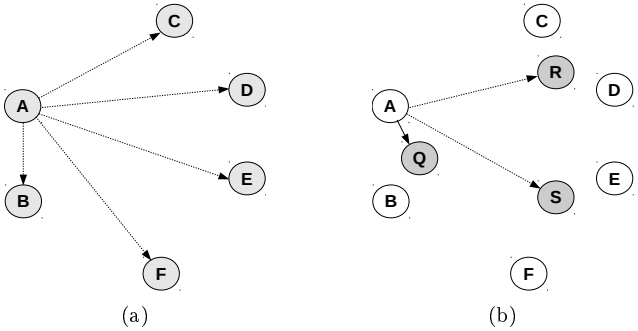
Quando a consistência definida pelo cliente é eventual, a replicação é concluída no momento em que uma das réplicas contactadas confirma para o cliente que o dado foi gravado. Como o dado é replicado em $f + 1$ *data centers*, as outras f réplicas são atualizadas assincronamente. No contexto de tolerância a faltas, o cliente não tem garantia de que as outras f réplicas terão sucesso na operação.

Quando consistência eventual é utilizada, é possível replicar os dados de várias maneiras. Como exemplo, considere-se seis *data centers*

nomeados como A...F. Um cliente está localizado no *data center* A e será realizada uma escrita de dados. Em um primeiro caso (Figura 13(a)), a confirmação da escrita é feita apenas em A, e a atualização do dado é enviada de A para todos os outros *data centers* em segundo plano. Este caso garante menor latência, pois o *data center* mais próximo é atualizado e a resposta é retornada ao cliente. No que tange a atualizações, estas são propagadas aos outros *data centers* em segundo plano.

Em outro cenário (Figura 13(b)), a replicação é parcial (Seção 2.1.1): apenas os *data centers* menos custosos armazenam o dado. Considere os *data centers* Q, R e S que possuem custores menores que os demais. O *data center* mais próximo ao cliente não foi escolhido devido a métricas de custo, confirmando que, num caso destes, é mais barato armazenar em Q do que em A. Como a latência de Q está no limite definido pela aplicação, o dado é armazenado em Q e não em A.

Figura 13 – SPANStore: replicação (a) integral e (b) parcial.



Fonte: adaptado de (WU et al., 2013).

Quando o cliente especifica consistência forte, SPANStore usa um protocolo de efetivação de duas fases (*Two-Phase Commit* - 2PC) (COULOURIS; DOLLIMORE; KINDBERG, 2012, p.732) para executar a escrita de dados. Primeiro, tenta-se obter um *lock* em todos os provedores determinados. Em caso de falha, é verificado se foi possível bloquear ao menos $f + 1$ provedores. Em caso positivo, as escritas são realizadas. Caso contrário, não há possibilidade de realizar a escrita. De qualquer maneira, todos os bloqueios são liberados após a operação.

Para realizar a leitura, inicialmente busca-se o dado em todos os provedores relacionados ao objeto. Em caso de falha, verifica-se se existe interseção entre os provedores que retornaram o dado e os provedores nos quais a escrita foi efetivada. Se houver interseção, o

dado (de versão mais atual) é retornado, caso contrário, não é possível realizar a leitura. Os conjuntos de réplicas preferidas (leitura ou escrita) podem ser alterados quando o padrão de acesso a um objeto é alterado.

Podem ocorrer alterações nos valores praticados pelos provedores (Apêndice A, items 1 a 5), o que pode reconfigurar o conjunto de provedores menos custosos, sob o ponto de vista do cliente. Existem ferramentas que permitem estimar custos em provedores de nuvens (Apêndice A, item 27). Entretanto, o diferencial do SPANStore é a consideração de itens dinâmicos como latências momentâneas e a disponibilidade de provedores no momento em que a recomendação for elaborada para a aplicação. Contudo, o elemento PMan é um ponto fraco do sistema por se tratar de um elemento único (não-replicado). Porém, ele não afeta significativamente o desempenho sistema, por ser atualizado esporadicamente. Além disso, a parada de funcionamento desse componente apenas resultaria na desatualização das tabelas de latências e acesso a objetos. Observa-se que a janela de recuperação do mesmo (hora) é razoavelmente grande. Por fim, é notável mencionar que o SPANStore não garante a integridade do dado.

3.1.7 MDStore

O protocolo MDStore oferece um serviço de armazenamento chave-valor que tolera faltas bizantinas (CACHIN; DOBRE; VUKOLIĆ, 2014). Dados e metadados são separados de tal forma a permitir que $3f + 1$ réplicas armazenem os metadados, enquanto apenas $2f + 1$ réplicas armazenam os dados. Essa redução no número de réplicas que armazenam dados é um diferencial do MDStore, sendo que ele tem suporte a múltiplos escritores (MWMR), provê semântica livre de espera e oferece consistência atômica (Seção 2.2.2).

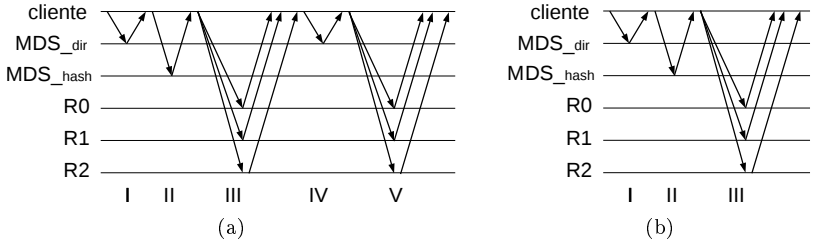
MDStore define um mecanismo denominado *timestamped storage*, que consiste em armazenar um *timestamp* TS (inteiro, iniciando em zero) junto do valor x (iniciado em $\perp$). A escrita é definida por TSWrite(ts, v), tal que apenas se ts $\geq TS$, ts é salvo em TS e v é salvo em x. A leitura é realizada por TSRead() e retorna TS e x.

Réplicas de metadados do MDStore contém dois componentes: um registrador de *timestamps*, denominado MDS_{dir}, e um vetor de registradores de *hash*, denominado MDS_{hash}. O MDS_{dir} é um registrador MWMR que armazena o último *timestamp* e a identificação das réplicas de dados que contêm o dado. Cada registrador do MDS_{hash} é SWMR e contém o *hash* do dado, cuja versão é definida pelo *timestamp*,

também armazenado no registrador. O objetivo do *hash* é permitir a verificação da integridade do dado. *Timestamps* são compostos por um número inteiro e um identificador do cliente (cid). Se dois *timestamps* apresentam valores iguais ($ts_1.num = ts_2.num$), aquele que possuir o maior identificador de cliente ($ts.cid$) será considerado o maior *timestamp* (configurando assim um *timestamp* multi-escritor, Seção 2.2.2).

Para escrever um dado no MDStore (Figura 14(a)), o cliente primeiro obtém o *timestamp* mais atual (fase I)[5]. A seguir, incrementa o *timestamp* e grava o novo *timestamp* juntamente com o *hash* do dado (fase II). O cliente então envia o dado, juntamente com o *timestamp*, às réplicas de dados (fase III) e a gravação de dados termina quando pelo menos $f + 1$ réplicas confirmam ao cliente. O próximo passo (fase IV) é atualizar os metadados com o novo *timestamp* e a localização das réplicas de dados que confirmaram a escrita do dado. Por fim (fase V), as réplicas de dados são notificadas (mensagem *commit*) de que podem remover a antiga versão do dado.

Figura 14 – MDStore: (a) escrita e (b) leitura de dados.



Fonte: adaptado de (CACHIN; DOBRE; VUKOLIĆ, 2014).

Para ler os dados (Figura 14(b)), o cliente inicia obtendo o *timestamp* mais recente (fase I) e, depois, obtém o *hash* do dado (fase II). Por fim, solicita às réplicas de dados a leitura do dado (fase III). Entre a leitura do *timestamp* mais recente e a obtenção do dado é possível que outro cliente escritor tenha atualizado o dado, sendo necessário, portanto, realizar novamente a operação de leitura.

MDStore é livre de espera em operações de escrita (Seção 2.2.2). Dessa forma, o consenso é realizado em apenas um passo. O uso de um *timestamp* multi-escritor impede contenção direta entre clientes durante a escrita. Entretanto, enquanto houver escritores atuando, a operação de leitura poderá encontrar dados novos (mais atualizados

[5]Os identificadores MDS representam as réplicas de metadados na Figura 14(a).

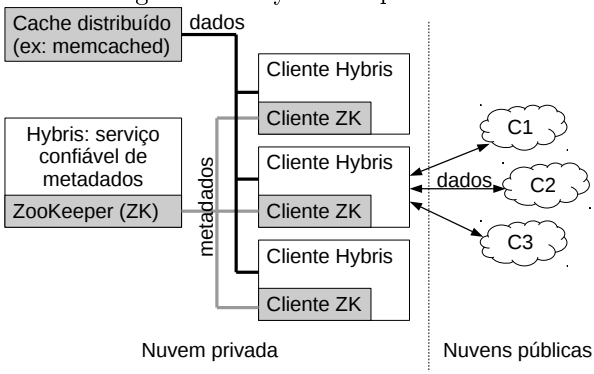
que os metadados buscados inicialmente). O MDStore provê liberdade de espera para a leitura, sob o risco, portanto, de ocasionar ciclos adicionais de leitura de metadados.

3.1.8 Hybris

Hybris é um sistema de armazenamento chave-valor que utiliza nuvens híbridas (DOBRE; VIOTTI; VUKOLIĆ, 2014). Metadados são armazenados em provedores de nuvens privadas ativas confiáveis. Dados são distribuídos em provedores de nuvens públicas passivas, sujeitas a ocorrência de eventos (ex: corrupção de dados, acesso indevido) que podem causar faltas Bizantinas. Hybris apresenta uma redução no número de provedores que armazenam o dado: apenas $f + 1$ provedores, no caso normal. Em caso de faltas, até $2f + 1$ provedores podem ser contactados.

O Hybris (Figura 15) utiliza um serviço de metadados confiável (*Reliable MetaData Service*, RMDS) na nuvem privada. O RMDS tolera faltas de parada. Um cliente Hybris acessa ao RMDS para manipular metadados e interage com múltiplos provedores de nuvem para armazenar os dados, que podem ser armazenados em *cache* local (por exemplo, usando o programa *memcached*) para aumentar desempenho.

Figura 15 – Hybris: arquitetura.



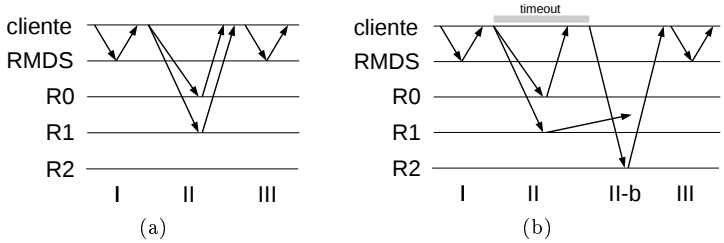
Fonte: adaptado de (DOBRE; VIOTTI; VUKOLIĆ, 2014).

Hybris é MWMR e oferece semântica linearizável (Seção 2.2.2). A semântica de escrita é livre de espera e a semântica de leitura é livre de obstrução. Escrita e leitura de dados são realizadas por meio

das operações *put* e *get*, respectivamente. É possível remover e listar dados, acessando o RMDS via operações *delete* e *list*. Hybris mantém os metadados bastante pequenos (cerca de 50 bytes por chave) para alcançar escalabilidade no provedor privado. Os metadados contém um *timestamp*, ponteiros para os provedores que armazenam os dados, *hash* dos dados e o tamanho do dado. O tamanho do dado é útil no caso de um provedor malicioso enviar dados de tamanho enorme, configurando um ataque DoS. Se forem lidos dados excedentes em relação ao valor especificado pelo tamanho, a obtenção do dado é cancelada. O dado é multiversão (Seção 2.2.2). Um *garbage collector* pode ser acionado para excluir versões antigas de dados.

Para realizar a escrita (Figura 16(a)), o cliente inicialmente obtém do RMDS o *timestamp* mais recente (fase I). Hybris utiliza um *timestamp* multi-escritor (Seção 2.2.2) para prover atomicidade. Considere k a chave definida pelo cliente, no contexto chave-valor. O cliente calcula o novo *timestamp* (ts_{new}) e o dado é escrito em $f + 1$ provedores de nuvens públicas, sob a chave $k \mid ts_{new}$. Um temporizador é disparado pelo cliente. No caso comum, os $f + 1$ provedores de dados confirmam a gravação de dados antes do temporizador expirar.

Figura 16 – Hybris: escrita (a) no cenário comum, e (b) com falta.



Fonte: adaptado de (DOBRE; VIOTTI; VUKOLIĆ, 2014).

Em um caso com falta (Figura 16(b)), até f provedores de dados adicionais podem ser contactados para armazenar o dado (fase II-b). Assim que o cliente possuir $f + 1$ confirmações de armazenamento de dados, o cliente salva os novos metadados no RMDS (fase III). Esses metadados são escritos apenas se o *timestamp* a ser gravado for maior que o último *timestamp* gravado no RMDS. É necessário que o RMDS possua essa capacidade de executar uma atualização condicional.

O cliente inicia a leitura solicitando metadados ao RMDS. A seguir, solicita os dados ao primeiro provedor de nuvens que contém os dados. O recebimento de dados é limitado por um temporizador, de

duração estimada em função do tamanho do dado. Se o temporizador expirar ou se o dado não for validado com o *hash*, os próximos provedores (até f) de dados são contactados. Em um pior caso, durante a leitura podem ocorrer problemas de consistência eventual dos provedores de nuvem (Seção 2.2.5) ou esgotamento prematuro de temporizadores.

Em cenários de escritas concorrentes a uma leitura, é possível que o *garbage collector* remova valores anteriores que o cliente esteja tentando acessar. Nesse cenário, o cliente pode contactar os provedores de dados usando quóruns (Seção 2.1.5), e assim obter a resposta correta de pelo menos um provedor. Para evitar o risco de o cliente usar metadados desatualizados enquanto tenta ler os dados, Hybris utiliza notificadores (Seção 2.2.2) que podem informar ao cliente sobre atualizações em metadados. O cliente poderia realizar novamente a leitura de metadados, ao invés de receber notificações sobre alterações nos metadados, mas essa não é a abordagem praticada pelo Hybris.

Erasure codes (EC) (Seção 2.2.4) podem ser usados para reduzir a quantidade de dados armazenados em cada provedor. Entretanto, o uso de EC aumenta o número de provedores necessários para armazenar o dado[6]. Considere um esquema de EC (k,m), tal que são necessários quaisquer k fragmentos para reconstruir o dado. O dado codificado com EC é armazenado em $f + k$ provedores (em vez de $f + 1$). Os *hashes* de $f + k$ fragmentos também são armazenados junto aos metadados.

No Hybris, os clientes priorizam latência para armazenar dados somente nos $f+1$ provedores mais rápidos. A leitura de dados também é feita iniciando pelo provedor de menor latência. Experimentos no Hybris registraram, assim como no DepSky (Seção 3.1.3), que provedores apresentam latências diferentes de acordo com o tamanho do dado. Em outra palavras, determinados provedores são melhores para operações de dados pequenos, enquanto outros respondem melhor em operações de dados volumosos. Os resultados do Hybris mostram que provedores manifestam diferentes latências dependendo também da operação (escrita ou leitura). São considerados, portanto, esses fatores para a definição da ordem final de acesso aos provedores. Hybris restringe o armazenamento de metadados a uma localização geográfica local.

[6] Uma discussão sobre EC na avaliação desta tese (Seção 7.2.6) menciona esse fato.

3.1.9 Comentários sobre geo-replicação de dados

O uso de temporizadores em sistemas que operam sobre redes não síncronas é comum. Hybris (Seção 3.1.8) utiliza um temporizador para decidir se consegue armazenar dados em apenas $f + 1$ provedores. Em caso de falha, provedores adicionais precisam ser contactados. DepSky e Gnothi (Seção 3.1.3) sugerem o uso de temporizadores para melhorar a escrita e a leitura de dados, atuando apenas em réplicas preferidas. Réplicas adicionais são contactadas após um *timeout*. Temporizadores são usados também para verificar omissão ou parada de líder em protocolos que ordenam requisições com uso de sequenciadores fixos (por exemplo, PBFT, Seção 2.1.4).

São diversos os problemas que surgem ao usar temporizadores em redes não síncronas. Um problema prático consiste no fato de que contratos de utilização de recursos em provedores de nuvens são feitos com base em necessidades e cargas médias, e não em valores de pico, devido a custos (CHO; AGUILERA, 2012). Assim, praticamente pode-se esperar que ocorram incrementos na latência, mediante eventos como particionamento de rede ou sobre-alocação de recursos. Outro problema encontra-se na definição precisa de um temporizador, que requer informações de tempo máximo de processamento, tempo de comunicação em rede e diferença entre relógios dos processos (Seção 2.1.2). É difícil definir essas informações fora do contexto de ambientes síncronos.

No contexto de armazenamento, sistemas que evitam o uso de temporizadores ainda estão sujeitos a desperdício de recursos. O uso de quóruns (Seção 2.1.5), praticado pelo DepSky (Seção 3.1.3) e pelo MDStore (Seção 3.1.7), dispensa temporizadores. Requisições pendentes são canceladas quando o quórum mínimo é obtido, tornando o protocolo livre de espera (Seção 2.2.2). Em escrita de dados, o cancelamento pode ocorrer em diversos momentos, inclusive quando o dado é quase totalmente enviado ao provedor. Esses dados escritos parcialmente no provedor são taxados, apesar de não serem usados pelo protocolo.

A paralelização de tarefas é uma estratégia que pode aumentar a eficiência de sistemas, reduzindo a latência geral. Reduzir latência é o objetivo de vários sistemas que toleram faltas. O Vivace (Seção 3.1.4) busca manter um bom desempenho durante congestionamentos de rede, tornando mensagens de metadados menores e com prioridade de roteamento. O DepSky, o SPANStore e o Hybris mencionam o uso de réplicas preferidas, a fim de obter menor latência nas operações. Uma possibilidade pouco explorada é a execução de tarefas paralelamente ao envio de dados. Uma análise de tráfego em sistemas de armazena-

mento remoto (HU; YANG; MATTHEWS, 2010) sugere que a execução de tarefas de pré-processamento (cifragem, compressão, criação de metadados), que fazem uso intensivo de CPU, podem reduzir a latência geral de operação. Apenas no Vivace (Seção 3.1.4) observou-se uma melhoria de eficiência relacionada à paralelização de tarefas. No caso, a leitura de dados e a escrita de metadados mais recentes (*write-back*, Seção 2.1.5) são realizadas em paralelo. Em nenhum dos outros trabalhos relacionados observou-se a prática desta paralelização.

O uso de latência dos provedores tem ganhado importância na construção de sistemas geo-distribuídos mais eficientes. O SPANStore (Seção 3.1.6) atualiza recomendações fornecidas ao usuário observando latências entre provedores e padrões de acesso aos dados. No DepSky (Seção 3.1.3), os critérios sugeridos para a ordem de acesso a provedores são a latência e o custo do provedor. A latência tende a apresentar relação direta com localização geográfica. Porém, há provedores próximos aos clientes que possuem latências maiores do que provedores mais distantes (COUTO et al., 2014). Assim, é possível usar provedores de menor latência e simultaneamente favorecer a sobrevivência do dado.

Um fato conhecido e que se aplica ao cenário de nuvens híbridas é o uso de provedores de dados locais (próximos ao cliente) para favorecer desempenho. Provedores locais ao cliente de modo geral não são taxados e possuem baixa latência. RACS apresenta, dentre seus trabalhos futuros, o uso integrado de um PC desktop local atuando como servidor, uma nuvem e um *cluster*. Vivace (Seção 3.1.4) usa réplicas locais como *cache* para aumentar desempenho. Para cada réplica remota, há uma réplica local que é atualizada de forma síncrona. Réplicas remotas são atualizadas assincronamente. Hybris (Seção 3.1.8) menciona o uso de réplicas locais para aumentar desempenho.

Uma comparação entre os trabalhos de armazenamento apresentados (Tabela 1) permite observar que, apesar de tolerância a faltas bizantinas ter sido tratada em trabalhos anteriores, a maioria dos trabalhos trata apenas faltas de parada. Nota-se que Gnothi usa temporizadores mas não menciona uso de latências para possíveis ajustes no temporizador. Apenas o Hybris e o SPANStore usam efetivamente as latências dos provedores na operação de seus protocolos (Vivace e DepSky apenas sugerem esse uso como forma de melhorar a eficiência do sistema). Em detalhe, o SPANStore considera o cliente localizado no interior de um *data center*, onde encontram-se também as réplicas. Clientes reais (usuários) acessam o cliente do SPANStore localizado no *data center* mais próximo. Portanto, apenas no Hybris a latência entre um cliente externo (ao *data center*) e o provedor é considerada.

Tabela 1 – Características dos trabalhos de armazenamento em nuvens.

Trabalho	Réplicas de Dados	F	Consist.	EC	*timer*	Latência
RACS	$m + f$	p	eventual	S	N	N
ICStore	$m + f$	p	eventual	opc	N	N
DepSky	$3f + 1$	B	eventual	opc	N	N*
Vivace★	$2f + 1$	p	atômica	N	N	N*
Gnothi	$f + 1$	p	atômica	N	S	N
SPANStore	$f + 1$ ou $2f + 1$	p	eventual / atômica	N	S	S
MDStore	$2f + 1$	B	atômica	N	N	N
Hybris	$f + 1$ a $2f + 1$	p / B	atômica	opc	S	S

F=tipo de falta, p=parada, B=bizantina, opc=opcional, EC=usa *erasure code*, *timer*=usa temporizadores, Latência=considera latências observadas pelo cliente.

★ Para o Vivace foi considerado apenas o algoritmo ABD modificado.

* Sugere uso de provedores com menor latência, pelo cliente.

Fonte: próprio autor (2017).

O armazenamento de dados com uso de *erasure code* é obrigatória apenas no RACS. No DepSky-CA, usa-se EC por necessidade de espalhar o fragmento da chave de cifragem. O Hybris apresenta o EC como possibilidade, ao custo de aumentar o número de provedores. Discussões sobre EC e replicação integral são apresentadas em ponto posterior desta tese (Seção 7.2.6).

3.2 COORDENAÇÃO DE METADADOS

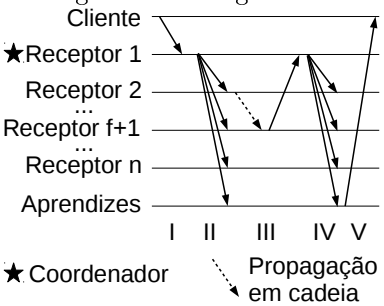
A coordenação de metadados tem como objetivo controlar a concorrência entre os múltiplos clientes escritores que acessam o sistema de armazenamento. Os metadados contém as informações que mantém os estados replicados. Nesta seção são apresentados alguns dos principais trabalhos que tratam de coordenação de estados e possuem relação com as contribuições desta tese. São eles: Multi-Ring Paxos, BAMCast, Raft e CRANE.

3.2.1 Multi-Ring Paxos

Multi-Ring Paxos (MARANDI; PRIMI; PEDONE, 2012) é um algoritmo que coordena a execução de instâncias do Ring Paxos (MARANDI et al., 2010). O serviço provido pelo Multi-Ring Paxos é particionado. Cada partição contém uma instância do Ring Paxos. Clientes podem comunicar-se com uma ou múltiplas partições.

O Ring Paxos tolera falta de parada, e seu funcionamento é descrito a seguir. Primeiro (Figura 17), o cliente envia o pedido a um receptor que possui o papel especial de coordenador (passo I). O Coordenador envia a proposta (o pedido com uma informação agregada pelo coordenador[7]) a todas as réplicas (receptores e aprendizes) (passo II). Os receptores se comunicam por meio de um anel lógico (passo III), de maneira semelhante a uma replicação em cadeia (Seção 2.1.1). O receptor i manifesta seu voto e encaminha-o ao receptor $i+1$. Os encaminhamentos terminam quando forma-se uma quantidade suficiente de votos (maioria). Nesse ponto, o receptor que possuir votos da maioria das réplicas encaminha esses votos ao coordenador. Ao verificar que a proposta foi aceita, o coordenador envia a todas as réplicas a decisão final sobre o pedido (passo IV). Os aprendizes podem executar o pedido e responder ao cliente (passo V).

Figura 17 – Ring Paxos.



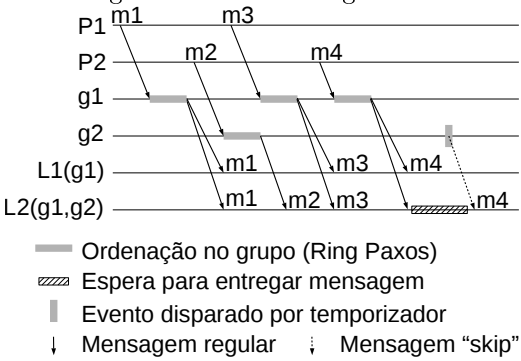
Fonte: adaptado de (MARANDI; PRIMI; PEDONE, 2012).

Como exemplo de execução do Multi-Ring Paxos, considere dois proponentes P1 e P2 (Figura 18). Existem dois aprendizes L1 e L2 e dois grupos denominados g1 e g2. O critério determinístico de aplicação

[7] O Ring Paxos adiciona um identificador único (ID) ao pedido. Esse ID será usado pelo aprendiz, ao final do processo, para executar o pedido em uma instância de consenso. Detalhes dessa utilização não estão no escopo deste texto.

de mensagens para aprendizes inscritos em mais de um grupo é simples: primeiramente são consideradas mensagens oriundas de g1, e a seguir são consideradas mensagens advindas de g2. L1 é inscrito para receber notificações do grupo g1. L2 recebe notificações de g1 e g2.

Figura 18 – Multi-Ring Paxos.



Fonte: adaptado de (MARANDI; PRIMI; PEDONE, 2012).

Inicialmente, P1 envia uma proposta m1 a g1. O grupo g1 resolve a ordenação da mensagem m1 (uma instância de Ring Paxos) e envia notificações aos aprendizes L1 e L2. A seguir, P2 envia m2 a g2. Após a execução de m2 em g2, apenas o aprendiz L2 é notificado, pois apenas L2 está inscrito em g2. O proponente P1 envia m3 a g1. Após a resolução de m3, L1 e L2 são notificados. Por fim, P2 envia m4 a g1. O grupo g1 resolve a ordenação de m4 e notifica L1 e L2. L1 recebe a notificação de g1 e aplica m4. L2 recebe m4 mas não pode aplicar. A última mensagem que L2 aplicou foi m3, vinda de g1. Portanto, L2 precisa receber uma mensagem de g2, para poder aplicar uma outra mensagem de g1. L2, portanto, aguarda. Em momento posterior (após um temporizador alcançar seu limite), g2 envia uma mensagem "skip" para L2. Nesse momento, L2 pode aplicar m4.

O protocolo Multi-Ring Paxos orquestra a interação entre instâncias do Ring-Paxos (denominadas *grupos*). A descrição a seguir omite a interação com o cliente para fins de simplicidade. Os aprendizes inscrevem-se nos grupos para receber mensagens. Se um aprendiz se inscreve em apenas um grupo, ele recebe mensagens ordenadas pelo grupo. Um aprendiz inscrito em múltiplos grupos usa um procedimento determinístico para reunir as mensagens entregues pelos diferentes grupos, por exemplo, a estratégia *round-robin*. Grupos enviam mensagens *skip* quando não possuem novas requisições.

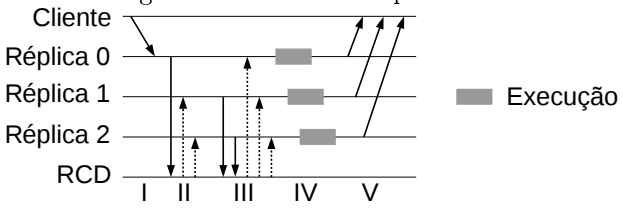
Ring Paxos é um protocolo desenvolvido para ter alto desempenho. O Multi-Ring Paxos pode ser usado para escalar o Ring Paxos, ampliando seu desempenho de forma praticamente linear, ao inserir mais anéis de Ring Paxos (BENZ; SOUSA; PEDONE, 2016). A escala linear deve-se ao fato do uso de particionamento nos dados.

3.2.2 BAMCast

BAMCast é um protocolo BFT para rede WAN (SILVA et al., 2013b). O protocolo é implementado sobre registradores compartilhados distribuídos (RCD) (Seção 2.2.2). O RCD é implementado em nível de máquina física, em uma rede exclusiva. Por meio de virtualização (Seção 2.3), máquinas virtuais (MV) têm acesso ao RCD de maneira restrita. Cada MV pode ler quaisquer registradores, mas pode escrever somente em um único registrador exclusivo. Essa abordagem é semelhante ao A2M (*Attested Append-Only Memory*), no qual é permitida apenas a anexação de dados (*append*) (CHUN et al., 2007).

Inicialmente (Figura 19), o cliente do BAMCast envia um pedido à réplica líder (fase I). O líder salva o pedido ordenado na RCD (fase II, mensagem *Propose*). A RCD notifica as demais réplicas sobre o pedido ordenado. Cada réplica responde positivamente (fase III, mensagem *Accept*) à solicitação do líder. Assim que cada réplica (incluindo o líder) possui aceites de uma maioria de réplica, executa o pedido (fase IV) e envia a resposta ao cliente (fase V). O cliente aceita a resposta quando possuir respostas iguais de uma maioria de réplicas.

Figura 19 – BAMCast: protocolo.



Fonte: adaptado de (SILVA et al., 2013a).

O uso de um componente confiável (RCD) permite que o sistema tolere faltas bizantinas com apenas $2f + 1$ réplicas, em vez de $3f + 1$. Além disso, o BAMCast é um protocolo com poucas mensagens de comunicação, ainda que os RCD comuniquem-se na rede exclusiva.

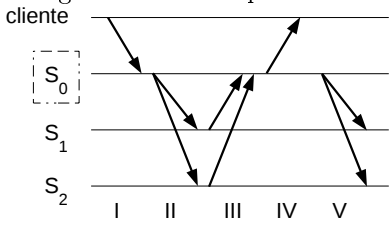
3.2.3 Raft

Raft é um algoritmo de consenso utilizado para replicar um registro de ações (*log*) (ONGARO; OUSTERHOUT, 2014). Um *log* pode ser utilizado para implementar RME. O Raft tolera faltas de parada e foi criado para ser mais compreensível que o Paxos, o que proporcionou o surgimento de muitas implementações (Apêndice A, item 29).

No Raft, uma réplica atua como líder e as demais réplicas funcionam como seguidoras. Atualizações no estado são enviadas apenas do líder para as réplicas, simplificando o gerenciamento do *log* replicado. O Raft usa temporizadores aleatórios na eleição do líder, o que resolve conflitos de forma simples. Por fim, a comunicação entre líder e réplicas inclui informações adicionais, como o último pedido executado pela réplica. Com o uso de quóruns (Seção 2.1.5) o Raft opera durante alterações em membros do grupo. O estado permanece consistente devido à interseção entre visões consecutivas diferentes.

A operação do Raft (Figura 20) inicia com o cliente enviando um pedido à réplica líder (fase I). O líder recebe o pedido, associa uma ordem ao mesmo e apresenta (*Propose*) o pedido ordenado às demais réplicas (fase II). Cada réplica seguidora aceita (*Accept*) a proposta do líder (fase III). Quando o líder reúne o aceite de uma maioria de réplicas (incluindo o próprio aceite), ele executa o pedido e responde (*Reply*) ao cliente (fase IV). Na ocorrência de um próximo pedido, ou por meio de um *heartbeat* (*piggyback*), o líder envia às demais réplicas a informação sobre o pedido aceito (*Commit*). Cada réplica seguidora pode, então, executar o pedido.

Figura 20 – Raft: protocolo.



Fonte: próprio autor (2017).

Réplicas recebem *heartbeats* do líder. Se após algum tempo uma réplica não receber comunicações do líder (*heartbeat* ou novos pedidos), a réplica torna-se *candidata* e solicita eleição. Réplicas candidatas enviam solicitações de votos *(RequestVote)* para as demais réplicas, que

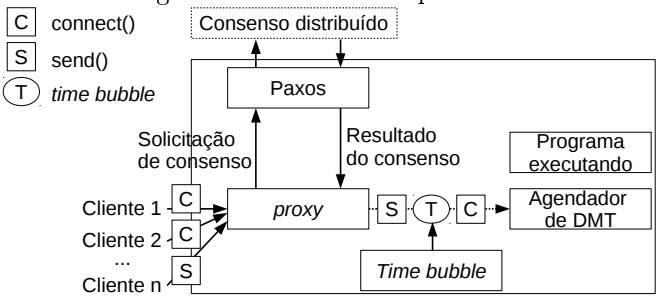
respondem as solicitações *(Vote)*. Uma réplica não aceita uma solicitação de voto apenas se: i) possui um *log* mais atualizado do que o *log* do proponente, e ii) ainda não há um líder eleito. Quando uma solicitação de voto é aceita por uma maioria de réplicas, escolhe-se um novo líder.

3.2.4 CRANE

CRANE é um protocolo de RME que trata o não-determinismo causado pela execução de pedidos em servidores com múltiplas *threads* (CUI et al., 2015). O não-determinismo de processamento é tratado por um algoritmo denominado *time bubbling*[8]. Relógios lógicos são usados para controlar a comunicação entre *threads*.

Os pedidos enviados ao CRANE são interceptados por um *proxy* (Figura 21). Quando uma requisição altera o estado do sistema (por exemplo, requisição S), uma solicitação de consenso (Paxos, neste caso) é feita para ordenar a execução da requisição. Após o resultado do consenso distribuído, as requisições são enviadas ao Agendador de DMT (*Deterministic MultiThreading*). Eventualmente são inseridos espaços de tempo (bolhas) pelo *Time bubble*, a fim de ordenar a execução entre diferentes *threads*. O Agendador de DMT atua na execução das requisições, em múltiplas *threads* do programa que está executando.

Figura 21 – CRANE: arquitetura.



Fonte: adaptado de (CUI et al., 2015).

O CRANE incorpora o Paxos via interceptação (mais detalhes na Seção 3.2.5). Cada requisição do cliente é capturada via *socket* e, quando necessário, o Paxos é acionado. A execução do Paxos é transparente, tanto para o cliente quando para a aplicação.

[8] Não-determinismo de processamento está fora do escopo desta tese.

3.2.5 Comentários sobre coordenação de metadados

Os procotolos apresentados nesta seção (3.2) podem ser utilizados para realizar a coordenação dos pedidos de armazenamento. Entretanto, em geral protocolos de coordenação de estados, por mais simples ou eficientes que sejam, requerem esforço da aplicação para implementar sua interação. CRANE (Seção 3.2.4) menciona a complexidade de usar interfaces de aplicações como ZooKeeper (Seção 2.2.6) para coordenar a replicação de estados.

Um algoritmo de coordenação de estados pode ser incorporado a um ambiente existente. Há pelo menos de três maneiras de realizar tal incorporação (LUNG et al., 2000; FELBER; NARASIMHAN, 2004; BESSANI; LUNG; FRAGA, 2005): integração, interceptação e serviço. Essas maneiras são descritas a seguir.

A abordagem de **integração** consiste em construir ou modificar um componente do ambiente, a fim de acrescentar uma funcionalidade. Na **interceptação**, mensagens enviadas aos destinatários são capturadas e mapeadas em um sistema de comunicação de grupo. Esse processo é feito de forma transparente à aplicação. Um exemplo é o uso de *interfaces* do sistema operacional (NARASIMHAN; MOSER; MELLIAR-SMITH, 1997) para interceptar *system calls* relacionadas à aplicação. Outro exemplo é o CRANE (Seção 3.2.4), que torna transparente a coordenação de estados realizada com o Paxos (Seção 2.1.4) por meio de interceptação dos pedidos. A abordagem de **serviço** define uma camada de *software* entre o cliente e a aplicação. Essa camada provê a execução ordenada dos pedidos enviados à aplicação.

Há diversos protocolos de coordenação de estados disponíveis na literatura (Seção 3.2). É possível incorporar um protocolo de coordenação de estados em um ambiente como o Kubernetes (Seção 2.3), que possui código aberto, o que possibilita o desenvolvimento de extensões de forma irrestrita. Tal proposta é inexistente até o presente momento.

Alguns trabalhos apresentam protocolos de coordenação de estados que fazem uso de componentes confiáveis. Por exemplo, *wormholes* podem ser usados como componentes confiáveis para permitir a difusão confiável de mensagens (VERONESE et al., 2013). No BAMCast (Seção 3.2.2), uma sub-rede confiável foi criada usando registradores distribuídos compartilhados. O Kubernetes possui um componente de armazenamento chamado *etcd* (Seção 2.3) que pode ser utilizado como um componente confiável. Dessa maneira, o uso de componentes préexistentes no Kubernetes pode permitir a criação de um protocolo de coordenação de estados simples de se projetar e integrar.

3.3 ARQUITETURAS DE ARMAZENAMENTO EM NUVEM

Nesta seção são descritos os trabalhos SCFS e GlobaFS. Ambos trabalhos apresentam arquiteturas que implementam as funcionalidades de armazenamento de dados e coordenação de metadados. Essas funcionalidades são disponíveis, entretanto, em módulos separados.

3.3.1 SCFS

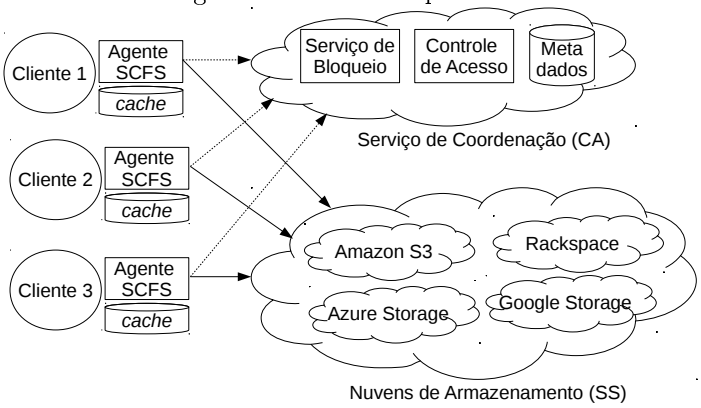
O SCFS (*Shared Cloud-backed File System*) é uma arquitetura que permite montar um sistema de arquivos utilizando provedores de nuvem (BESSANI et al., 2014). Dois componentes principais compõem a arquitetura do SCFS (Figura 22): um serviço de armazenamento (*Storage Service* - SS) e um coordenador de serviços (*Consistency Anchor* - CA). O SS é representado por provedores passivos (Seção 2.2.5). Sistemas de armazenamento de múltiplos provedores (exemplo: DepSky, Seção 3.1.3) podem ser utilizados como SS. O CA deve ser instanciado em servidores que tenham a capacidade de realizar processamento. Sistemas como o ZooKeeper (Seção 2.2.6) podem realizar a coordenação de estados. Provedores comerciais oferecem apenas consistência eventual, que pode não ser suficiente para alguns tipos de aplicações (Seção 2.2.2). O cliente possui um *cache* que contém as informações do sistema de arquivos replicado. Junto ao cliente encontra-se também um agente do SCFS que comunica-se com o SS e com o CA.

O SCFS consegue prover consistência forte utilizando provedores que oferecem consistência eventual. Para alcançar essa característica, os procedimentos de escrita e leitura foram modificados conforme segue. Considere um par chave-valor com os identificadores id e v, respectivamente. Para realizar uma escrita de dados (Figura 23(a)), o agente primeiramente gera um *hash* h de v. Em seguida, o cliente grava v no SS, sob a chave $id{+}h$ (concatenação). Após a gravação de v, h é gravado no CA, sob a chave id. Estas informações do CA garantem a consistência forte, e são chamadas de *âncoras de consistência*.

Para ler um dado (Figura 23(b)), o agente solicita ao CA o *hash* do dado. Após obter h, o agente busca o dado no SS, utilizando a chave $id{+}h$, até obter o valor v. Por fim, é verificado se o *hash* do valor retornado pelo SS é igual ao *hash* fornecido pelo CA. Caso não seja, é retornado nulo.

O SCFS faz uso de um *cache* local para melhorar a eficiência das operações. Quando um usuário abre um arquivo, o agente lê o

Figura 22 – SCFS: arquitetura.



Fonte: adaptado de (BESSANI et al., 2014).

Figura 23 – SCFS: algoritmos de (a) escrita e (b) leitura.

WRITE(id, v):
w1: $h \leftarrow \mathsf{Hash}(v)$
w2: SS.write($id \mid h, v$)
w3: CA.write(id, h)

(a)

READ(id):
r1: $h \leftarrow$ CA.read(id)
r2: **while** $v = null$ **do** $v \leftarrow$ SS.read($id \mid h$)
r3: **return** $(\mathsf{Hash}(v) = h)?v : null$

(b)

Fonte: (BESSANI et al., 2014).

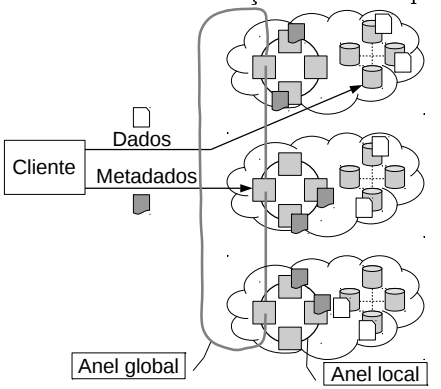
metadado (de CA). Se a operação for de escrita, é criado um *lock* para o arquivo. A seguir, o agente lê o conteúdo do arquivo (de SS) e salva este conteúdo no *cache* local. As operações de leitura e escrita são operações locais (i.e., operam sobre o *cache*). Quando ocorre escrita em um arquivo, é feita atualização do conteúdo em *cache* e em partes do metadado (tamanho e data da última atualização). A leitura de um arquivo envolve apenas a busca do dado no *cache*, já que o conteúdo foi obtido (de SS) quando o arquivo foi aberto. Fechar um arquivo requer a sincronização dos dados e metadados. Primeiramente, o arquivo é copiado para o disco local e para as nuvens (SS). A seguir, o metadado no *cache* é modificado, e enviado ao serviço de coordenação (CA). Por fim, o *lock* é removido, caso o arquivo tenha sido aberto para escrita.

3.3.2 GlobalFS

GlobalFS (PACHECO et al., 2016) implementa um sistema de arquivos distribuído em provedores de nuvem. Arquivos e pastas são armazenados em *data centers* específicos, de acordo com o padrão de acesso. É possível executar operações locais e inter-regiões. Operações locais resultam em baixa latência. Operações que envolvem mais de uma região usam *multicast* atômico para garantir a consistência. São toleradas faltas de parada.

Dados e metadados são tratados separadamente (Figura 24). Dados são distribuídos (particionados) segundo um esquema de *distributed hash tables* (DHT). A replicação de dados depende do padrão de acesso do dado. Dados muito lidos e pouco alterados (exemplo: dados de programas e aplicações) localizam-se em uma partição global. A replicação é feita em *data centers* de diferentes regiões e tolera-se faltas de maior amplitude (desastres). Assim, dados podem ser lidos de várias regiões, obtendo-se baixa latência. Dados dependentes de localidade (exemplo: arquivos temporários do cliente) são replicados em *data centers* de uma mesma região. Esse cenário configura uma partição local.

Figura 24 – GlobalFS: interação entre componentes.



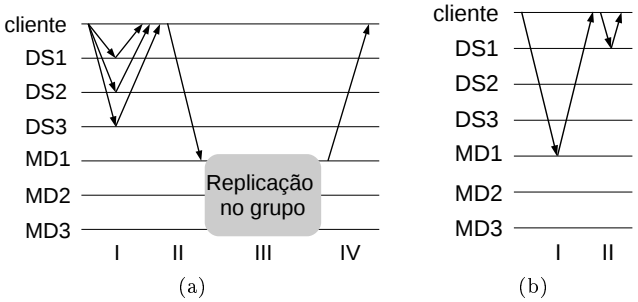
Fonte: adaptado de (PACHECO et al., 2016).

Os metadados contém informações sobre o arquivo (tamanho do arquivo, proprietário, diretos de acesso) e ponteiros para os blocos de dados. Metadados são replicados usando RME (Seção 2.1.4). O *multicast* atômico (anel global) é implementado via Multi-Ring Paxos (Seção 3.2.1).

Operações em partições locais são executadas via RME em *data centers* de mesma região. Apenas uma réplica responde ao cliente. Operações multi-partições não-coordenadas (operações que não possuem relações entre si) podem ser executadas simultaneamente nas respectivas partições de maneira independente. Operações multi-partições coordenadas (por exemplo, renomear um arquivo, movendo-o de partição) requer mais passos de execução. Todas as partições envolvidas com a execução de um comando são contactadas, para informar se o comando pode ser executado ou não. De forma similar a um protocolo de efetivação de duas fases (Seção 2.1.4), cada comando só é executado quando todas as partições confirmam a possibilidade de execução. Operações somente-leitura são executadas em apenas uma réplica.

Para escrever um dado (Figura 25(a)), o cliente inicialmente cria o bloco de dados, sob um identificador único gerado aleatoriamente. O cliente então envia o dado às réplicas de dados (DS1..DS3), e aguarda a confirmação da gravação (fase I). A seguir (fase II), envia os metadados à replica de metadados mais próxima (MD1). A réplica de metadados executa a replicação do metadado (fase III). A seguir, o cliente obtém a confirmação da gravação do dado (fase IV). A leitura de um dado (Figura 25(b)) inicia com a obtenção dos metadados na réplica de metadados mais próxima (fase I). A seguir, o cliente obtém o bloco de dados da réplica de dados mais próxima (fase II).

Figura 25 – GlobalFS: operações de (a) escrita e (b) leitura.



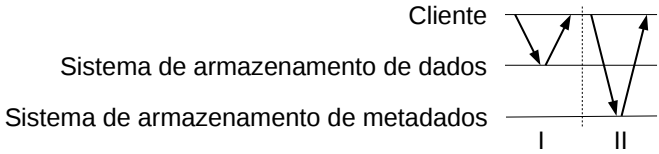
Fonte: próprio autor (2017).

3.3.3 Comentários sobre arquiteturas de armazenamento em nuvens

As arquiteturas GlobalFS (Seção 3.3.2) e SCFS (Seção 3.3.1) separam as camadas de armazenamento de dados e coordenação de metadados. Esse tratamento diferenciado permitiu o surgimento de estratégias como o uso de réplicas testemunhas em votações (PÂRIS; LONG, 1991), réplicas preferidas (§ 3.1.5, 3.1.6) e a redução no número de réplicas que armazenam metadados (MDStore, Seção 3.1.7).

Em um sistema de arquitetura modular, a operação de escrita pode ser representada em forma de protocolo (Figura 26). Primeiramente (fase I), um sistema realiza o armazenamento de dados. A seguir (fase II), outro sistema armazena os metadados de forma coordenada. Essa sequência de fases é importante para dar visibilidade aos dados apenas quando os metadados estiverem efetivamente gravados.

Figura 26 – Operação de escrita representada como protocolo composto por diferentes sistemas.



Fonte: próprio autor (2017).

Arquiteturas de armazenamento (Seção 3.3) são compostas de subsistemas (Tabela 2). O SCFS usa o DepSky (Seção 3.1.3) para armazenar dados e o BFT-SMaRt (Seção 2.1.4) para armazenar metadados. O GlobalFS usa quóruns (Seção 2.1.5) para armazenar dados e o Multi-Ring Paxos (Seção 3.2.1) para armazenar metadados. O sistema FITS (Capítulo 6), proposto nesta tese, armazena dados usando o RafeStore (Capítulo 4) e armazena metadados de maneira coordenada usando o DORADO (Capítulo 5).

3.4 CONCLUSÕES DO CAPÍTULO

Neste capítulo foi apresentada a literatura relacionada a armazenamento de dados geo-distribuídos (Seção 3.1), coordenação de metadados (Seção 3.2) e arquiteturas de sistemas de armazenamento em provedores de nuvem (Seção 3.3). Com base nessa literatura, foram

Tabela 2 – Composições de sistemas de armazenamento.

	Armazenamento de...	
Sistema Composto	Dados	Metadados
SCFS	DepSky	BFT-SMaRt
GlobalFS	Quóruns	Multi-Ring Paxos
FITS	RafeStore	DORADO

Fonte: próprio autor (2017).

identificadas oportunidades de desenvolvimento para um novo protocolo de armazenamento (Seção 3.1.9), bem como integração de coordenação de metadados (Seção 3.2.5) em um ambiente disponível.

As soluções para armazenamento de dados e coordenação de metadados estão desenvolvidas de forma independente nesta tese, sob os sistemas RafeStore (Capítulo 4) e DORADO (Capítulo 5), e estão acopladas pelo sistema FITS (Capítulo 6). O prelúdio do sistema FITS e sua característica de composição teve por objetivo fornecer uma melhor compreensão do papel de suas componentes, que são detalhadas nos capítulos a seguir.

4 ARMAZENAMENTO DE DADOS COM ANTECIPAÇÃO DE PEDIDOS

Se alguém rir da sua ideia, veja isso como um sinal de potencial sucesso!
Jim Rogers

Este Capítulo apresenta a Antecipação de Pedidos (AdP) (Seção 4.1) e o *RafeStore*, que aplica a AdP no contexto de armazenamento de dados em múltiplos provedores de nuvens (Seção 4.1.1). Essas duas seções fundamentam a definição do termo "dado de conteúdo independente", que é utilizado na seção de modelo de sistema (Seção 4.1.2). Os algoritmos (Seção 4.1.2) detalham o funcionamento do sistema de armazenamento com a aplicação da técnica AdP. Por fim, conclusões (Seção 4.2) encerram o Capítulo. O termo "réplica" é eventualmente usado neste capítulo para referenciar locais de armazenamento de dados (provedores de nuvens ou repositórios locais).

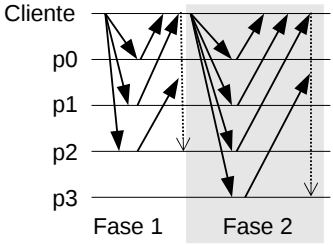
4.1 PROPOSTA DA TÉCNICA DE ANTECIPAÇÃO DE PEDIDOS

Um fato relevante na exposição inicial da técnica AdP é que sistemas que usam quóruns contam com respostas de apenas um subconjunto de réplicas existentes no sistema (Seção 2.1.5). Graças à existência de réplicas redundantes cujas respostas são opcionais, quóruns toleram faltas por meio de mascaramento. A interseção entre subconjuntos (quóruns) de réplicas contém um número suficiente de servidores corretos (maioria). Assim, a consistência do dado replicado é garantida.

É comum cancelar requisições em operações que envolvem quóruns. Para ilustrar essa situação, considere o exemplo genérico a seguir (Figura 27). Quatro servidores (p0...p3) são envolvidos na operação de um protocolo que possui duas fases. Na primeira fase, três servidores são contactados. Quando respostas chegam de dois servidores, a requisição enviada ao terceiro servidor é cancelada. Na segunda fase, quatro servidores são contactados. Quando o cliente possui respostas de três servidores, a requisição enviada ao quarto servidor é cancelada.

O protocolo ilustrado (Figura 27) possui uma desvantagem ao ser aplicado no cenário de computação em nuvem, especificamente no contexto de armazenamento de dados. De maneira mais concreta, o exemplo pode representar a gravação de dados (fase I) seguida pela gravação

Figura 27 – Exemplo de protocolo que usa quóruns.



Fonte: próprio autor (2017).

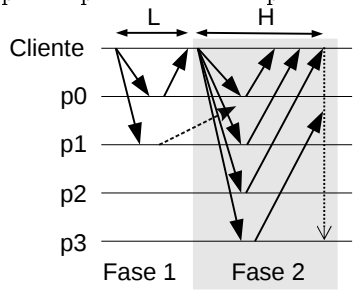
de metadados (fase II). Ao instanciar esse protocolo em um cenário de computação em nuvens, o cancelamento da operação de escrita pode deixar dados residuais armazenados no servidor. A quantidade de dados escritos antes do cancelamento pode variar de 0% a 100% do dado a ser gravado. Isso implica em possíveis custos adicionais. No momento da criação da proposta da AdP (NETTO et al., 2015), o provedor Amazon não cobrava pela operação de escrita de dados. No momento da escrita desta tese, cobra-se por operação de escrita, sendo este mais caro do que o valor da operação de leitura (Apêndice A, item 28, Escrita = \$0.01 por 1000 solicitações, Leitura = \$0.01 por 10.000 solicitações). Além desses custos, o armazenamento de dados é taxado pela quantidade de dados gravados no provedor. Dessa forma, cancelar requisições pode resultar em custo extra de armazenamento inutilizado.

Nesta tese é proposta a redução do desperdício causado pelo cancelamento de pedidos por meio de uma antecipação de fases de comunicação. Antecipação, neste contexto, significa enviar requisições de uma próxima fase antes do término da fase atual. Outra característica presente nessa técnica é a paralelização do tratamento de pedidos, pois pedidos de uma fase podem terminar durante as próximas fases. Perde-se isolamento entre fases, mas ganha-se no tempo total de execução, bem como na eficiência da realização de tarefas. Por meio do uso de informações prévias dos componentes utilizados nas fases, aliado ao tratamento dos pedidos entre as fases, torna-se possível dispensar o envio excedente de pedidos. A estratégia de paralelizar o tratamento de pedidos entre fases foi denominada Antecipação de Pedidos (AdP).

É possível aplicar a AdP ao protocolo que usa quóruns exemplificado anteriormente. A antecipação ocorre conforme descrito a seguir. Quando a primeira resposta da fase 1 chega ao cliente (advinda de p0), a fase 2 é iniciada (Figura 28). Quando a fase 2 termina, é verificado se

as respostas remanescentes da fase 1 chegaram. Caso essa verificação se confirme, o protocolo é encerrado. Se ao término da fase 2 uma ou mais respostas da fase 1 ainda não chegaram, é preciso tomar ações extras para atingir o número de respostas esperado. Essas ações adicionais, bem como a duração das fases (L e H) são detalhados mais adiante.

Figura 28 – Antecipando pedidos em um protocolo que usa quóruns.



Fonte: próprio autor (2017).

4.1.1 AdP em armazenamento de dados

Há um tipo de armazenamento no qual o dado possui múltiplas versões e o conteúdo do dado não tem relação com valores anteriores do mesmo dado. Realizar armazenamento segundo essa abordagem é viável em diversos cenários. Por exemplo, considere um sistema que organiza uma fila de pessoas.A fila pode ser modelada como a sequência de valores de uma variável. Cada novo valor (novo nome inserido na fila) é gravado sob uma nova versão da variável. O primeiro valor recebe a versão 1, o segundo valor é associado à versão 2, e assim por diante. Os nomes das pessoas que estão na fila não possuem relação de dependência entre si.

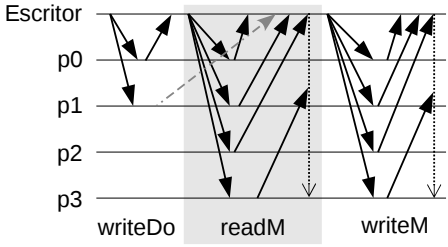
Outro exemplo pode ser um fórum de discussão. O tópico do fórum pode ser definido como o valor inicial de uma variável. Postagens sequenciais ao tópico podem ser consideradas como próximos valores da variável. Assim, cada postagem é associada a uma versão da variável. Para listar a discussão basta mostrar os valores da variável, em todas as suas versões. Em uma lista de discussão é comum que as postagens estejam relacionadas, mas seus conteúdos não são obrigatoriamente dependentes entre si. É possível, por exemplo, que um usuário poste uma propaganda que não tem nenhuma relação com as demais

postagens. Adicionalmente, em alguns fóruns (exemplo: Facebook) não há problema se surgirem postagens entre o tempo que o usuário lê as mensagens do fórum e posta uma nova mensagem.

Quando o conteúdo do dado não tem relação com seus valores anteriores, pode-se definir que esse dado possui *conteúdo independente*. Novos valores podem ser escritos como *blind writes*, ou operações de somente-escrita (AGRAWAL; KRISHNASWAMY, 1991). Para um dado multiversão (Seção 2.2.2) de conteúdo independente, apenas os metadados são atualizados a partir de valores anteriores. Nos metadados encontram-se as versões do dado, que são únicas e sequenciais.

É possível antecipar pedidos ao armazenar dados de conteúdo independente em múltiplos provedores de nuvem. Considere o exemplo de um protocolo com quóruns em três fases (Figura 29), no qual quatro réplicas (p0...p3) armazenam metadados (como no DepSky, Seção 3.1.3) e três réplicas (p0...p2) armazenam dados (como no MDStore, Seção 3.1.7). Com essa configuração é possível tolerar uma falta bizantina no sistema. No MDStore, três réplicas são contactadas para garantir que duas réplicas armazenam os dados. Com uso da AdP, somente duas réplicas são contactadas para armazenar dados. Apenas em casos com faltas, uma terceira réplica é contactada.

Figura 29 – Exemplo de escrita de dado multiversão com AdP, caso normal.

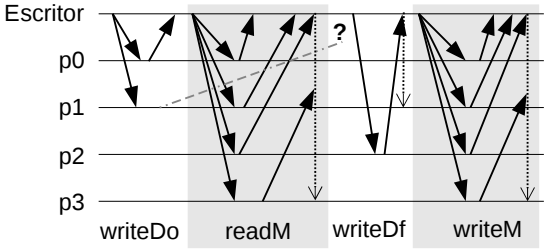


Fonte: próprio autor (2017).

Inicialmente (Figura 29), o novo valor do dado é enviado para ser escrito nas réplicas p0 e p1 (fase writeDo). Quando a primeira resposta da fase writeDo chega (oriunda da réplica p0), a fase readM é iniciada, a fim de obter a última versão do dado. Ao término da fase readM, é verificado se a resposta remanescente da fase writeDo chegou. Em um caso de sucesso (Figura 29), todas as respostas da fase writeDo chegam antes do término da fase readM. Na sequência, atualiza-se o metadado com uma nova versão do dado (fase writeM).

É possível que ao término da fase readM existam requisições iniciadas na fase writeDo que não foram concluídas (Figura 30). Nesse caso, uma fase adicional (writeDf) é necessária. A terceira réplica (p2) é contactada para armazenar o dado. A escrita de metadados (fase writeM) só pode ser iniciada após o cliente obter a confirmação de gravação do dado de mais uma réplica, totalizando duas confirmações. No exemplo (Figura 30), considera-se que a réplica adicional contactada (p2) confirma a gravação. O pedido enviado para a réplica de dados inicialmente contactada (p1) é cancelado e a fase de escrita da nova versão do dado é executada.

Figura 30 – Exemplo de escrita de dado multiversão com AdP, caso com falta.



Fonte: próprio autor (2017).

O uso da antecipação de pedidos requer a consideração do tempo de percurso entre o cliente e as réplicas. Nesse caso, é proposto o uso do tempo de ida e volta de um sinal em um caminho de rede: o *round-trip time* (RTT). Para fins de simplificação, o RTT é considerado como a latência entre o cliente e a réplica[1]. A fim de obter bom desempenho, uma das réplicas de dados (p0, na Figura 28) deve ser a mais rápida, no conjunto das réplicas. Dessa forma, o início da fase de leitura de metadados é iniciada o mais cedo possível. Denota-se a latência dessa réplica de dados mais rápida pela letra L. A latência representada pela letra H é a latência despendida na fase seguinte (Fase 2, na Figura 28). Pode-se concluir que a latência da segunda réplica de dados (p1) deve ser menor ou igual a L+H.

Reiterando as afirmações, L representa a latência da réplica mais rápida, dentre todas as réplicas (nesse caso, p0). H significa a latência da réplica mais lenta existente apenas no subconjunto das $2f+1$ réplicas mais rápidas (nesse caso, p2). Denota-se S como a latência da réplica

[1] Na Seção 7.2 a diferença entre esses conceitos emergirá.

de dados (i.e., que participa da Fase 1) mais lenta (nesse caso, p1). Obtém-se assim a condição que torna favorável o uso da AdP:

Definição 1. *A latência das réplicas deve obedecer à regra:* $S \leq (L + H)$.

É notável que pelo menos uma réplica que armazena dados deve oferecer a menor latência possível. Quando isso ocorre, a próxima fase é antecipada o mais breve possível. Além disso, as demais réplicas de dados devem possuir latências menores do que as réplicas de metadados. A recomendação geral, portanto, é que o provedor mais lento não deve ser usado para armazenar dados, mas sim metadados.

4.1.2 Modelo de sistema

Considera-se um sistema distribuído assíncrono. Consumidores e provedores estão conectados por uma rede que possui conexões confiáveis, que significa que em algum momento as mensagens são entregues. O dado é armazenado em uma estrutura do tipo chave-valor. Usuários podem ler e escrever dados nos provedores de armazenamento por meio de *interfaces read*(chave) e *write*(chave, valor), respectivamente. Essas *interfaces* mantém a conexão aberta até receber uma resposta do provedor, ou até que a operação seja cancelada. Não há concorrência de escrita, é possível que exista um escritor e múltiplos leitores (SWMR, Seção 2.2.2). Os provedores são entidades passivas, no sentido de que eles não executam código ou comunicam-se entre eles (Seção 2.2.5). Usuários interagem com os provedores por meio de uma biblioteca. O dado é multiversão de conteúdo independente (Seção 4.1.1).

O sistema utiliza quóruns (Seção 2.1.5) para tolerar faltas bizantinas. Usuários podem falhar arbitrariamente enquanto lêem dados. Entretanto, usuários podem falhar apenas por parada durante operações de escrita. Provedores são corretos ou podem apresentar comportamento Bizantino. O número de provedores de metadados contactados é igual a $3f + 1$, sendo f o número máximo de faltas toleradas. São necessários $2f + 1$ provedores para armazenar os dados. Porém, em casos nos quais não ocorrem faltas, apenas $f + 1$ provedores são contactados.

4.1.3 Projeto do RafeStore

O cenário de computação em nuvem é propício para a aplicação da AdP. No cenário de armazenamento de dados, ao contactar um menor número de provedores (comparado a trabalhos anteriores como o

DepSky, Seção 3.1.3) custos de rede e armazenamento são reduzidos. Outra consequência da aplicação da AdP é a redução na latência observada pelo cliente. Isso torna-se relevante especialmente no cenário de computação em nuvens, onde as latências são elevadas.

Foi criado o sistema de armazenamento de dados nomeado *RafeStore* (***R**eliable, **a**vailable, **f**ast and **e**conomic **Stor**age*), no qual é utilizada a AdP. No RafeStore, dados são escritos antes de qualquer leitura de metadados (Figura 29). Logo, o identificador do dado não inclui obrigatoriamente identificação da versão do dado. É comum representar dados de uma variável chamada "nome" como "dados-nome". Os dados dessa variável podem ser representados como "dados-versao1-nome", "dados-versao2-nome", etc.

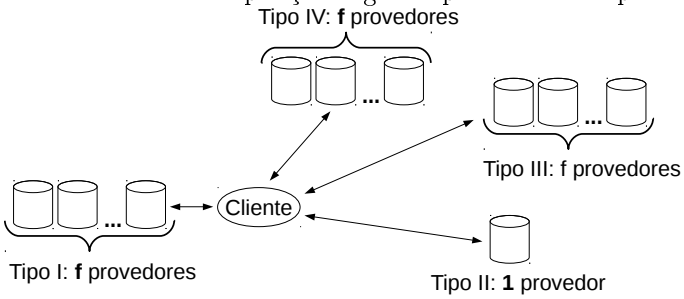
O identificador do dado é composto por um identificador universal e o resumo criptográfico (*hash*) do dado. O identificador universal é o UUID (*Universally Unique IDentifier*) (LEACH; MEALLING; SALZ, 2005) e o *hash* é gerado pelo algoritmo MD5 (Seção 2.2.3). O UUID é gerado considerando características do cliente e o *hash* é gerado considerando características do dado. Cada operação de escrita gera uma nova versão do dado, que é registrada no arquivo de metadados. Um registro de metadado pode ser representado pela tupla (v, uId, (r_0,...,r_f), h), onde: v é a versão do dado, uId é o identificador do dado, r_i ($i = 0...f$) é o provedor de dados de ordem i e h é o *hash* do dado. Um único arquivo de metadados registra informações de todas as versões do dado.

O conhecimento prévio das latências dos provedores é útil para escolher quais provedores devem ser responsáveis por armazenar dados e quais devem armazenar apenas metadados. Pode-se estabelecer (Figura 31) sugestões para permitir o funcionamento correto do RafeStore:

- Deve-se usar f provedores próximos ao usuário (tipo I), a fim de favorecer o desempenho. Esses provedores são destinados à redundância dos dados e metadados. Logo, podem ser provedores simples (baratos, de menor qualidade) ou privados. Dados e metadados são armazenados nesses provedores.
- Deve-se manter pelo menos um provedor robusto (tipo II) distante, para aumentar a sobrevivência do dado (COUTO et al., 2014). Um dado geograficamente distribuído corre menos risco de desaparecer mediante catástrofes de larga escala (Seção 2.2.5). Essa réplica deve manter a cópia mais confiável dos dados.
- Deve-se manter f provedores distantes (tipo III). Esses provedores devem estar tão distantes quanto o provedor de dados mais distante, a fim de manter válida a Definição 1. Esses provedores

armazenam principalmente metadados, mas podem ser contactados para armazenar dados, em caso de faltas.
- Os demais f provedores (tipo IV) armazenam apenas metadados, e não possuem recomendações especiais.

Figura 31 – RafeStore: disposição sugerida para usuário e provedores.



Fonte: próprio autor (2017).

Pode-se orientar a escolha de provedores considerando também a consistência oferecida (Seção 2.2.5). O provedor do tipo II deve usufruir de consistência forte (exemplo: oferecida pelo provedor Azure, Seção 2.2.5). É oportuno também que pelo menos um provedor do tipo I ofereça consistência forte (preferencialmente, aquele que fornece a primeira resposta na antecipação de pedidos). Demais provedores podem ser atendidos por consistência eventual (exemplo: oferecida pelo provedor Amazon).

4.1.4 Algoritmos do RafeStore

Nesta seção são descritos os algoritmos do RafeStore, que realizam escrita e leitura de dados. A escrita de dados no RafeStore é executada pelo cliente por meio do comando write(key, value) (Algoritmo 1, linha 1). Inicialmente, um identificador único do dado é gerado (linha 2). A seguir, $f + 1$ provedores de dados são contactados simultaneamente para armazenar o dado (linha 5). A função wQuorum (Algoritmo 3, linha 1) envia pedidos de gravação de dados a provedores. Quando o cliente recebe a resposta do primeiro provedor de dados (Algoritmo 1, linha 6), inicia-se a leitura dos metadados (linha 7). Durante a leitura de metadados, a gravação de dados continua ocorrendo, pois a requisição aos provedores é feita de forma paralela (*parallel for* dispara *threads*, Algoritmo 3).

Quando a leitura de metadados termina, o cliente verifica se todas as requisições de escrita de dado foram concluídas (linha 8). Se essa verificação resultar em falso, mais provedores de dados são solicitados a gravar dados (linha 9). De qualquer maneira, a escrita de metadados é iniciada apenas quando pelo menos $f+1$ provedores confirmarem que o dado foi salvo (linha 10). Para minimizar o uso de rede e custos, ao término da fase de escrita de dados, requisições de escrita pendentes são canceladas[2] (linha 11). Uma nova versão do dado é calculada (linha 12), o novo metadado é construído (linha 13) e o novo metadado é concatenado com o metadado anterior (linha 14). Todos os provedores de metadados são solicitados a armazenar o metadado (linha 17). Porém, para concluir essa operação, apenas um quórum de $2f+1$ respostas positivas é suficiente (linha 18). Requisições de escrita de metadado pendentes são canceladas, a fim de concluir a operação (linha 19).

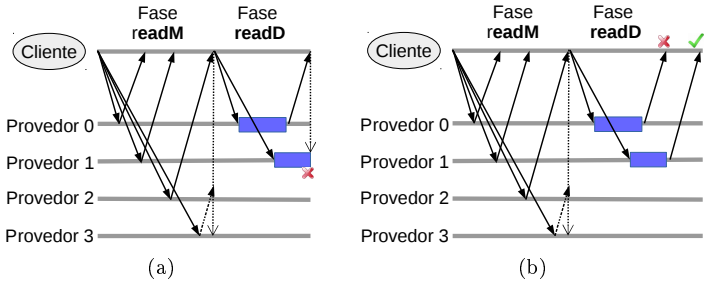
Algoritmo 1 RafeStore: operação de escrita

```
1:  procedure WRITE(key, value)
2:      uId := UUID + "_" + hash(value)
3:      global c := 0   ▷ quant. de resp. da escrita de dados, ver alg. 3
4:      global va := [ ]        ▷ conjunto de respostas válidas, ver alg. 3
5:      wQuorum(uId, value, 0, f)                  ▷ inicia a fase writeDo
6:      wait until c ≥ 1
7:      m := readMetadata(key, 2 * f + 1)      ▷ executa a fase readM
8:      if c < (f + 1) then
9:          wQuorum(uId, value, f + 1, 2 * f)     ▷ inicia fase writeDf
10:     wait until c ≥ f + 1
11:     parallel for 0 ≤ i ≤ 2 * f do provider_i.cancelPending()
12:     newDataVersion := max(m[i] : 0 ≤ i ≤ 2 * f) + 1
13:     newMetadata := newDataVersion + ";"+ uId + ";" + va
14:     newMetadata := m + newMetadata
15:     mt := ⊥        ▷ inicializa conjunto de provedores de metadados
16:     parallel for 0 ≤ i ≤ n − 1 do
17:         mt[i] := provider_i.put(key+"m",newMetadata)
18:     wait until |{i : mt[i] = "ok"}| ≥ 2 * f + 1
19:     parallel for 0 ≤ i ≤ n − 1 do provider_i.cancelPending()
```

[2]O cancelamento de requisições em provedores é ignorado para réplicas que já forneceram resposta. É possível incluir uma operação de limpeza para remover dados residuais do provedor, e assim eliminar custos de armazenamento indevido.

A leitura de dados (Figura 32(a)) inicia com o cliente requisitando metadados aos provedores de metadados (Algoritmo 2, linha 2). Ao receber respostas suficientes ($2f + 1$), os provedores de dados são requisitados sobre o dado (linhas 3 e 4). O cliente aguarda até receber uma resposta válida (linha 5). Requisições de dados pendentes são canceladas (linha 6) e o valor do dado é entregue ao cliente (linha 7).

Figura 32 – Operação de leitura: (a) caso normal e (b) com falta.



Fonte: próprio autor (2017).

Se o primeiro dado lido pelo cliente for inválido (Figura 32(b)), é preciso aguardar a chegada de dados lidos de outros provedores. Uma leitura pode falhar, dentre outros motivos, por atraso no envio do dado (por parte de um provedor) ou por um dado corrompido.

Algoritmo 2 RafeStore: operação de leitura

```
1: function READ(key)
2:     m := readMetadata(key, 2f + 1)
3:     parallel for i in m.providersId do
4:         data[i] := provider_i.get(m.getUniqueId())
5:     wait until ∃x|data[x] is valid
6:     parallel for i in m.providersId do provider_i.cancelPending()
7:     return data[x]|data[x] is valid
```

Existem ações auxiliares incluídas nas operações de leitura e escrita. O procedimento wQuorum (Algoritmo 3, linha 1) contacta provedores desde o índice p_{ini} até o índice p_{end}, a fim de salvar um valor *value* sob uma referência *key*. Na medida em que os provedores confirmam a escrita (linha 4), variáveis globais (declaradas no Algoritmo 1) são atualizadas (linhas 5 e 6) para refletirem o estado atual do quórum de respostas. A função readMetadata (linha 7) solicita aos provedores

de metadados informações que estão sob a referência *key* (linha 10). Quando o número suficiente de respostas corretas (*threshold*) é alcançado (linha 13), solicitações pendentes são canceladas (linha 14) e o metadado é retornado ao cliente (linha 15).

Algoritmo 3 Algoritmos auxiliares

```
1:  procedure WQUORUM(key, value, ini, end)
2:      parallel for ini ≤ i ≤ end do
3:          ok[i] := provider_i.put(key, value)
4:          if ok[i] = "ok" then
5:              c := c + 1          ▷ nova confirmação de escrita de dados
6:              va := va ∪ i                          ▷ nova resposta válida
7:  function READMETADATA(key, threshold)
8:      m :=⊥
9:      parallel for 0 ≤ i ≤ n − 1 do
10:         t_i := provider_i.get(key+ "m")
11:         if verify(t_i) then
12:             m[i] := t_i
13:     wait until |{∀i : m[i] ≠⊥}| ≥ threshold
14:     parallel for 0 ≤ i ≤ n − 1 do provider_i.cancelPending()
15:     return m
```

O seguinte raciocínio apresenta a lógica de funcionamento do protocolo. A disponibilidade dos dados é garantida porque o dado é armazenado em um quórum de $f + 1$ provedores, de tal forma que até f provedores podem falhar. A operação de leitura obterá o dado do primeiro provedor disponível que fornecer uma resposta correta. A integridade do dado pode ser verificada com o *hash* que se encontra junto aos metadados. Caso o dado esteja corrompido, o mesmo poderá ser lido de outro provedor. Os metadados também estarão sempre disponíveis, pois em um quórum de $3f + 1$ provedores, pelo menos $2f + 1$ provedores armazenam os metadados. A leitura de metadados recuperará $2f + 1$ metadados, dos quais pelo menos $f + 1$ devem ser iguais e atualizados. Com $f + 1$ metadados iguais, é possível atestar a atualidade do metadado (não foi enviada uma resposta antiga). O metadado é auto-verificável (criptografia assimétrica) e, portanto, íntegro.

4.2 CONCLUSÕES DO CAPÍTULO

Este capítulo apresentou o *RafeStore*, um sistema eficiente de armazenamento de dados em múltiplos provedores de nuvem. O *RafeStore* tolera faltas bizantinas, um característica importante ao se considerar o uso de provedores públicos (Seção 2.2.5). A eficiência do *RafeStore* deve-se ao uso da técnica de Antecipação de Pedidos (AdP), também descrita neste capítulo. Outro fator de aceleração de desempenho do *RafeStore* é o uso de dispositivos de armazenamento próximos ao cliente, o que maximiza o paralelismo oferecido pela AdP.

Uma limitação do *RafeStore* é a semântica SWMR, que permite a atuação de um escritor por vez, em cada dado. O próximo capítulo descreve o DORADO, uma solução para realizar a coordenação de metadados e assim fornecer uma semântica MWMR. Assim é possível controlar a concorrência em cenários de múltiplos clientes lendo e escrevendo informações em um dado.

5 COORDENAÇÃO DE METADADOS NO KUBERNETES

Entre no jogo!
Helen Tomas

Este capítulo apresenta o DORADO (*or**D**ering **O**ve**R** sh**A**re**D** mem**O**ry*), um protocolo para coordenar metadados em um sistema de armazenamento em provedores de nuvem. Inicialmente, motiva-se a integração de coordenação de metadados no ambiente Kubernetes (Seção 5.1). A seguir, a proposta de integração (Seção 5.2) bem como o modelo de sistema (Seção 5.2.1) são descritos. Por fim, a especificação (Seção 5.2.2) e os algoritmos (Seção 5.2.3) detalham a operação das réplicas[1] do DORADO.

5.1 MOTIVAÇÃO PARA INTEGRAR COORDENAÇÃO DE METADADOS EM UM GERENCIADOR DE *CONTAINERS*

A utilização de máquinas virtuais (MV) em *data centers* tornou prático o provisionamento dinâmico de recursos. Por consequência, foi possível implantar o modelo de custos sob demanda (*pay-per-use*). Nesse contexto, *containers* (Seção 2.3) tornam o provisionamento de recursos ainda mais rápido e flexível, quando comparados às tradicionais MV (XAVIER et al., 2013; FELTER et al., 2015). Diversas companhias se reuniram e fundaram a *Cloud Native Computing Foundation* (CNCF). Um dos objetivos da CNCF é orientar a adoção de *containers* nos provedores de computação em nuvem.

O primeiro resultado prático da CNCF é o Kubernetes (Seção 2.3). Utilizar o Kubernetes na coordenação de metadados é vantajoso devido aos diversos recursos já disponíveis no Kubernetes, como monitoramento e balanceamento de carga. O Kubernetes pode replicar *containers*, provendo tolerância a faltas ao serviço de coordenação de metadados. Entretanto, o Kubernetes não tem a capacidade de manter sincronizado o estado dos *containers*. Volumes externos podem ser utilizados para persistir o estado. Porém, os volumes devem ser protegidos contra falhas. Além disso, acessos concorrentes devem ser controlados pela aplicação. O Flocker (Apêndice A, item 17) é um *software* que

[1] O termo "réplica" é usado neste capítulo para denotar *containers* que possuem o estado replicado.

pode restaurar o estado de um *container* que foi migrado para outra máquina física. Essa restauração de estado é feita com uso de volumes externos. Contudo, o Flocker tem por objetivo portabilidade e mobilidade, mas não a replicação ativa de *containers*. Ademais, o acesso a um mesmo volume por *containers* replicados deve ser controlado pela aplicação, sob pena de provocar corrupção de dados (Apêndice A, item 38).

Pode-se usar RME (Seção 2.1.4) para manter sincronizados os metadados replicados em *containers*. Algoritmos derivados do Paxos têm sido propostos para replicar estados (Seção 3.2). Há iniciativas de executar algoritmos de consenso no Kubernetes (OLIVEIRA et al., 2016), com o *software* de consenso atuando junto da aplicação. Cada *container* replicado tem que dispor do *software* de coordenação de estados e interagir com ele. Implementar RME em aplicações existentes não é uma tarefa trivial (CUI et al., 2015).

O fato do Kubernetes ser uma ferramenta de código livre possibilita sua extensão sem restrições. É possível incorporar a coordenação de metadados no Kubernetes, tornando-a transparente para o usuário (Seção 3.2.5). Na abordagem de integração, componentes podem ser modificados ou novos componentes podem ser adicionados à arquitetura. Retirar a tarefa de coordenação de metadados da aplicação e fornecê-la automaticamente oferece vantagens. Primeiro, é possível deixar a coordenação de metadados transparente para a aplicação. Ações de coordenação de metadados podem ser alocadas em uma parte do ambiente, de tal forma que seja possível apenas indicar para o ambiente que determinada aplicação precisa ter o estado replicado.

Em segundo lugar, ao fazer com que a aplicação não precise mais interagir com a ferramenta de coordenação de metadados, essa ferramenta não precisa mais estar junto da aplicação. No contexto de virtualização em *containers*, isso significa que o *container* no qual se encontra a aplicação não precisará mais hospedar a ferramenta de coordenação de metadados, ou qualquer parte de biblioteca necessária para contactar a coordenação de metadados. Consequentemente, o tamanho do *container* é reduzido, o que é um benefício proporcional à quantidade de réplicas do *container*.

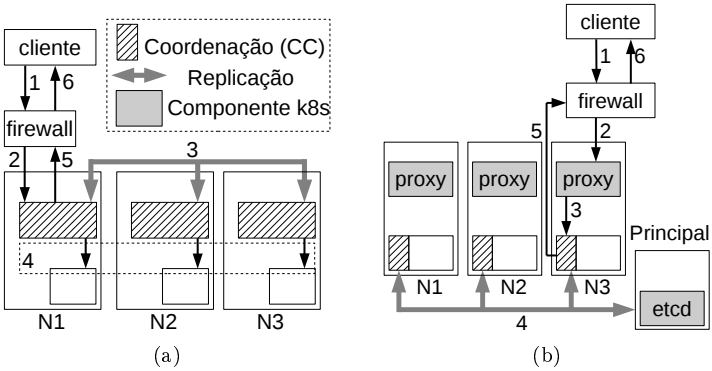
O Kubernetes foi desenvolvido para ser executado em um único provedor de nuvem. Recentemente foi criada a federação de instâncias do Kubernetes (Apêndice A, item 10), previamente denominada Ubernetes (Apêndice A, item 9). Entretanto, além da operação dessa federação ainda ser incipiente, a coordenação de metadados distribuída não é a abordagem utilizada pelos trabalhos mais próximos considerados nesta seção (GlobalFS e SCFS, Seção 3.3). A distribuição da

coordenação de estados implementada por algoritmos derivados do Paxos ainda não é recomendada em ambientes amplamente distribuídos como a WAN (AILIJIANG; CHARAPKO; DEMIRBAS, 2016). Portanto, mesmo com vistas a uma futura distribuição do Kubernetes entre provedores, esta tese considera a execução do Kubernetes em um único provedor de nuvem.

5.2 PROPOSTA DE INTEGRAÇÃO PARCIAL

A coordenação de metadados é incorporada ao Kubernetes por meio de integração (Seção 5.1). Duas alternativas são consideradas para integrar coordenação de metadados no Kubernetes (Figura 33). Apesar da segunda abordagem ser escolhida para esta tese, é relevante descrever a possibilidade a seguir. Na primeira abordagem (Figura 33(a)), um componente novo (CC) é criado para realizar a coordenação de metadados. Nesse componente é executado um protocolo de consenso já existente. O cliente envia o pedido (1), que é entregue para o CC (2) por meio do *firewall*. O protocolo de consenso ordena o pedido (3), que é executado em todos os *containers* (4). A resposta da execução (5) é encaminhada ao cliente (6), por meio de um *firewall*.

Figura 33 – Arquiteturas para integrar coordenação de metadados ao Kubernetes: integração (a) total e (b) parcial.



Fonte: próprio autor (2017).

Na segunda abordagem (Figura 33(b)), é proposto o uso de um componente já existente no Kubernetes e a criação de um novo algoritmo de consenso. O *etcd* (Seção 2.2.6), um componente já disponível

no Kubernetes, é usado na condição de memória compartilhada (MC) pelas ações de coordenação de metadados. A coordenação de metadados encontra-se nos *containers* que possuem estado replicado. O pedido é entregue pelo *proxy* ao *container* replicado (3). Antes de executar o pedido, a coordenação de metadados (CC) interage com a MC via protocolo de replicação (4). Após a definição da ordem, cada *container* replicado executa o pedido. Apenas o *container* que recebeu o pedido responde (5). A resposta é entregue ao cliente pelo *firewall* (6).

O uso do *etcd* como MC apresenta vantagens como o reuso de um componente disponível e a construção de um protocolo simples com base na MC. Ademais, ao usar a MC observa-se uma possível redução no custo de comunicação. Protocolos de consenso (Seção 2.1.4) trocam frequentemente mensagens entre os membros do grupo. Implementações de RME sobre MC podem se beneficiar com um menor número de mensagens (DETTONI et al., 2013). Membros do grupo podem apenas ler e escrever na MC, ao invés de comunicar-se com os outros membros. A redução de custos de comunicação pode ocorrer com o uso de novas tecnologias que implementam MC distribuída. Por exemplo, uma MC pode ser implementada sobre pastas compartilhadas por MV (Seção 3.2.2). Nesse caso, se mais de uma MV é alocada no mesmo *host*, a comunicação em rede para manter a MC consistente é menor.

5.2.1 Modelo de sistema

Há um conjunto $\mathcal{S}$ de *containers* replicados, dos quais um máximo $f < |\mathcal{S}|/2$ podem falhar por parada. *Containers* não se recuperam de falhas, mas novos *containers* podem ser iniciados para substituir os *containers* que falharam, reconfigurando o sistema (SCHNEIDER, 1990). Uma quantidade arbitrária (mas finita) de clientes pode acessar o sistema. *Containers* comunicam-se via MC. A MC é um repositório chave-valor acessível via operações de leitura e escrita. Ela pode notificar processos (que estejam inscritos) sobre eventos que ocorrem nos dados (por exemplo, atualização). A MC associa um identificador único a cada dado que é armazenado, e é tolerante a faltas de parada.

A interação entre clientes e *containers* é parcialmente síncrona. *Containers* replicados possuem o estado replicado por meio da execução de todos os pedidos na mesma ordem (via implementação da RME).

5.2.2 Especificação

A fim de manter o estado replicado nos *containers*, pelo menos duas propriedades precisam ser satisfeitas:

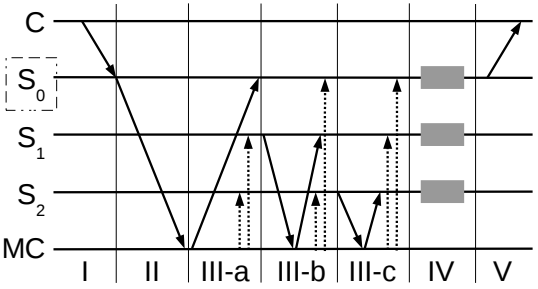
- Segurança (*Safety*): cada *container* correto executa pedidos que recebe na mesma ordem relativa.
- Vivacidade (*Liveness*): cada pedido é executado, em algum momento, por *containers* corretos.

Um requisito do protocolo de ordenação é que qualquer réplica deve ser apta a receber pedidos do cliente. Isso é importante pois, além do balanceamento de carga (entre nós) realizado pelo *firewall*, o Kubernetes realiza balanceamento de carga (entre *containers*) em nível de nó, por meio do componente *proxy* (Seção 2.3). A maioria dos protocolos de coordenação de estados (Seção 3.2) recebe pedidos apenas via réplica que exerce o papel de sequenciador fixo.

O protocolo DORADO foi criado para definir a ordem dos pedidos executados em *containers* que, além de estarem replicados, devem também possuir o estado replicado. O funcionamento do DORADO é exemplificado a seguir. Em um cenário exemplo com três réplicas ($S_{0..2}$), uma falta de parada é tolerada ($f = 1$). O cliente pode enviar o pedido para uma réplica qualquer. Se a réplica que recebe o pedido do cliente é líder (Figura 34, fase I), a réplica irá definir uma ordem e salvar o pedido ordenado na MC (fase II). A réplica líder é a única que tem autoridade para definir a ordem dos pedidos. As demais réplicas executam pedidos seguindo a ordem definida pelo líder. A MC confirma o salvamento do pedido ordenado e notifica as demais réplicas (fase III-a). Quando as outras réplicas recebem as notificações do pedido ordenado, elas aceitam o pedido. A seguir, salvam na MC seus aceites (fases III-b e III-c). Quando uma réplica coleta aceites de uma maioria de réplicas (incluindo o próprio aceite), ela pode aprender (executar) o pedido (fase IV) (Seção 2.1.4). A réplica que recebe o pedido responde ao cliente o resultado da execução (fase V).

Ao final da fase III-b, as réplicas S_0 e S_1 já possuem aceites de uma maioria de réplicas. Dessa forma, a fase III-c poderia ser ignorada. Isso significa que uma falta de parada da réplica S_2 é tolerada. Analogamente, uma parada na réplica S_1 é tolerada, ao considerar-se que a fase III-c é executada com sucesso. Dessa maneira ocorre a tolerância a faltas nas réplicas (exceto na réplica líder, caso que está descrito mais adiante).

Figura 34 – DORADO: pedido enviado ao líder.

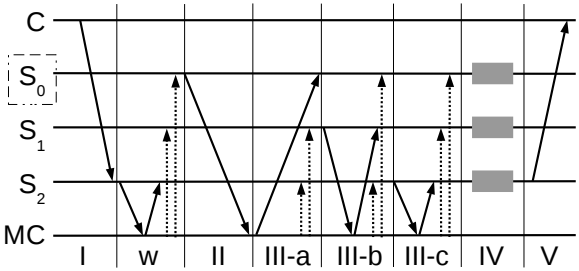


Fonte: próprio autor (2017).

Se o cliente não recebe a resposta do pedido após um período de tempo (*timeout*), ele reenvia o pedido, que provavelmente chegará a uma outra réplica[2].

Quando uma réplica não-líder recebe o pedido do cliente (Figura 35, fase I), uma ação adicional é necessária. A réplica não-líder define a ordem do pedido como sendo zero e grava este aceite na MC (fase w). A MC notifica as demais réplicas sobre esse aceite, mas apenas a réplica líder atua. A réplica líder define uma ordem para o pedido e a partir de então (fase II) o protocolo flui da mesma forma que no caso anterior (Figura 34). Da mesma forma, o cliente é respondido apenas pela réplica que recebeu o pedido (fase V).

Figura 35 – DORADO: pedido enviado para réplica seguidora.



Fonte: próprio autor (2017).

[2]O Kubernetes (Seção 2.3) faz balanceamento de carga entre *containers*.

5.2.2.1 Eleição de Líder

Protocolos que usam sequenciadores fixos precisam recorrer a temporizadores para monitorar o funcionamento do líder (Seção 2.1.4). No DORADO, uma réplica aciona um temporizador quando percebe a existência de um pedido no sistema. Uma réplica constata a existência de um pedido quando recebe-o de um cliente ou quando recebe uma notificação da MC sobre o pedido. Se um temporizador expira e um pedido permanece sem ordem (i.e., o líder não atuou sobre o pedido), a réplica salva na MC uma solicitação de eleição de líder (SEL). A MC notifica réplicas, que também salvam suas SELs na MC.

Quando uma réplica reúne SELs de uma maioria de réplicas, o líder é escolhido. O novo líder é definido como a primeira réplica que salvou uma SEL na MC. Devido a possíveis inversões de rede ou concorrência, a réplica lê da MC o conjunto completo de SELs. Assim, define-se precisamente qual réplica foi a primeira a registrar uma SEL.

5.2.2.2 Replicação no *etcd*

Esta seção esclarece aspectos da execução do DORADO como um cliente do *etcd*. O *etcd* possui a capacidade de ser distribuído, com uso do algoritmo Raft (Seção 3.2.3), tornando-o assim tolerante a faltas de parada. DORADO usa o *etcd* para persistir pedidos dos clientes, para difundir mensagens entre as réplicas e para orientar a eleição de uma réplica líder. Ao implementar a RME via DORADO, as réplicas executam todos os pedidos na mesma ordem. Nesse contexto, a ordenação provida pelo *etcd* (via Raft integrado ao mesmo) não é suficiente para definir a ordem de pedidos dos clientes no Kubernetes. Considere os seguintes fatos:

- Fato 1: o mecanismo de notificação do *etcd* não entrega mensagens usando ordem total.
- Fato 2: o identificador único[3] (UID) gerado pelo *etcd* é monotônico, mas não é garantidamente sequencial. Dados criados em outras chaves podem criar saltos no UID observado por um cliente que acompanha as alterações de valor na chave.

Quando diferentes clientes enviam pedidos simultaneamente, os UIDs associados aos pedidos poderiam ser usados para decidir a ordem dos pedidos. Entretanto, devido ao Fato 1, réplicas poderiam ser notifi-

[3] Identificador associado a valores criados sob uma chave, no contexto chave-valor.

cadas com saltos entre os UIDs recebidos. Não é possível aguardar por notificações referentes aos saltos, devido ao Fato 2 (quantas notificações devem ser esperadas?). No processo de eleição de líder do DORADO (Seção 5.2.2.1), essa questão foi resolvida realizando-se a leitura de todos os dados antes de definir o novo líder. Porém, essa estratégia é viável nas eleições apenas porque não é uma operação frequente. É ineficiente aplicar essa solução na ordenação de pedidos porque é necessário fazer a leitura de muitos dados (mensagens *propose* e *accept*).

Cabe uma nota sobre o Raft e o processo de eleição realizado pelo DORADO. O DORADO usa o UID para escolher um líder (Seção 5.2.2.1). A consistência do UID é garantida pelo Raft quando o *etcd* é replicado. Portanto, apenas durante eleições o Raft é essencial para garantir que o DORADO permaneça seguro (*safety*). O DORADO usa um líder próprio para definir a ordem de execução das requisições.

5.2.3 Algoritmos do DORADO

Diversas informações compõem uma requisição modelada pelo DORADO (Algoritmo 4). A chave gerada pelo *etcd* (linha 2) identifica unicamente a requisição. A ordem da requisição (linha 3) será definida pelo DORADO. A quantidade de réplicas ativas e o endereço de rede da réplica atual (linha 4) também são armazenados na requisição.

Algoritmo 4 Modelagem de uma requisição

```
1: type Request struct
2:     key                ▷ gerado pela memória compartilhada (MC)
3:     CI                                  ▷ instância de consenso
4:     q, IP  ▷ número de réplicas ativas e endereço IP do container
5:     reqID          ▷ identificador do pedido (provido pelo cliente)
6:     op, lv                              ▷ operação e visão do líder
7:     resultOfExecution        ▷ resultado da execução da requisição
8:     ini  ▷ instante no qual um pedido desordenado chega à réplica
9:     leci                        ▷ última instância de consenso executada
```

O cliente associa à requisição um identificador do pedido (linha 5), único sob o ponto de vista do cliente. A operação a ser executada e a visão atual do líder também constam na requisição (linha 6). Após a execução da requisição, o resultado é armazenado na mesma (linha 7). Quando uma requisição chega ao DORADO, o instante de tempo é capturado (linha 8). Essa informação é utilizada por tempo-

rizadores. Por fim, a ordem da última requisição executada (linha 9) é útil para controlar execução de requisições durante troca de visão.

Uma réplica do DORADO inicialmente obtém seu endereço de rede (Algoritmo 5, linha 2). A seguir, consulta o servidor de APIs do Kubernetes (Seção 2.3) para obter o endereço das demais réplicas (linha 3). No princípio, não existe nenhum pedido para ser ordenado ou executado (linhas 4 e 5). Uma ferramenta de grupo é usada para que uma *thread* comunique-se com outras *threads* (linha 6). Nenhuma réplica começa sendo líder (linha 7) e o sistema inicia sem um líder (linha 8). Não há mensagens de eleição e o estado inicial possui valor nulo (linha 9). O tempo de espera para atuação do líder é definido como 1 segundo (linha 10). Define-se a primeira ordem de execução de requisição disponível (linha 11) e não há requisições anteriores executadas (linha 12). Por fim, a réplica inscreve-se na memória compartilhada (MC) para receber atualizações (linha 13).

Algoritmo 5 Inicialização da Réplica (*main*)

```
1:  upon event ⟨ S_i , Init ⟩ do
2:      myIP := loadIP()                          ▷ ler IP da placa de rede
3:      replicas := loadReplicasFromKubernetes()
4:      wL := {}                  ▷ pedidos sem ordem aguardando o líder
5:      wQ := {}                          ▷ pedidos aguardando quórum
6:      bGroup := new bGroup()          ▷ comunicação entre threads
7:      IAmTheLeader := false   ▷ nenhuma réplica inicia sendo líder
8:      clv := 0                                  ▷ visão corrente (de líder)
9:      votes := {}, state := 0 ▷ mensagens de eleição e estado inicial
10:     δ := 1000ms               ▷ timeout para monitoramento do líder
11:     availableCI := 1    ▷ próxima instância de consenso disponível
12:     lastECI := 0                ▷ Nenhuma requisição foi ordenada
13:     registerInSM()        ▷ inscreve a réplica na MC (notificações)
```

Qualquer réplica do DORADO pode receber pedidos (Algoritmo 6). Ao receber um pedido, a réplica associa: i) uma ordem ao pedido, caso seja líder; ou ii) zero, caso não seja líder (linha 2). Um novo pedido é criado com as informações necessárias (linha 3). Caso a réplica não seja líder (linha 4), adiciona-se o pedido (com ordem zero) a uma lista de pedidos que aguardam associação de ordem pelo líder (linha 5). Adicionalmente, nesse caso é iniciado o monitoramento do líder (linha 6). A seguir, o pedido é salvo na MC (linha 7). O sistema aguarda até que algum pedido recebido pela réplica seja executado (linha 9). Quando o pedido recebido pela réplica é executado na referida instância/*thread*

de execução (linha 10), a réplica responde ao cliente (linha 11).

Algoritmo 6 Recebimento de pedido do cliente (*hand requests*)

```
1:  upon event ⟨ S_i , IncomingRequest | reqID, op ⟩ do
2:      if IAmTheLeader then ci := nextCI() else ci := 0
3:      nReq := ⟨ci, len(replicas), myIP, reqID, op, LView, ∅, Now()⟩
4:      if ¬ IAmTheLeader then
5:          wL := wL ∪ {nReq}          ▷ inserido na lista de pendentes
6:          checkLeaderMonitor()
7:      disseminateRequest(nReq)                  ▷ salva pedido na MC
8:      repeat
9:          executed := bGroup.receive()
10:     until executed.reqID = reqID    ▷ espera execução do pedido
11:     deliver executed.resultOfExecution to the client
```

Ao receber uma notificação da MC (Algoritmo 7), a réplica verifica se o pedido ainda não possui ordem definida pelo líder (linha 2). Nesse caso, se a réplica em questão é a réplica líder (linha 3), uma ordem é associada ao pedido (linha 4). Se a réplica não é líder, o pedido é colocado em uma lista de pedidos que aguardam manifestação do líder (linha 6) e o monitoramento do líder é iniciado (linha 7).

Algoritmo 7 Receber (da SM) notificação de pedido (*MC watcher*)

```
1:  upon event ⟨ S_i , RequestArrival | r ⟩ do   ▷ r is of Request type
2:      if r.CI=0 then
3:          if IAmTheLeader then
4:              r.CI := nextCI()                  ▷ define ordem do pedido
5:          else
6:              wL := wL ∪ r
7:              checkLeaderMonitor()
8:      else
9:          if (r.CI>lastECI) ∧ (r.IP≠myIP) ∧ r.lv=clv then)
10:             RA:=⟨r.CI,r.q,myIP,r.reqID,r.op,r.lv,∅,Now(),lastECI⟩
11:             disseminateRequest(RA)
12:             trigger ⟨ qw, NotifyRA | r⟩   ▷ notifica quórum, Alg. 8
```

Quando uma réplica recebe um pedido ordenado, ainda não executado e desconhecido (linha 9), a réplica cria uma mensagem de aceite (linha 10) do pedido. A réplica salva o aceite na MC (linha 11) e notifica o módulo de verificação de quórum sobre esse pedido (linha 12).

O verificador de quórum (Algoritmo 8) verifica a existência de suficientes notificações de aceites de pedidos tal que o pedido possa ser aprendido (e consequentemente, executado). Ao receber uma notificação de aceite de pedido (linha 1), obtém-se os aceites do referido pedido, considerando a visão atual e o identificador do pedido (linha 2). Calcula-se a quantidade de pedidos aceitos (linha 3) e o montante referente a uma maioria de réplicas (linha 4). Essas informações são comparadas a fim de verificar se o pedido foi aceito por uma maioria de réplicas (linha 5). Em caso positivo, dispara-se a execução do pedido (linha 6) e o mesmo é removido do conjunto de pedidos pendentes para execução (linha 7).

Algoritmo 8 Verifica quóruns de notific. de pedido (*quorum watcher*)

```
1: upon event ⟨ qw, NotifyRA | r ⟩ do ▷ quorum watcher, request
2:     tmpAceites := ∀x|x ∈ wQ ∧ x.view=clv ∧ x.reqID=r.reqID
3:     already := len(tmpAceites)  ▷ tmpAceites é uma variável local
4:     majority := (r.q / 2) + 1
5:     if already ≥ majority then
6:         trigger ⟨ex, NotifyExecution | r⟩      ▷ notifica exec., Alg 9
7:         wQ := wQ \ {∀x|x ∈ tmpAceites}
```

O módulo de execução de pedidos (Algoritmo 9) inicialmente cria uma lista vazia de pedidos a serem executados (linha 2). No recebimento de um pedido aprendido (linha 3), o mesmo é inserido no conjunto de pedidos a serem executados (linha 4). A chegada de um pedido aprendido pode implicar na execução de um ou mais pedidos (linha 5). O próximo pedido a ser executado é aquele que possui a menor ordem de execução (linha 6). Se esse pedido (de menor ordem) possui a ordem de execução imediatamente superior ao último pedido executado, o mesmo pode ser executado (linha 7). A execução de um pedido (linha 8) implica na retirada do mesmo da lista de pedidos a serem executados (linha 9). Após executar um pedido (linha 10), atualiza-se o índice do último pedido executado (linha 11). A execução de um pedido cuja responsabilidade de resposta é da réplica (linha 12) gera uma notificação ao módulo de recebimento de pedidos (linha 13). Repetem-se as verificações enquanto houver pedidos a serem executados (linha 14).

Seis funções auxiliares integram o código do DORADO (Algoritmo 10). A função *checkLeaderMonitor* (linha 1) inicia o monitoramento do líder (linha 3), caso esse monitoramento ainda não esteja iniciado (linha 2). O monitoramento do líder (linha 4), por sua vez, tem por objetivo verificar o estado de pedidos ainda não ordenados. Perio-

Algoritmo 9 Execução de Pedidos (*executor watcher*)

```
 1: upon event ⟨ ex , Init ⟩ do                    ▷ executado apenas uma vez
 2:     wExecution := {}
 3: upon event ⟨ ex , NotifyExecution | r ⟩ do ▷ executor, request ).
 4:     wExecution := wExecution ∪ {r}
 5:     repeat
 6:         req := r_i|∀r_i, r_j ∈ wExecution : r_i.CI ≤ r_j.CI
 7:         execute := (req.CI = lastExecCI+1)
 8:         if execute then
 9:             wExecution := wExecution \ {req}
10:             state := executeRequest(req.op, state)
11:             lastECI := req.CI
12:             if this replica must answer request to the client then
13:                 bGroup.send(req);
14:     until ¬ execute | wExecution = ⊥
```

dicamente é verificado se existe algum pedido que não foi ordenado pelo líder da visão atual, e se esse pedido está há muito tempo (*timeout*) sem ordem (linha 8). Como primeira ação, o monitor sinaliza o início de sua atuação (linha 5). Para cada pedido pendente de ordenação, é verificado (linha 8): i) se o pedido está aguardando há mais tempo que o permitido; ii) se não há eleições em progresso; e iii) se a visão do líder relativa ao pedido ainda é atual. Em caso positivo, uma eleição é solicitada (linha 9) e o monitor do líder é desativado (linha 10). O monitor do líder pode ser também desativado (linha 13) devido a não haver mais pedidos pendentes de ordenação (linha 12).

Ao iniciar eleições, uma réplica verifica se a referida solicitação de eleição de líder (SEL) ainda não foi enviada (linha 15). Em caso negativo, sinaliza-se que há eleições em andamento (linha 16). Após a confecção da mensagem de SEL (linha 17), a mensagem é disseminada entre as demais réplicas (i.e., salva na MC) (linha 18). Pedidos são armazenados na MC sob a chave *request_section* (linha 20). A cada pedido X criado, obtém-se o identificador de X gerado pela MC (linha 21) e insere-se X no conjunto de pedidos que aguardam um quórum suficiente para execução (linha 22). Votos são armazenados sob a chave *vote_section* (linha 24). Cada voto disseminado é inserido em um conjunto de votos conhecidos (linha 25). Não é necessário recuperar o identificador criado para o voto (conforme é feito para o pedido, na linha 21) porque é preciso reler todos os votos no momento de decidir

quem é o líder. A obtenção do próximo número de ordem de execução compõe-se de duas operações: obter o número de ordem atualmente disponível (linha 27) e incrementar o número de ordem atual (linha 28).

Algoritmo 10 Algoritmos auxiliares

```
 1: procedure checkLeaderMonitor
 2:     if ¬ leaderMonitorOn then
 3:         launch leaderMonitor()
 4: procedure leaderMonitor
 5:     leaderMonitorOn := true
 6:     repeat                          ▷ EiP = election in progress, δ=timeout
 7:         n := Now()
 8:         if (∃x|x ∈ wL ∧ (n − x.ini ≥ δ) ∧ ¬EiP ∧ x.lv = clv) then
 9:             launch startElections()
10:             leaderMonitorOn := false      ▷ evita eleições adicionais
11:         sleep(δ / 5)                              ▷ ciclo de verificação
12:     until wL = ⊥ | ¬ leaderMonitorOn
13:     leaderMonitor := false
14: procedure startElections
15:     if this replica didn't vote for a (clv+1) leader view then
16:         EiP := true                               ▷ eleição em andamento
17:         newVote := ⟨ clv+1, myIP, lastECI ⟩
18:         disseminateVote(newVote)
19: procedure disseminateRequest(req)
20:     createdKey := saveInSharedMemory(req, "request_section")
21:     req.key := createdKey
22:     wQ := wQ ∪ {req}
23: procedure disseminateVote(vote)
24:     saveInSharedMemory(vote, "vote_section")
25:     votes := votes ∪ {vote}
26: function nextCI
27:     current := availableCI
28:     availableCI := availableCI+1
29:     return current
```

Um módulo específico recebe notificações de eleições da MC (Algoritmo 11). O líder de uma visão anterior não participa de eleições de uma visão seguinte (linha 6). O início da eleição é demarcado pela inserção da mensagem de eleição no conjunto apropriado (linha 7). A seguir, calcula-se a quantidade que representa a maioria de réplicas no

sistema (linha 8) e a quantidade de votos acumulados (linha 9). Se há votos suficientes (linha 10), atualiza-se a visão corrente (linha 11) e todos os votos são lidos da MC (linha 12). O novo líder é definido como sendo a réplica que manifestou a primeira SEL (linha 13). A réplica atualiza sua percepção sobre liderança (linha 14). Se a réplica atual for o novo líder (linha 15), verifica a ordem do último pedido que foi executado (linha 16). Essa ordem é obtida do conjunto de réplicas que votou na eleição. O novo líder atualiza o índice de próxima instância de consenso disponível (linha 17). Pedidos pendentes de ordenação são reordenados pelo novo líder (linha 18). A despeito do papel da réplica (líder ou seguidora), a eleição é concluída (linha 19). Se a réplica recebe uma SEL mas ainda não reúne votos suficientes para definir o novo líder (linha 21), ela cria o seu voto (linha 22) e salva-o na MC (linha 23).

Algoritmo 11 Notificação sobre eleição

```
1:  type ElectionMessage struct
2:      newView                                  ▷ proposição de nova visão
3:      IP                                       ▷ IP do emissor do voto
4:      lastECI
5:  upon event ⟨ S_i , Election | em ⟩ do        ▷ election message
6:      if ¬ IAmTheLeader ∧ em.newView=(clv+1) then
7:          votes := votes ∪ {em}
8:          majority := (len(replicas)/2) + 1 )
9:          already := len(∀x | x ∈ votes ∧ x.newView=(clv+1)) )
10:         if already ≥ majority then           ▷ quórum suficiente?
11:             clv := clv + 1
12:             loadAllElectionMessagesFromSM()
13:             leaderIP:=r_i|∀r_j ∈votes:(r_i.view=clv ∧ r_i.key< r_j.key)
14:             IAmTheLeader:=(myIP=leaderIP)
15:             if IAmTheLeader then
16:                 h:=r_i.leci|∀r_j ∈votes:(r_i.view=clv∧r_i.leci> r_j.leci)
17:                 availableCI := h+1
18:                 ReorderPendingRequests()     ▷ reordenar wL
19:             EiP := false                     ▷ fim de eleição
20:         else
21:             if ∄x|x ∈ votes ∧ x.IP=myIP ∧ x.view=(clv+1) then
22:                 newVote := ⟨ clv+1, myIP, lastECI ⟩
23:                 disseminateVote(newVote)
```

5.2.4 Provas

Nesta seção é mostrado que o protocolo DORADO é correto, em termos de seus algoritmos (Seção 5.2.3) e suas propriedades de Segurança (*Safety*) e Vivacidade (*Liveness*) (Seção 5.2.1).

Nas provas a seguir, o termo Definição é usado para declarações baseadas em literatura ou fatos estabelecidos. Axiomas são relacionados a declarações relativas ao protocolo DORADO.

As seguintes definições são relacionada à memória compartilhada:

Definição 1. *A memória compartilhada (MC) é capaz de enviar notificações sobre eventos em dados (Seção 5.2.1).*

Definição 2. *Um dado salvo na MC é imediatamente confirmado para o escritor e em algum momento (parcialmente síncrono) é notificado para todas os assinantes de notificações (Seção 5.2.1).*

Sobre tolerância a faltas (Seção 2.1.3), as seguintes definições são apresentadas:

Definição 3. *No sistema devem existir 2f+1 réplicas para que sejam toleradas f faltas de parada. Nesse contexto, pelo menos f+1 réplicas no sistema irão comportar-se corretamente.*

Definição 4. *Em um sistema tolerante a faltas de parada com 2f+1 réplicas, a maioria de réplicas é equivalente numericamente a f+1 réplicas.*

Vivacidade e Segurança: os Axiomas 1, 2 e 3, a seguir, se referem ao protocolo DORADO.

Axioma 1. *O protocolo DORADO utiliza 2f+1 réplicas (Seção 5.2.1).*

Axioma 2. *Todas as réplicas do DORADO estão inscritas na MC (Algoritmo 5, linha 13).*

Axioma 3. *No DORADO, uma réplica recebe o pedido do cliente (Algoritmo 6, linha 1) e a seguir armazena-o na MC (Algoritmo 6, linha 7).*

Considerando as proposições apresentadas, pode-se elaborar o seguinte lema:

Lema 1. *Um pedido enviado ao sistema será em algum momento conhecido por uma maioria de réplicas.*

Prova: O pedido enviado pelo cliente será armazenado na MC, conforme Axioma 3. De acordo com as Definições 1 e 2 e os Axiomas 1 e 2, quando uma réplica grava um dado na MC, as demais réplicas ($2f$ réplicas) são notificadas sobre o pedido. Conforme a Definição 3, em algum momento pelo menos f réplicas são notificadas do pedido. Dessa maneira, em algum momento uma maioria (Definição 4) de réplicas (incluindo a réplica que recebeu o pedido do cliente) conhecerá o pedido que foi enviado pelo cliente ao sistema. □

Do Lema 1 pode-se obter o seguinte corolário:

Corolário 1. *Uma informação gravada na MC será em algum momento conhecida por uma maioria de réplicas.*

A seguir, considere os seguintes axiomas:

Axioma 4. *Quando uma réplica conhece um pedido que ainda não aprendeu, ela aceita o pedido e salva o seu aceite uma única vez na MC.*

Axioma 5. *Quando uma réplica possui $f + 1$ aceites de pedidos (incluindo o seu próprio aceite), ela aprende o pedido.*

Considerando as proposições apresentadas, pode-se enunciar o seguinte lema:

Lema 2. *Um pedido proposto vai ser aprendido em algum momento por pelo menos uma maioria de réplicas.*

Prova: de acordo com o Lema 1, um pedido proposto será conhecido por uma maioria de réplicas. No momento em que uma réplica conhece um pedido que ainda não aprendeu, ela difundirá seu aceite para as outras réplicas (Axioma 4). Cada aceite difundido será, em algum momento, conhecido por uma maioria de réplicas (Corolário 1). Cada réplica desta maioria por sua vez difundirá também o seu aceite. Quando pelo menos as $f+1$ réplicas integrantes da maioria difundirem seus aceites, cada réplica que receber estes aceites poderá aprender o pedido (Axioma 5). Quando cada réplica integrante da maioria aprender o pedido, a maioria das réplicas terá aprendido o pedido. □

O Lema 2 prova a vivacidade do DORADO. Depois que um pedido é aprendido, ele pode ser executado. Apenas a réplica que recebeu o pedido do cliente responderá ao cliente (Algoritmo 9, linha 12). Mesmo em casos nos quais réplicas não-líder falhem por parada, o DORADO prossegue pois apenas uma maioria de réplicas é suficiente para garantir o avanço do protocolo.

Segurança: o Axioma a seguir trata de protocolos que utilizam sequenciadores fixos.

Axioma 6. *Em protocolos que utilizam sequenciadores fixos, o líder define uma sequência única de ordem dos pedidos (Seção 2.1.4).*

Pode-se então declarar o seguinte lema:

Lema 3. *Todo pedido aprendido será executado em todas as réplicas corretas na mesma sequência.*

Prova: após uma réplica aprender um pedido (Lema 2), a réplica pode executar o pedido. Todo pedido é ordenado por um único líder (Axioma 6). Portanto, todas as réplicas corretas executarão os pedidos em uma mesma ordem. □

A segurança do DORADO é provada pelo Lema 3, pois há sempre uma maioria de réplicas envolvidas na execução de pedidos (Algoritmo 8, linha 5).

Segurança na escolha do líder: será mostrado que as propriedades de segurança e vivacidade permanecem válidas durante um processo de eleição no DORADO. O axioma a seguir estabelece a ação que provê determinismo na escolha do líder.

Axioma 7. *Para decidir o novo líder, todos os votos de eleição são carregados da MC (Algoritmo 11, linha 12), sendo possível verificar precisamente qual foi a primeira réplica que votou (Algoritmo 11, linha 13.*

Todas as réplicas escolherão o mesmo líder conforme registra o Axioma 7. Isto garante a segurança do processo de eleição, visto que todas as réplicas escolherão o mesmo líder.

Vivacidade do processo de eleição: a partir do Corolário 1, pode-se declarar o axioma que segue.

Axioma 8. *Quando uma réplica inicia uma eleição, em algum momento uma maioria de réplicas estará ciente dessa eleição.*

O próximo Axioma deriva do momento de decisão sobre quem será o novo líder:

Axioma 9. *Uma réplica pode decidir quem é o novo líder após reunir votos de eleição de uma maioria de réplicas (Algoritmo 11, linha 10).*

Diante dos axiomas mencionados, pode-se declarar o seguinte lema:

Lema 4. *Toda eleição iniciada por uma réplica irá ser concluída.*

Prova: a difusão dos votos é garantida pelo Axioma 8. Após a difusão de votos em quantidade suficiente, a réplica poderá escolher o novo líder, conforme Axioma 9. □

O Lema 4 garante a vivacidade do processo de eleição.

Segurança do estado durante uma troca de líder: a definição a seguir advém de uma propriedade inerente dos sistemas de votação por quórum.

Axioma 10. *A utilização de quóruns tais que suas interseções contenham pelo menos uma maioria de réplicas garante a obtenção do valor mais recente mantido por um grupo de réplicas (Seção 2.1.5).*

O valor mantido pelas réplicas, mencionado no Axioma 10, é o estado das réplicas. Mais especificamente, refere-se ao estado que foi modificado por pedidos que foram aprendidos no sistema.

A eleição de líder no DORADO considera interseções de quóruns, conforme declara o axioma a seguir.

Axioma 11. *O novo líder no DORADO reúne os números de ordem de pedidos existentes na maioria de réplicas votantes (Algoritmo 11, linha 16) para definir qual será o próximo número de ordem disponível (Algoritmo 11, linha 17).*

A interseção de uma maioria de réplicas garante a obtenção do valor estável mais recente mantido pelo grupo de réplicas. Entretanto, com o intuito de selecionar uma maioria de forma determinística e eliminar a consideração de múltiplas maiorias, escolhe-se a maioria de réplicas seguindo a ordem das réplicas votantes. Ou seja, as $f+1$ réplicas votantes comporão a maioria da qual será obtido o maior número de ordem. Esse valor será utilizado pelo próximo líder como ponto de partida da ordenação de novos pedidos ou reordenação de pedidos ainda não aprendidos (ordenados por um líder anterior). O seguinte axioma formaliza esse comportamento:

Axioma 12. *A primeira maioria de réplicas é composta pelas primeiras $f+1$ réplicas que solicitaram eleição.*

Diante das informações mencionadas, é possível declarar o lema a seguir.

Lema 5. *O novo número de ordem estabelecido pelo novo líder é maior que qualquer número de ordem que tenha sido aprendido pela primeira maioria de réplicas.*

Prova: O último número de ordem utilizado é conhecido por um grupo majoritário de réplicas (Axiomas 10 e 11). Adicionalmente, o número de ordem a ser utilizado pelo novo líder não dependerá de qual réplica assuma esse papel (Axioma 12). □

O Lema 5 garante a segurança do estado das réplicas após uma troca de líder.

LIMITAÇÕES: o fato de usar a MC como canal de comunicação entre réplicas inviabiliza o uso de mensagens *batch* que poderiam reduzir o número de mensagens enviadas na rede. Por exemplo, em vez de enviar 2 notificações para uma réplica, em casos de alta contenção a MC poderia enviar apenas uma mensagem contendo as duas notificações. É possível que a MC seja configurável para realizar tal aprimoramento.

5.3 CONCLUSÕES DO CAPÍTULO

Este capítulo apresentou o DORADO, uma solução para coordenar metadados no Kubernetes. DORADO faz uso de uma memória compartilhada, implementada por um componente existente no Kubernetes (o *etcd*). Caracteriza-se, assim, como um protocolo leve e de projeto simples. O próximo capítulo detalha o sistema FITS, que utiliza em sua arquitetura o sistema de armazenamento RafeStore e o sistema de coordenação de metadados DORADO.

6 SISTEMA INTEGRADO PARA ARMAZENAMENTO DE DADOS EM MÚLTIPLAS NUVENS

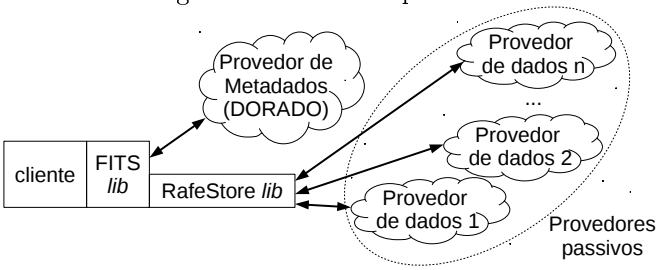
Never think you've seen the last of everything.
Eudora Welty

Este capítulo descreve o sistema FITS (*e**F**fect**I**ve s**T**orage for the ma**S**ses*), anunciado na Seção 3.3.3. São descritos arquitetura, protocolo e algoritmos do FITS.

6.1 COMPOSIÇÃO DO ARMAZENAMENTO DE DADOS COM A COORDENAÇÃO DE METADADOS

O sistema FITS orquestra as operações de coordenação de metadados e armazenamento de dados (Figura 36). O FITS usa os sistemas RafeStore e DORADO (Capítulos 4 e 5) de forma modular, assim como outras soluções (SCFS, Seção 3.3.1, e GlobalFS, Seção 3.3.2). Os dados são armazenados em múltiplos provedores e a coordenação de metadados ocorre em um único provedor. O cliente utiliza uma biblioteca[1] (FITS *lib*) para interagir com os provedores de dados e metadados[2].

Figura 36 – FITS: arquitetura.



Fonte: próprio autor (2017).

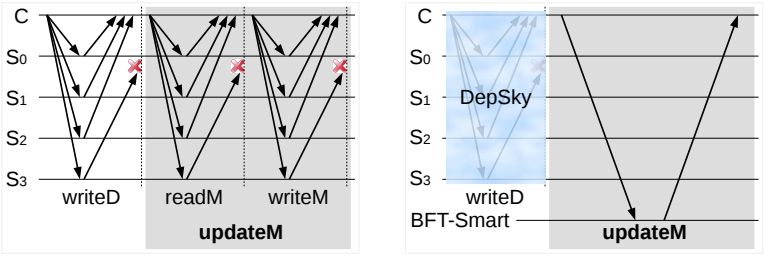
A solução de armazenamento de dados desta tese manipula dados de conteúdo independente (Seção 4.1.1). A escrita desse tipo de dado utiliza três fases de quóruns, operando em modo otimista (Figura 37(a)). Na primeira fase, os dados são escritos (writeD). A seguir,

[1] O SCFS usa um agente junto ao cliente, ao invés de biblioteca. Porém, o SCFS é um sistema de arquivos, natureza diferente do FITS.

[2] O DORADO tem operação transparente; o FITS e o RafeStore usam bibliotecas.

os metadados são lidos (readM). Por fim, os metadados são atualizados (writeM). A composição do armazenamento de dados e da coordenação de metadados requer que as fases de leitura e escrita de metadados tornem-se uma única fase de atualização de metadados.

Figura 37 – SCFS: escrita de dados de conteúdo independente: (a) quóruns isolados; (b) junção de duas fases: etapa updateM.



(a) (b)

Fonte: próprio autor (2017).

Para uma melhor compreensão, inicialmente será descrito como é possível fazer uso dos sistemas DepSky (Seção 3.1.3) e BFT-SMaRt (Seção 2.1.4) para realizar a escrita de dados de conteúdo independente (Figura 37(b)). A escrita de dados é realizada por um quórum de escrita do DepSky[3], enquanto a atualização de metadados é executada por uma solicitação ao BFT-SMaRt.

No FITS, a composição dos sistemas RafeStore e DORADO atua de forma semelhante[4] (Figura 38(a)) à composição realizada no SCFS (Seção 3.3.1). É relevante destacar a interseção de fases utilizada pelo RafeStore (Figura 38(b)). Não é possível[5] que o RafeStore atue no espaço de execução do DORADO (o círculo vermelho indica chegada de resposta do RafeStore durante execução do DORADO). Em outras palavras, não é possível que dentro do DORADO ocorra a verificação sobre a chegada de respostas enviadas pelo RafeStore.

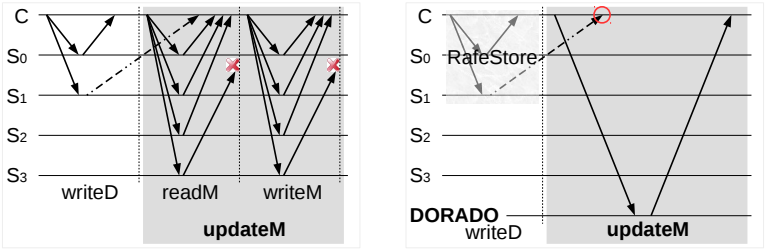
Deseja-se manter os benefícios proporcionados pela antecipação de pedidos ao acoplar o RafeStore e o DORADO. Assim, uma adapta-

[3] Na verdade, o DepSky não realiza escritas cegas. Entretanto, a consideração é útil para fins de comparação com a proposta.

[4] O protocolo DORADO tolera apenas faltas de parada. A Figura 38 considera quatro servidores na operação do DORADO apenas para que a analogia com a composição DepSky + BFT-SMaRt seja mais direta.

[5] A impossibilidade apresentada aqui refere-se ao projeto do RafeStore nesta tese.

Figura 38 – FITS: escrita de dados de conteúdo independente. (a) Quóruns isolados; (b) junção de duas fases: fase updateM.



(a) (b)

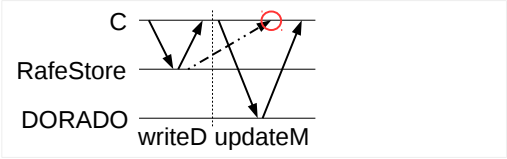
Fonte: próprio autor (2017).

ção foi realizada de maneira diferente da solução que faz uso do DepSky e do BFT-SMaRt. A próxima seção apresenta esse acoplamento.

6.2 PROPOSTA DE ACOPLAMENTO

Para realizar a composição do RafeStore com o DORADO, o FITS considera cenários otimistas e pessimistas. O primeiro caso é otimista (Figura 39): a escrita de dados é concluída antes do fim da atualização de metadados (o círculo vermelho na figura indica o término da escrita de dados do segundo provedor). Nesse cenário, os ganhos de redução de custos e latência fornecidos pelo RafeStore são mantidos integralmente.

Figura 39 – FITS: escrita com sucesso.

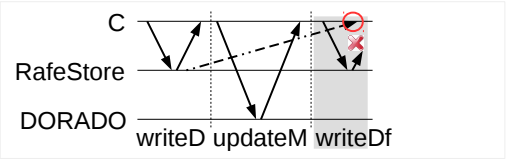


Fonte: próprio autor (2017).

Um segundo caso (Figura 40) conclui a atualização de metadados sem que a escrita de dados esteja completamente finalizada. É

necessário então solicitar que uma réplica adicional armazene o dado[6]. Entretanto, antes de a réplica adicional confirmar a escrita, chega a resposta da réplica inicialmente contactada. Dessa maneira, cancela-se a solicitação (um "x" representa o cancelamento) para a réplica adicional, pois os metadados que foram atualizados continuam válidos.

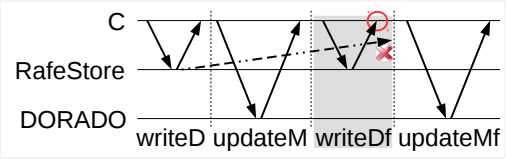
Figura 40 – FITS: escrita contactando provedor extra de dados.



Fonte: próprio autor (2017).

Em um terceiro cenário (Figura 41), a atualização de metadados é concluída e a escrita de dados não é completada. Uma réplica adicional é contactada, e a resposta da réplica adicional chega antes da resposta da réplica que foi inicialmente contactada. Nesse caso, é cancelado o pedido feito à primeira réplica, e os metadados são atualizados para informar em quais réplicas estão gravados os dados.

Figura 41 – FITS: escrita contactando provedor extra de metadados.



Fonte: próprio autor (2017).

A operação de escrita no FITS é detalhada em passos a seguir (Algoritmo 12). Uma variável global controla o número de provedores de dados que confirmaram a gravação de dados (linha 2). Inicialmente, é gerado um identificador único para o dado (linha 3). A seguir, um quórum de $f + 1$ provedores de dados é acionado (RafeStore) para armazenar os dados (linha 4). O sistema aguarda até que pelo menos um provedor de dados confirme a gravação do dado (linha 5). O serviço de metadados (DORADO) é acionado para registrar os identificadores dos provedores que armazenam os dados (linha 6). Após a atualização de

[6] Considera-se neste exemplo o caso onde apenas uma falta é tolerada (f=1).

dados, calcula-se quantas gravações de dados não foram confirmadas (linha 7). Se todas as gravações de dados solicitadas foram confirmadas, a gravação de dados é concluída. Entretanto, se houver gravações de dados pendentes (linha 8), réplicas adicionais são contactadas. Uma nova solicitação de escrita de dados é feita (linha 9), e o sistema aguarda até que existam pelo menos $f+1$ confirmações de provedores de dados (linha 10). Após a confirmação da gravação de dados em uma quantidade mínima de provedores, é verificado se os metadados devem ser alterados (linha 11). Em caso positivo, os metadados são atualizados com os identificadores dos provedores que confirmaram a gravação de dados (linha 12).

Algoritmo 12 FITS: escrita de dados de conteúdo independente

```
 1: procedure WRITE(key, value)
 2:     Global c := 0          ▷ confirmações válidas de escrita de dado
 3:     uid := UUID + "_" + hash(value)
 4:     wQuorum(uid, value, ID of f + 1 data providers)
 5:     wait until at least one answer arrives from data providers
 6:     updateMetadata(uid, ID of f + 1 metadata providers)
 7:     missingAnswers := (f + 1) - c              ▷ ver c em wQuorum
 8:     if missingAnwers > 0 then
 9:         wQuorum(uid, value, ID of data providers not contacted)
10:         wait until exists (f + 1) confirmed writes from data prov.
11:         if metadata changed then
12:             updateMetadata(uid, ID of servers that answered)

13: procedure WQUORUM(key, value, setOfServers)
14:     parallel for each S_i in setOfServers do
15:         answer := dataProvider[S_i].put(key, value)
16:         if answer = "ok" then
17:             c := c + 1
```

A leitura de dados (Algoritmo 13) começa com a obtenção dos metadados (linha 2). Se nenhuma informação for encontrada (linha 3), a leitura é encerrada. Caso contrário, todos os provedores de dados (linha 5) são acionados para realizar a leitura dos dados (linha 6). A cada resposta obtida, verifica-se se o dado está íntegro (linha 7). Quando a primeira resposta correta for obtida, cancela-se os pedidos pendentes (linha 8), a resposta obtida é retornada e a leitura é encerrada.

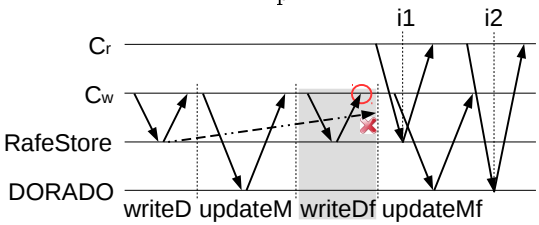
Há uma situação específica que requer uma leitura de metadados adicional (linha 9). Considere um cliente escritor C_w, cuja escrita

Algoritmo 13 FITS: leitura de dados de conteúdo independente

```
1:  procedure READ(key)
2:      m := readMetadata(key)
3:      if m == ∅ then return ∅
4:      repeat
5:          parallel for each S_i in m.IDs do
6:              answer := dataProvider[S_i].read(key)
7:              if m.hash == hash(answer) then
8:                  cancel pending requests return answer
9:          m := readMetadata(key)
10:     until true
```

ocorre em um cenário pessimista (conforme Figura 41).

Figura 42 – FITS: cenário específico de leitura de dados.



Fonte: próprio autor (2017).

No momento em que C_w (Figura 42) está atualizando os metadados pela segunda vez (fase updateMf), um outro cliente leitor C_r solicita a leitura de dados (instante de tempo i1). É possível que C_r não consiga acessar o único provedor de dados que confirmou os dados (Algoritmo 12, linha 5). Temporizadores ou corrupção de dados podem provocar uma falta nesse primeiro provedor de dados. O segundo provedor de dados informado pelos metadados está desatualizado, o que pode levar o cliente leitor a ficar sem acesso ao dado. Assim, é necessário (Algoritmo 13, linha 9) que o cliente releia os metadados (instante i2). O cliente poderá, então, contactar os $f + 1$ provedores de dados disponíveis. Essa questão é mencionada no Hybris (Seção 3.1.8), embora ele prefira utilizar notificadores (Seção 2.2.2) para informar modificações de metadados ao cliente. Leituras adicionais de metadados, durante operações de leitura, também são previstas no MDStore (Seção 3.1.7).

6.3 CONCLUSÕES DO CAPÍTULO

Este capítulo detalhou o sistema FITS, em termos de arquitetura, protocolo e algoritmos. De maneira análoga a outras soluções arquiteturais (exemplo: SCFS, Seção 3.3.1), dois sistemas específicos atuam em camadas com funcionalidades distintas: o armazenamento de dados e a coordenação de metadados. Assim, consegue-se obter os benefícios proporcionados pelas duas soluções componentes: eficiência no armazenamento de dados provida pelo RafeStore e coordenação de metadados transparente provida pelo DORADO. O próximo capítulo avalia individualmente as soluções integrantes do sistema FITS.

7 AVALIAÇÃO

Sempre parece impossível, até que esteja feito.
Nelson Mandela

Este capítulo apresenta resultados experimentais de duas propostas apresentadas nesta tese. A técnica de antecipação de pedidos (Seção 4.1) é aplicada no sistema RafeStore e é avaliada na Seção 7.2. A integração de coordenação de metadados no Kubernetes (Seção 5.2), representada pelo protocolo DORADO, é avaliada na Seção 7.3.

7.1 AMBIENTE DE AVALIAÇÃO

Para realizar a avaliação experimental, foram utilizados os ambientes *Cluster* e *Nuvens*, descritos a seguir.

- *Cluster*: quatro computadores Intel i7 3.5GHz, QuadCore, cache L3 8MB, 12GB RAM, 1TB HD 7200 RPM. Esses computadores estão localizados em Florianópolis, Santa Catarina, Brasil. Uma rede local (LAN) 10/100MBits conecta os computadores. Múltiplos sistemas operacionais foram instalados nesses computadores, sendo dois utilizados nas avaliações deste capítulo:
  - *Debian* 7.4 Wheeze, 64 bits, kernel 3.2.54-2, com JVM Oracle JDK 1.7.
  - *Ubuntu* Server 14.04.3 64 bits, kernel 3.19.0-42.

  O *Debian* foi utilizado na avaliação do RafeStore (implementado em Java) e o *Ubuntu* foi usado na avaliação do DORADO (implementado em Go). Um quinto computador atuou como cliente na avaliação da antecipação de pedidos, o qual também foi conectado na mesma LAN. Esse computador é um Intel i7 3.07GHz, QuadCore, cache L3 8MB, 16GB RAM, 2TB HD com 5400RPM.
- *Nuvens*: serviços de armazenamento em nuvem (ambiente WAN) disponíveis nos provedores HP Helion, Microsoft Azure e Amazon.

7.2 AVALIAÇÃO DO RAFESTORE

7.2.1 Considerações iniciais

O protocolo RafeStore (Capítulo 4) foi projetado para apresentar desempenho superior a protocolos que usam quóruns puros, como o DepSKy (Seção 3.1.3). Porém, para que isso aconteça, é preciso que as condições de latência (Definição 1, Seção 4.1.1) sejam favoráveis. Assim, é possível formular a primeira hipótese a ser observada nesta avaliação:

Hipótese 1. *Se a latência dos provedores de dados obederecem à Definição 1 (Seção 4.1.1), o protocolo RafeStore obterá desempenho superior a protocolos que usam quóruns Bizantinos, como o DepSky.*

Para avaliar a Hipótese 1, dois experimentos foram conduzidos. O primeiro foi realizado no ambiente *Cluster*. O segundo foi executado no ambiente *Nuvens*. Em ambos cenários, apresentados a seguir, os resultados confirmam a hipótese.

7.2.2 Experimento no *Cluster*

O experimento foi realizado no *Cluster*, com uso do sistema operacional Debian. As latências dos provedores foram configuradas com os valores 0, 161, 308 e 181 (milisegundos), respectivamente, para os provedores $p_0 ... p_3$. Esses valores foram obtidos da literatura (TERRY et al., 2013) e representam latências de um cliente na China e provedores no oeste dos Estados Unidos, na Inglaterra e na Índia, respectivamente. As latências foram associadas aos provedores de modo a obedecer a Definição 1 (Seção 4.1.1). Para simular as latências, foi utilizado o emulador de WAN chamado *netem* (Apêndice A, item 61), executado via comando *tc*.

Todos os provedores armazenam metadados, e apenas os provedores p_0 e p_1 armazenam dados. Em caso de faltas, o servidor p_2 também deve armazenar dados. O tamanho do dado considerado foi de 1KB até 1MB, variando o tamanho do arquivo por um fator multiplicativo de 4. Essa faixa de tamanho cobre desde uma postagem em um fórum até uma fotografia digital. Os pedidos foram embaralhados considerando os tamanhos diferentes de dados e os protocolos avaliados (descritos a seguir), a fim de distribuir eventuais perturbações. Para cada tamanho de dado foram realizadas 20 replicações do teste.

O RafeStore foi implementado em Java e está disponível (Apêndice A, item 14). Para melhor comparação, os protocolos do DepSky (Seção 3.1.3) foram reimplementados na mesma base de código do RafeStore. O DepSky-A realiza replicação total, enquanto o DepSky-CA usa *erasure codes* (Seção 2.2.4) para reduzir a quantidade de dados armazenados em cada provedor. A biblioteca JEC (Apêndice A, item 31) foi utilizada para a execução dos *erasure codes*. O DepSky tolera faltas bizantinas, contactando $3f + 1$ provedores de dados, garantindo que $2f + 1$ provedores armazenarão dados e metadados. RafeStore contacta apenas $f + 1$ provedores para armazenar dados, e contacta f provedores adicionais apenas quando a Definição 1 (Seção 4.1.1) não é atendida. Privacidade não foi implementada (o DepSky-CA utiliza criptografia pra prover privacidade ao dado). De qualquer maneira, autores do DepSky afirmam que os custos da criptografia e do *erasure code* são baixos para o DepSky-CA (5% para leitura e 10% para escrita) (BESSANI et al., 2013).

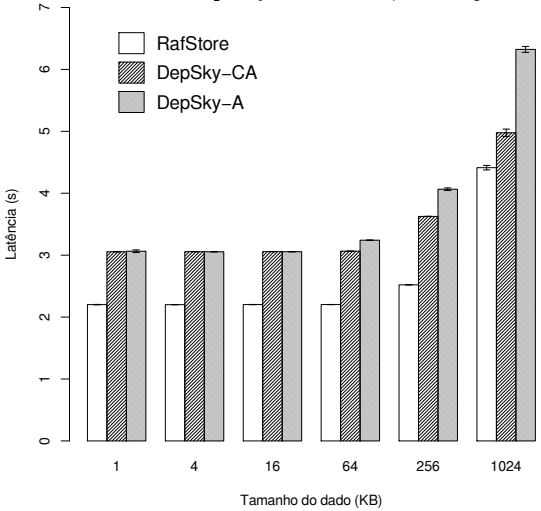
7.2.3 Resultados no *Cluster*

RafeStore apresentou desempenho melhor do que os protocolos do DepSky, para todos os tamanhos de dado considerados (Figura 43). Portanto, a Hipótese 1 é verdadeira. A vantagem do RafeStore sobre o protocolo DepSkyCA diminui quando o dado aumenta de 256KB para 1MB. Também nesse intervalo de tamanho de dado, o DepSky-CA aumenta sua vantagem em relação ao protocolo DepSky-A. Esses resultados indicam que o RafeStore talvez não se comporte bem (isto é, a Hipótese 1 não seja verdadeira) com tamanhos de dado maiores do que os tamanhos considerados nesse experimento.

É possível analisar os passos do RafeStore e observar o esforço despendido em cada etapa. Foi marcado o tempo despendido por etapa durante cada execução do protocolo. A operação de escrita no protótipo do RafeStore é demarcada pelas seguintes etapas:

- *genId*: geração do identificador de dados.
- *writeDo*: escrita de dados no primeiro provedor de dados.
- *readM*: leitura de metadados.
- *crucial*: recebimento da resposta dos provedores de dados (exceto o primeiro).
- *writeDf*: ativação de provedores adicionais.
- *writeM*: escrita de metadados.
- *shutdown*: procedimentos de encerramento.

Figura 43 – RafeStore e DepSky: latências, execução no *Cluster*.



Fonte: próprio autor (2017).

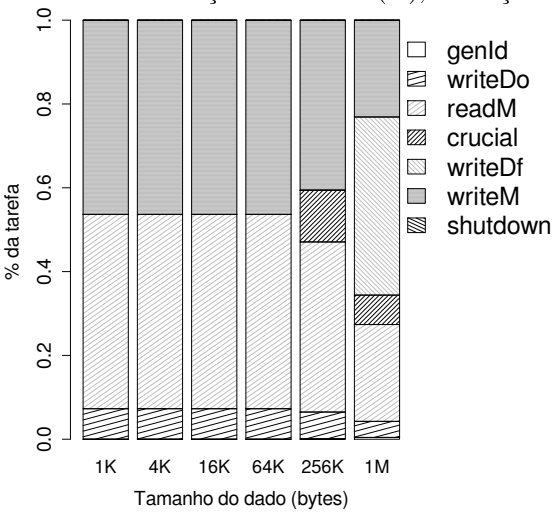
Na maioria dos tamanhos de dados avaliados (Figura 44), predominam as atividades de escrita de dados no primeiro provedor, leitura de metadados e escrita de metadados. As demais atividades despendem um tempo insignificante. Para dados a partir de 256KB, as tarefas *crucial* e *writeDf* foram evidenciadas. Apesar de o RafeStore apresentar menor latência em comparação aos demais protocolos (Figura 43) mesmo nos maiores tamanhos de arquivo, recursos não foram economizados pois o provedor de dados adicional foi contactado nesses casos (dados maiores ou iguais a 256KB).

O comportamento de etapas ocorreu de forma muito semelhante no protocolos DepSky-A e DepSky-CA. Por exemplo, no DepSky-A (Figura 45) a etapa de escrita de dados seguiu a proporção do tamanho de dado, predominando a partir de 16KB. Esse comportamento é ideal, pois quanto maior o dado, maior o tempo para gravá-lo nos provedores.

7.2.4 Novas hipóteses

Considerando os resultados obtidos no experimento executado no *Cluster* (Seção 7.2.3), mais hipóteses foram criadas e um novo experimento foi projetado para ser executado em Nuvens.

Figura 44 – RafeStore: atuação de tarefas (%), execução no *Cluster*.

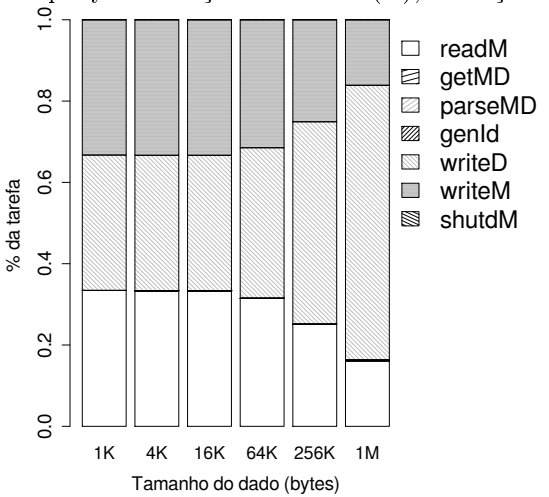


Fonte: próprio autor (2017).

A operação de escrita do RafeStore (Figura 28) tem uma fase antecipada que depende da resposta do provedor de dados mais rápido. No cenário de Nuvens, provedores de armazenamento em nuvem costumam ter latências da ordem de centenas de milisegundos. Quando o cliente dispõe de um dispositivo local ou um provedor privado para armazenar dados, pode-se supor que esse armazenamento local é o provedor mais rápido, dentre os demais. Isso pode beneficiar o RafeStore, acelerando a transição da fase *writeDo* para a fase *readM*, caso este armazenamento local seja usado pra armazenar dados. Além disso, armazenar dados em provedores de nuvens requer, além da disponibilidade do provedor, bom funcionamento da rede que leva ao provedor. Portanto, um dispositivo de armazenamento próximo ao cliente pode favorecer a disponibilidade. Para armazenar dados localmente, é preciso que o espaço disponível seja suficiente. No caso de armazenamentos locais mais limitados, pode-se salvar apenas metadados. Nesse contexto, pode-se declarar a segunda hipótese:

Hipótese 2. *Se um provedor de armazenamento local for utilizado para armazenar metadados, ou dados e metadados, o RafeStore pode apresentar um desempenho acentuadamente melhor em comparação a protocolos que usam quóruns Bizantinos.*

Figura 45 – DepSky-A: atuação de tarefas (%), execução no *Cluster*.



Fonte: próprio autor (2017).

As latências dos provedores podem ser organizadas de forma a obedecer a Definição 1 (Seção 4.1.1). Entretanto, mesmo quando a Definição 1 for quebrada, é possível que o RafeStore apresente desempenho similar aos quóruns tradicionais. Essa possibilidade deve-se ao fato de o RafeStore utilizar menos provedores de dados (de maneira similar ao MDStore, Seção 3.1.7) do que os protocolos do DepSky. Portanto, a terceira hipótese é declarada a seguir:

Hipótese 3. *Mesmo em configurações nas quais as latências dos provedores não obedeçam à Definição 1, RafeStore pode apresentar desempenho similar aos protocolos que usam quóruns Bizantinos.*

Consideradas as novas hipóteses (2 e 3), passa-se agora à realização de um experimento (Seção 7.2.5) para testar essas hipóteses.

7.2.5 Experimento em *Cluster* e Nuvens

Um segundo experimento foi conduzido para avaliar as Hipóteses 2 e 3. A Hipótese 1 é reavaliada a seguir, a fim de ratificar os resultados obtidos no *Cluster*. Nesta avaliação, é utilizado um computador disponível no ambiente *Cluster* e provedores de armazenamento em nuvem disponíveis no ambiente Nuvens (Seção 7.1).

Tabela 3 – Características dos provedores usados no experimento em *Cluster* e Nuvens.

id	Provedor	Região	RTT
p0	LOCAL	próximo ao cliente	5ms
p1	AZURE	oeste da Europa (Holanda)	315ms
p2	HP	leste dos EUA (Distrito de Columbia)	750ms
p3	AMAZON	oeste dos EUA (Oregon)	1200ms

Fonte: próprio autor (2017).

Os provedores foram rotulados para melhor referência (Tabela 3). LOCAL refere-se ao computador disponível no *cluster*. No provedor LOCAL foi instalado o banco de dados chave-valor Redis (Apêndice A, item 34). O serviço S3, oferecido pelo provedor Amazon, foi denominado como AMAZON. O armazenamento oferecido pelo provedor da Microsoft foi denominado como AZURE. O serviço de armazenamento Helion, da HP, foi rotulado como HP. Todos os provedores em nuvem possuem interface de armazenamento no formato de chave-valor. O cliente foi executado em um computador do *cluster* (um computador diferente daquele que hospeda o provedor LOCAL).

As latências de cada provedor foram obtidas com o comando *ping*. O valor de cada latência foi verificada algumas vezes para assegurar a ordem de latência entre provedores. Em testes preliminares, o provedor AMAZON possuía latência menor do que o provedor HP. Entretanto, passados 10 dias, o provedor HP apresentou latência menor do que o provedor AMAZON. As latências consideradas (Tabela 3) foram medidas segundos antes do início do experimento. Durante o experimento a relação entre as latências não mudou. O experimento foi realizado em fevereiro de 2015.

Para este experimento foram assumidas as mesmas condições do primeiro experimento (Seção 7.2.2). Entretanto, o tamanho do dado foi variado de 1KB até 16MB, por um fator multiplicativo de 4. O valor superior é próximo ao tamanho de uma foto de alta resolução. Esse limite superior foi incrementado para observar se valores maiores que 1MB continuam a degradar o desempenho do RafeStore, conforme tendência observada no primeiro experimento (Seção 7.2.3). Sessenta replicações de cada teste foram realizadas.

Os provedores foram organizados de três maneiras (Tabela 4), considerando as Hipóteses 1, 2 e 3. Na configuração que é favorável ao RafeStore, os provedores obedecem a Definição 1 (Seção 4.1.1) e o

Tabela 4 – Configurações do experimento em *Cluster* e Nuvens: arranjos de provedores

	Somente dados	Dados (extra)	Somente metadados
Config. 1	p0 and p1	p2	p3
Config. 2	p1 and p2	p3	p0
Config. 3	p0 and p3	p1	p2

Fonte: próprio autor (2017).

provedor LOCAL é usado para armazenar dados (Configuração 1). A seguir, o provedor LOCAL foi utilizado apenas para armazenar metadados (Configuração 2). Na última variação (Configuração 3), os provedores foram organizados de forma a violar a Definição 1 (Seção 4.1.1).

7.2.6 Resultados em *Cluster* e Nuvens

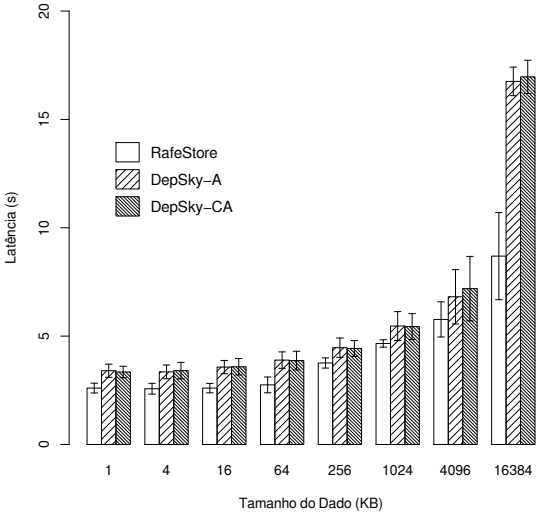
Na Configuração 1, o RafeStore apresentou desempenho melhor do que os outros métodos, em todos os tamanhos de dados avaliados (Figura 46). Esses resultados corroboram os resultados obtidos no primeiro experimento (Seção 7.2.3). Portanto, é possível afirmar que a Hipótese 1 é verdadeira também no cenário de nuvens. Quando os provedores respeitam a Definição 1, RafeStore tem desempenho melhor que os tradicionais quóruns.

Analisando a execução do RafeStore em nível de etapas (Figura 47), nota-se que apenas $f+1$ provedores de dados foram contactados, na escrita de dados com tamanho entre 1KB e 64KB. Entretanto, a partir da gravação de dados de tamanho 256KB, o provedor de dados adicional começou a ser contactado. A fase *writeDf* começa a possuir uma duração visível para dados iguais ou maiores que 256KB.

Uma informação relevante é a quantidade de vezes que os provedores de dados concluíram sua operação antes do término da fase de leitura de metadados (Tabela 5). Na Configuração 1 foi grande o número de execuções com sucesso, dentre as 60 execuções realizadas para cada teste, considerando dados com até 256KB.

A escrita de 16MB com o RafeStore apresentou um desempenho bastante superior aos protocolos do DepSky (Figura 46). Esse ganho considerável pode se originar do fato de que o RafeStore utiliza apenas $f+1$ provedores de dados, enquanto os protocolos do DepSky requerem $2f+1$ confirmações para realizar a gravação do dado.

Figura 46 – RafeStore e DepSky: latências, execução em *Cluster* e Nuvens, Configuração 1.



Fonte: próprio autor (2017).

Sobre a ativação de réplicas adicionais em dados maiores que 64KB, é notável que o tempo de gravação do dado supera o tempo necessário para que os metadados sejam gravados[1]. Esse fato é esperado e pode ser tratado com uma abordagem específica. Provedores de nuvem possuem APIs que permitem o envio de dados com múltiplas conexões. Na Amazon S3, pode-se enviar partes simultâneas do dado

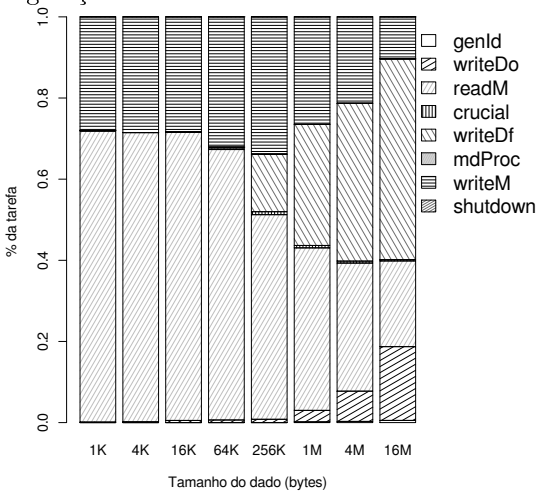
[1] Neste ponto surgem as consequências de considerar o RTT em vez da latência, conforme anunciado na Seção 4.1.1. Gravar dados maiores implica em tempos maiores (maior latência), fato que é desconsiderado ao utilizar-se apenas o RTT.

Tabela 5 – Quantidades de sucessos na antecipação de pedidos

data size	KB					MB		
	1	4	16	64	256	1024	4096	16384
Config. 1	60	59	60	52	0	0	0	0
Config. 2	52	54	47	58	60	53	59	46
Config. 3	0							

Fonte: próprio autor (2017).

Figura 47 – RafeStore: atuação de tarefas (%), execução em *Cluster* e Nuvens, Configuração 1.



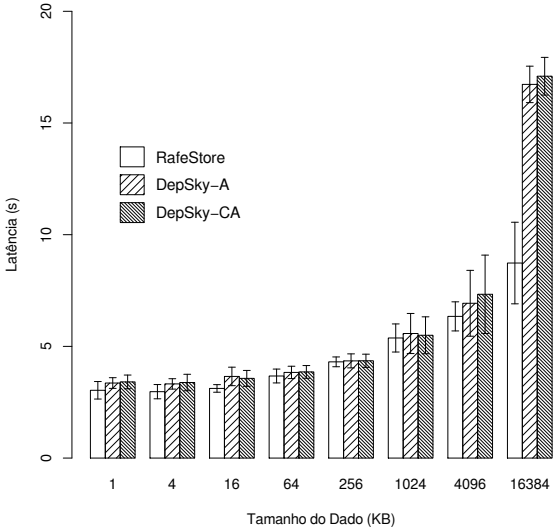
Fonte: próprio autor (2017).

(Apêndice A, item 35). A Azure, nuvem da Microsoft, permite o envio de múltiplos blocos (ou páginas) a fim de compor um único dado (Apêndice A, item 36). Dessa forma, pode-se utilizar o envio múltiplo do dado com o objetivo de concluir a etapa de escrita em tempo que satisfaça a Definição 1 (Seção 4.1.1). Essa alternativa remete à pergunta sobre o número de partes nas quais o envio deve ser fragmentado. Essa investigação é deixada para trabalhos futuros (Seção 8.4).

O envio em múltiplas partes pode favorecer também a operação de escrita de dados em caso de conexões de Internet lentas. Provedores costumam limitar operações de envio de dados com temporizadores. A Amazon S3, por exemplo, registra 20 segundos de limite (Apêndice A, item 15). No armazenamento de dados do tipo *blob* do provedor Microsoft Azure o tempo limite é de 90 segundos (Apêndice A, item 16). Com múltiplas conexões, é maior a chance de que as partes sejam gravadas, pois o tamanho das partes é menor que o tamanho integral. É possível ainda tratar o re-envio de apenas partes que falharam, mas possivelmente é preciso adaptar essa modificação na antecipação de pedidos (Figura 28).

Quando o armazenamento local é usado apenas para hospedar metadados (Tabela 4, Configuração 2), as latências do RafeStore au-

Figura 48 – RafeStore e DepSky: latências, execução em *Cluster* e Nuvens, Configuração 2.



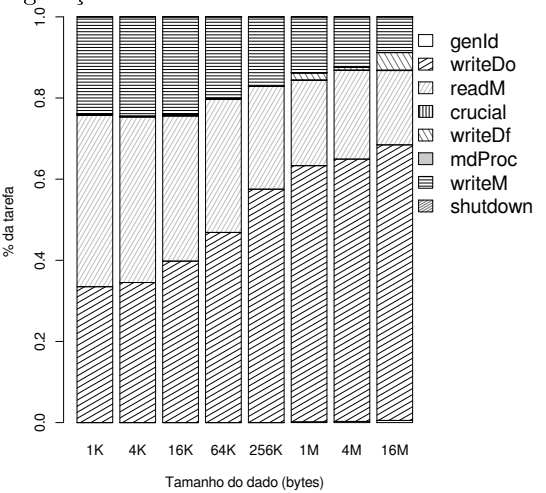
Fonte: próprio autor (2017).

mentam em relação àquelas observadas na Configuração 1. Entretanto, o RafeStore continua a apresentar melhores resultados do que o uso de quóruns Bizantinos tradicionais (Figura 49). Pode-se assim considerar a Hipótese 2 como verdadeira.

As distribuição de tarefas na Configuração 2 é bastante homogênea (Figura 49). Este comportamento é coerente: quanto mais dados a serem escritos, maior a duração da etapa de escrita de dados. Entretanto, a latência do RafeStore na Configuração 2 é maior que a latência na Configuração 1 (Figuras 46 e 48). Na Configuração 2 a Definição 1 (Seção 4.1.1) foi atendida na grande maioria das execuções (Tabela 5).

As latências observadas para a Configuração 3 (Figura 50) são similares aos resultados da Configuração 2 (Figura 48). Entretanto, conforme previsto, a Definição 1 é violada em todas as execuções (Tabela 5). A ativação do provedor de dados adicional ocorreu para os dados de todos os tamanhos avaliados (Figura 51). O RafeStore alcançou desempenho similar aos protocolos DepSky, para tamanhos de dados de 256KB, 1MB e 4MB. Para dados entre 1KB e 64KB, RafeStore apresentou melhor desempenho. Ainda, para o dado de 16MB,

Figura 49 – RafeStore: atuação de tarefas (%), execução em *Cluster* e Nuvens, Configuração 2.



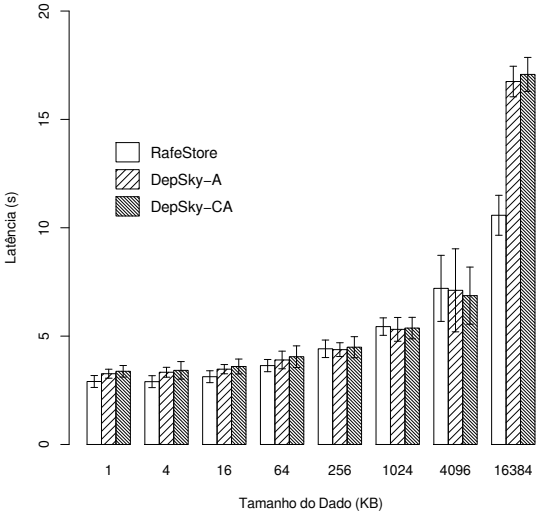
Fonte: próprio autor (2017).

o desempenho foi bastante superior. Pode-se portanto considerar a Hipótese 3 como verdadeira.

Os protocolos do DepSky apresentaram distribuição de etapas bastante semelhante, em todas as configurações avaliadas. Dessa maneira, toma-se como exemplo a distribuição do DepSky-A observada na Configuração 1 (Figura 52). Esse comportamento homogêneo deve-se ao fato de que os protocolos do DepSky enviam sempre pedidos a $3f+1$ provedores, obtendo a resposta dos $2f+1$ mais rápidos. Estes provedores não foram alterados, sob o ponto de vista dos procolos do DepSky, nas diferentes configurações (não há réplicas preferidas). O RafeStore apresentou melhor desempenho do que os demais protocolos, nos dados de maior tamanho. Na maioria das situações (exceto na Configuração 3, para os tamanhos de dados 256KB, 1MB e 4MB) foi melhor aguardar a resposta da gravação do dado integral de menos provedores do que usar *erasure codes* e espalhar o dado entre todos os provedores.

Sobre replicação integral e fragmentada: no contexto de tolerância a faltas, cabe um pequeno exemplo a favor da replicação integral (RI) em detrimento do uso de *erasure codes* (EC). Considere a necessidade de armazenar um arquivo de 10MB em provedores, de modo a tolerar uma falta bizantina ($f = 1$). Usando a estratégia RI

Figura 50 – RafeStore e DepSky: latências, execução em *Cluster* e Nuvens, Configuração 3.



Fonte: próprio autor (2017).

no RafeStore, o dado pode ser armazenado em 2 provedores de dados ($f + 1$), de tal forma que 20 MB serão ocupados. Ao usar a técnica EC, os dados são armazenados em 3 provedores de dados ($2f + 1$), requerendo apenas 15MB por causa da redução que o EC proporciona. Essa relação entre os espaços ocupados pelas técnicas varia de acordo com o número de faltas toleradas (Tabela 6).

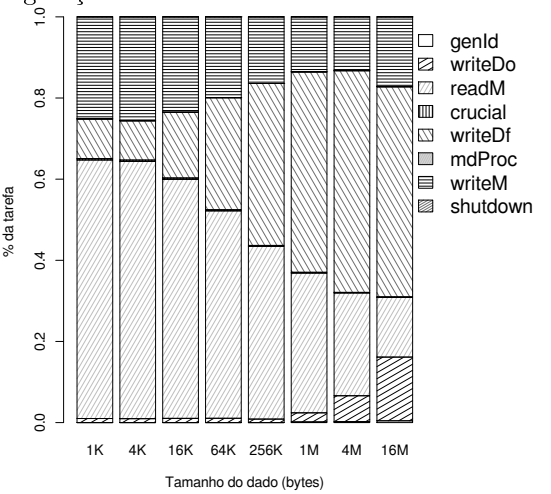
Aplicações que prezam pela sobrevivência do dado podem requerer um fator de tolerância a faltas maior que 1. Nesse caso, o espaço adicional consumido pela estratégia RI diminui na medida em que o

Tabela 6 – Espaço adicional (%): replicação integral x fragmentação

f	RI (MB)	EC (MB)	extra (MB)	% of extra
1	20	15	5	33.33
2	30	25	5	20
3	40	35	5	14.29
4	50	45	5	11.11
5	60	55	5	9.09

Fonte: próprio autor (2017).

Figura 51 – RafeStore: atuação de tarefas (%), execução em *Cluster* e Nuvens, Configuração 3.



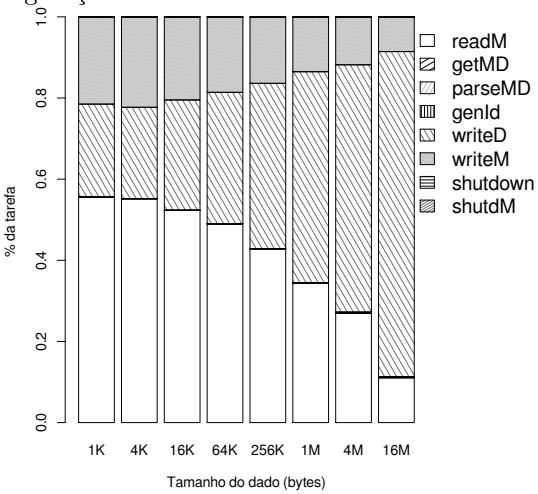
Fonte: próprio autor (2017).

número de faltas toleradas aumenta. Nesse contexto, a estratégia RI possui uma facilidade de recuperação de dados, quando comparada à estratégia EC. A transferência de estado pode ser feita de uma réplica para outra de maneira direta, na estratégia RI. Na estratégia de EC, para restaurar fragmentos é preciso restaurar o dado e reconstruir fragmentos. Transferência de estado é necessária durante a entrada de novas réplicas no sistema, seja para substituir réplicas faltosas ou para aumentar o número de faltas toleradas no sistema. O RafeStore usa a estratégia RI a fim de usar o menor número possível de provedores de dados. Hybris (Seção 3.1.8) também comenta a ampliação no número de réplicas, decorrente do uso de EC.

Conforme observado nos experimentos, a latência resultante da gravação de dados em provedores tem relação com o tamanho do dado e com o número de provedores nos quais o dado é gravado. Foi conduzido um micro-experimento usando o código original do DepSky (Apêndice A, item 32) a fim de obter outro conjunto de resultados e verificar o impacto do uso das técnicas RI e EC. O tamanho do dado foi variado de 1 byte até 3584 bytes[2], aumentando o tamanho de dado em 256

[2]Este tamanho de dado foi limitado pela entrada de caracteres no terminal, mas foi suficiente para suportar as assertivas apresentadas.

Figura 52 – DepSky-A: atuação de tarefas (%), execução em *Cluster* e Nuvens, Configuração 1.



Fonte: próprio autor (2017).

Tabela 7 – Espaço requerido (bytes) pelo DepSky em cada provedor.

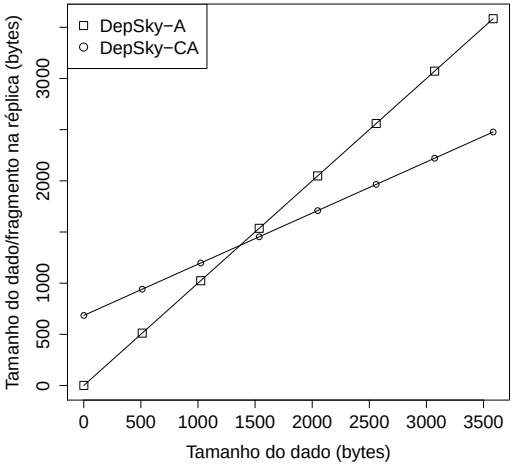
DepSky-A	1	512	1024	1536	2048	2560	3072	3584
DepSky-CA	684	941	1197	1453	1709	1965	2221	2477

Fonte: próprio autor (2017).

bytes a cada variação. No DepSky-A o dado é replicado integralmente em cada provedor, enquanto no DepSky-CA os tamanhos representados (Tabela 7 e Figura 53) significam o tamanho de cada fragmento. Para dados menores que 1KB, os fragmentos apresentam tamanho maior do que o dado integral. Isso pode ter ocorrido devido à criptografia aplicada pelo DepSky-CA, ou por informação adicional requerida pelo *erasure code*. Dessa maneira, pode haver uma vantagem em usar replicação integral ao invés de *erasure codes*, quando o tamanho do dado é pequeno[3]. Assim, se o dado deve ser confiável mas não precisa ser confidencial, o RafeStore pode ser a melhor escolha em termos de latência e utilização de espaço nos provedores.

[3]No RACS (Seção 3.1.1), metadados são replicados em todos os servidores, e menciona-se que "custos de replicação dos metadados não são proibitivos, a menos que a carga seja dominada por objetos muito pequenos".

Figura 53 – DepSKy: espaço ocupado nos provedores.



Fonte: próprio autor (2017).

7.3 AVALIAÇÃO DO DORADO

Nesta seção o DORADO é avaliado experimentalmente. O Kubernetes foi instalado no ambiente *Cluster* (Seção 7.1). Um computador atuou como máquina principal e outros três atuaram como nós (Seção 2.3). O Raft (Seção 3.2.3) não foi habilitado porque o *etcd* não operou em modo distribuído (foi executado apenas no nó principal). O sistema operacional utilizado nas máquinas foi o Ubuntu.

Testes preliminares com DORADO indicaram uma alta atividade no disco rígido da máquina principal. Essa atividade deve-se ao fato do *etcd* ter sido usado como memória compartilhada (MC). A fim de reduzir a latência, um disco virtual foi montado em memória RAM e o *etcd* foi configurado para armazenar os dados nesse disco. Embora a persistência dos dados fique comprometida nesse caso, pode-se resolver essa questão com discos de estado sólido (SSD), caso a redução de latência seja verificada. Assim, uma primeira pergunta de investigação busca verificar se o uso de MC apoiada sobre um disco virtual em RAM pode fornecer menor latência do que o uso da MC sobre um disco rígido.

Containers permitem uma alocação de recursos rápida, em comparação ao uso de máquinas virtuais tradicionais (Seção 2.3). Assim, pode-se supor que a vazão do sistema não cairá de forma abrupta ao aumentar o número de réplicas no sistema. Adicionalmente, o uso de

um protocolo baseado em memória compartilhada simplifica a comunicação de rede, o que pode ser benéfico na medida em que o número de réplicas cresce. Uma segunda pergunta de investigação pretende então avaliar se a vazão do sistema pode diminuir gradualmente na medida em que mais réplicas entram no sistema.

O protocolo DORADO (Seção 5.2) foi implementado na linguagem Go (Apêndice A, item 30), versão 1.4.2. A aplicação avaliada mantém um contador compartilhado. Três comandos foram usados alternadamente (estilo *round-robin*): *get* e *inc* retornam e incrementam o valor do contador, respectivamente. O comando *dou* divide o contador por 2 se o valor do contador for maior que 30. A aplicação foi hospedada em um *container*, cuja imagem está disponível no Hub (Apêndice A, item 41) do Docker, sob a tag *replicatedcalc*. O código-fonte da aplicação está disponível no GitHub (Apêndice A, item 42).

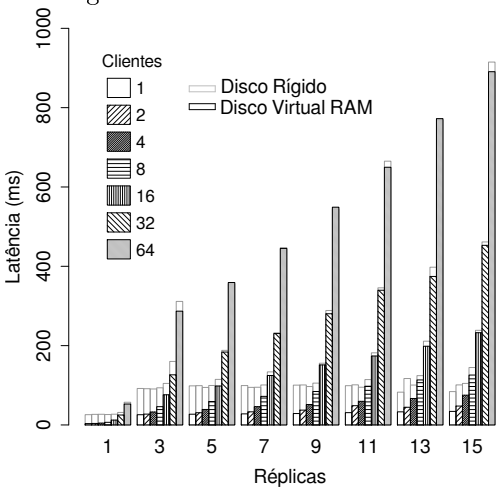
DORADO usa um coletor de lixo na MC para evitar o crescimento ilimitado de dados. Um mecanismo de *checkpoint* também é necessário para limitar a quantidade de mensagens necessárias para restaurar o estado. Entretanto, a implementação do protótipo atual não contém esses mecanismos.

Para investigar a influência da mídia que apoia a MC, um disco virtual de 4GB foi montado[4] na RAM da máquina principal do *cluster* Kubernetes. O experimento foi executado usando a MC sobre o disco rígido e posteriormente com a MC sobre o disco virtual. A relação entre o número de réplicas e a latência motivou a variação no número de réplicas de 1 a 15, seguindo a resiliência $n = 2f + 1$ (Seção 2.1.3). O número de clientes foi variado de 1 a 64, dobrando a quantidade de clientes a cada variação. Para simular o acesso de múltiplos clientes no sistema, foram utilizadas *Go routines*, que desempenham função semelhante a *threads*. O tamanho dos pedidos foi de cerca de 100 bytes. Cada cliente fez 50 solicitações ao sistema. Um cliente envia um próximo pedido somente após receber a resposta de um pedido atual.

Resultados: a latência é menor quando usa-se o disco virtual, em comparação ao uso do disco rígido (Figura 54, barras não-preenchidas referem-se ao disco rígido, barras preenchidas referem-se ao disco virtual em RAM). Entretanto, na medida em que mais clientes acessam o sistema, a diferença de latência devido ao uso das diferentes mídias decai. É fato que discos rígidos usam uma mídia rotacional, e a natureza mecânica do movimento da cabeça de leitura causa altas latências quando realiza leituras aleatórias (CHEN; KOUFATY; ZHANG, 2009). Ainda que a MC seja usada majoritariamente

[4] Foi usado o sistema de arquivos *tmpfs* para montar o disco virtual.

Figura 54 – DORADO: latência.



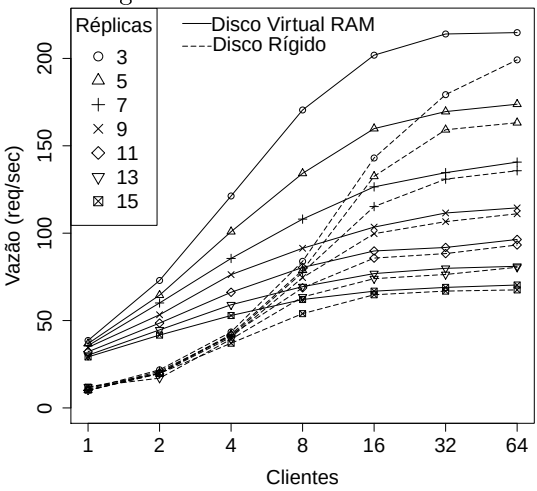
Fonte: próprio autor (2017).

para escrita, durante os experimentos observou-se que o *etcd* não faz uso de memória *cache*[5]. Isso significa que todas as operações de escrita são imediatamente efetivadas na mídia. A latência adicional requerida pelo disco rígido, em comparação ao disco virtual, existe tanto para 2 clientes quanto para 32 clientes acessando simultaneamente o sistema. Porém, no caso de muitos acessos simultâneos (neste experimento, a partir de 16 clientes), a latência adicional advinda do disco rígido se sobressai à latência de processamento das muitas requisições. O resultado foi uma indiferença na latência percebida, mediante muitas requisições, em ambos os cenários com as diferentes mídias avaliadas. Dessa maneira, a primeira pergunta de investigação está respondida.

A vazão apresentada pelo sistema, assim como a latência percebida pelos clientes, é influenciada pelo tipo de mídia usado na MC (Figura 55) apenas na situação em que poucos clientes acessam o sistema. Com o aumento do número de clientes, a vazão apresentada com o uso das diferentes mídias converge para um único valor. Relativo à segunda pergunta de investigação, observa-se que a vazão diminui gradualmente conforme mais réplicas ingressam no sistema.

[5] Foi verificada a informação de páginas *dirty* em proc/meminfo.

Figura 55 – DORADO: vazão.



Fonte: próprio autor (2017).

7.4 CONCLUSÕES DO CAPÍTULO

Este capítulo avaliou os sistemas RafeStore e DORADO, componentes do sistema FITS. O RafeStore foi executado em ambientes LAN e WAN, contando neste último com o uso de provedores de nuvem comerciais (Amazon, Microsoft Azure e HP Helion). Durante o desenvolvimento desta tese, a nuvem HP Helion mudou seu ramo de atuação (possibilidade apresentada em Seção 2.2.5), tornando-se especializada em provimento de soluções para nuvens híbridas. Isso reforça a relevância da *Intercloud* no armazenamento de dados críticos.

O DORADO foi avaliado em ambiente LAN, em um cenário de até 15 réplicas instanciadas em *containers*. Demonstrou-se a viabilidade de integrar coordenação de metadados em um ambiente disponível e promissor, o Kubernetes. O próximo capítulo conclui a tese, revisando os pontos principais e elencando as contribuições obtidas.

8 CONCLUSÃO

Não chore porque acabou, sorria porque aconteceu.
Dr. Seuss (Theodor Seuss Geisel)

Este capítulo conclui a tese. São revisitados as motivações e objetivos. A seguir, a visão geral do trabalho fornece uma síntese desta tese, sob a luz das contribuições e resultados obtidos. Na sequência, as contribuições deste trabalho são revistas. Finalmente, são identificados possíveis trabalhos futuros.

8.1 REVISÃO DAS MOTIVAÇÕES E OBJETIVOS

O armazenamento de dados é suscetível a falhas. Por isso, é usual fazer cópias dos dados, inclusive com uso de replicação geográfica, aumentando sua sobrevivência. Provedores de armazenamento em nuvens realizam geo-replicação usando *data centers* em diferentes locais.

Para aumentar a disponibilidade do dado de maneira eficaz e tolerar falhas eventualmente apresentadas por um provedor de nuvem, pode-se fazer uso de múltiplos provedores. Além disso, há aplicações críticas que requerem garantia de sucesso nas operações de leitura e escrita do dado. É possível garantir esse acesso com o uso de tolerância a faltas bizantinas (BFT). Porém, o custo de tolerar faltas bizantinas em ambientes de larga escala é uma barreira na adoção prática do BFT. O primeiro objetivo desta tese é a melhoria da eficiência do sistema de armazenamento de dados BFT.

Quando o dado é atualizado simultaneamente por múltiplos clientes, é preciso coordenar o acesso aos provedores. A integração da coordenação de metadados a um ambiente gerenciador de réplicas traz vantagens em comparação ao desenvolvimento de um novo sistema. Uma delas é que se pode implementar a nova funcionalidade com menor esforço, reaproveitando componentes existentes no ambiente, ao invés de criar um sistema totalmente novo. O segundo objetivo desta tese é a incorporação de coordenação de metadados no Kubernetes.

Por fim, o terceiro objetivo é especificar um sistema com arquitetura modular de armazenamento de dados em múltiplos provedores de nuvens que opere com as soluções de armazenamento de dados e coordenação de metadados propostas nesta tese.

8.2 VISÃO GERAL DO TRABALHO

Esta tese apresentou um estudo sobre como armazenar dados em múltiplos provedores de nuvens de maneira a tolerar eficientemente a ocorrência de faltas nos provedores.

O trabalho pode ser divido em três partes. A primeira consiste na elaboração da técnica denominada Antecipação de Pedidos (AdP). Essa técnica permite reduzir latência e custos em protocolos que utilizam quóruns. Com o uso da AdP foi construído o sistema de armazenamento RafeStore. O RafeStore foi avaliado e verificou-se suas vantagens em relação a outros protocolos. O objetivo de melhorar a eficiência do armazenamento de dados em múltiplos provedores de nuvem foi assim alcançado.

A segunda parte do trabalho abordou o tratamento de metadados, necessário para permitir a escrita de dados por múltiplos clientes simultaneamente. Buscou-se incorporar a coordenação dos metadados em um ambiente de gerenciamento de réplicas, de tal maneira que esse mecanismo de coordenação ficasse transparente aos usuários. Visando o uso da emergente tecnologia de *containers*, o Kubernetes foi escolhido para abrigar tal incorporação. O sistema DORADO foi desenvolvido para cumprir a função de coordenar os metadados. A avaliação de uma integração parcial do DORADO ao Kubernetes demonstrou a viabilidade dessa proposta. Assim, o objetivo de integrar coordenação de metadados a um ambiente de gerenciamento de réplicas foi alcançado.

Na última parte do trabalho, as soluções de armazenamento de dados e coordenação de metadados foram acopladas em um sistema de arquitetura modular denominado FITS. Esse sistema permite aplicar a AdP ao armazenamento de dados e coordenar o acesso de múltiplos usuários escritores de maneira transparente. Dessa maneira, alcançou-se o objetivo de obter simultaneamente as vantagens oferecidas pelos sistemas de armazenamento de dados e de coordenação de metadados.

8.3 CONTRIBUIÇÕES DESTA TESE

A presente tese tem três contribuições principais:

1. Uma técnica de antecipação de pedidos (AdP) (Seção 4.1), aplicada a um sistema de armazenamento denominado RafeStore.

2. A integração de coordenação de metadados no ambiente Kubernetes (Seção 5.2), que resultou no sistema DORADO.

3. Um sistema denominado FITS, capaz de operar simultaneamente os sistemas RafeStore e DORADO, provendo assim um armazenamento eficiente de dados de conteúdo independente com coordenação de metadados integrada (Seção 6.1).

A técnica de AdP melhora a eficiência de protocolos baseados em quóruns. A seguir estão listadas as características dessa contribuição:

- Hipótese: protocolos baseados em quóruns e que possuem múltiplas fases podem ser mais eficiente se utilizarem a informação de latência de cada servidor para antecipar pedidos entre fases.
- Requisito: conhecimento prévio das latências das réplicas.
- Resultados obtidos: menor latência, economia de recursos.
- Validação: a técnica foi aplicada em um sistema de armazenamento denominado RafeStore. O RafeStore foi inicialmente executado em rede LAN, com simulação de latências de rede WAN. Posteriormente, o experimento com o RafeStore foi realizado adicionando-se múltiplos provedores de nuvens. Os resultados obtidos em ambos os ambientes de rede corroboraram a conclusão sobre a hipótese.
- Artefatos: código-fonte do RafeStore (Apêndice A, item 14).

A proposta de integração de coordenação de metadados no Kubernetes permite a replicação de estados em *containers* de forma transparente para o usuário. Seguem características dessa contribuição:

- Hipótese: é possível integrar coordenação de metadados em *containers* no Kubernetes, tornando os *containers* mais eficientes e simplificando a operação da aplicação.
- Requisitos: o sistema alvo da integração precisa ter código aberto.
- Resultados:
  - redução no tamanho dos *containers*;
  - a aplicação tem coordenação de metadados disponível sem arcar com complexidades inerentes desse processo.
- Validação: foi implementado o protocolo DORADO, que usou um componente pré-existente no Kubernetes. A integração é parcial visto que no interior dos *containers* ainda reside código de coordenação de metadados. Para a integração total, é preciso mover

as ações de coordenação de metadados para um novo componente do Kubernetes, ou adicioná-las a um componente já existente.

- Artefatos: código-fonte do DORADO (Apêndice A, item 42).

A junção da antecipação de pedidos e da integração de coordenação de metadados em uma arquitetura permite o armazenamento de dados por múltiplos clientes de forma efetiva e transparente. Essa contribuição tem a seguinte Hipótese: é possível armazenar dados utilizando antecipação de pedidos em conjunto com a coordenação de metadados realizada de forma integrada no Kubernetes.

8.3.1 Publicações

As contribuições desta tese foram, em sua maioria, apresentadas à comunidade científica por meio de publicações, descritas a seguir.

1. Tolerância a Faltas e Intrusões para Sistemas de Armazenamento de Dados em Nuvens Computacionais. Hylson Netto, Lau Cheuk Lung e Rick Lopes de Souza. Minicurso ministrado no XIV Simpósio Brasileiro de Segurança da Informação e Sistemas Computacionais (SBSEG), em novembro de 2014.

2. RafStore: Armazenamento Confiável, Rápido e Geo-Replicado em Provedores de Nuvens. Hylson Netto, Tulio Ribeiro, Lau Cheuk Lung, Miguel Correia e Aldelir Luiz. Apresentado e publicado nos anais do XXXIII Simpósio Brasileiro de Redes de Computadores (SBRC), em maio de 2015. Menção honrosa.

3. *Anticipating Requests to Improve Performance and Reduce Costs in Cloud Storage*. Hylson Netto, Tulio Alberton Ribeiro, Lau Cheuk Lung, Miguel Correia e Aldelir Fernando Luiz. Publicado no ACM *Performance Evaluation Review* (PER), em 2015.

4. Replicação de Máquinas de Estado em Containers no Kubernetes: uma Proposta de Integração. Hylson Netto, Lau Cheuk Lung, Miguel Correia e Aldelir Fernando Luiz. Apresentado e publicado no anais do XXXIV Simpósio Brasileiro de Redes de Computadores (SBRC), em maio de 2016.

5. *State Machine Replication in Containers on Kubernetes*. Hylson Netto, Lau Cheuk Lung, Miguel Correia, Aldelir Fernando Luiz e Luciana Moreira Sa de Souza. Publicado no *Journal of Systems Architecture* (JSA), em dezembro de 2016.

Os artigos a seguir foram escritos em paralelo ao desenvolvimento desta tese e estão relacionadas a ela, mesmo quando encontra-se marginal ao escopo deste trabalho. Encontram-se junto às descrições abaixo as relações que o artigo possui com a tese.

1. *OB-STM: An Optimistic Approach for Byzantine Fault Tolerance in Software Transactional Memory.* Tulio Alberton Ribeiro, Lau Cheuk Lung e Hylson Vescovi Netto. Publicado nos anais do III Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC), em novembro de 2013. Relação com a tese: a utilização da ordem espontânea de mensagens na rede é uma forma de especulação, característica presente na técnica de antecipação de pedidos (Seção 4.1).

2. Lidando com Transações Interativas em um STM Tolerante a Faltas Bizantinas. Tulio Alberton Ribeiro, Lau Cheuk Lung e Hylson Vescovi Netto. Publicado nos anais do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), em maio de 2014. Relação com a tese: memórias transacionais em software são uma espécie de memória compartilhada. A coordenação de requisições enviadas aos *containers*, proposta nesta tese, utilizou um componente que usa tal abstração (Seção 5.2).

3. CloudSec - Um Middleware para Compartilhamento de Informações Sigilosas em Nuvens Computacionais. Rick Lopes de Souza, Hylson Vescovi Netto, Lau Cheuk Lung e Ricardo Felipe Custodio. Apresentado e publicado nos anais do XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG), em novembro de 2014. Relação com a tese: o CloudSec aborda, além de tolerância a faltas, aspectos de segurança em provedores de nuvens. Esse trabalho foi importante pois iniciou a prática de armazenamento de dados em nuvens.

4. *SSICC: Sharing Sensitive Information in a Cloud-of-Clouds.* Rick Lopes de Souza, Hylson Vescovi Netto, Lau Cheuk Lung e Ricardo Felipe Custodio. Publicado nos anais do IX *International Conference on Systems* (ICONS), em fevereiro de 2014. Relação com a tese: o SSICC é uma versão internacional, revisada e ampliada do trabalho CloudSec.

5. Replicação Tolerante a Faltas Eficiente em Bancos de Dados Orientados a Grafos. Ray Willy Neiheiser, Lau Cheuk Lung, Aldelir Fernando Luiz e Hylson Vescovi Netto. Publicado nos anais do

XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), em junho de 2016. Relação com a tese: investigou-se a possibilidade de trabalhar com armazenamento de dados em grafos e a antecipação de pedidos. Porém, constatou-se que o armazenamento de dados volumosos não é o objetivo desta tese (Seção 1.2).

6. *Towards a Transparent Cloud-Enabled Byzantine Fault-Tolerant Architecture for the Web.* Leandro Quibem Magnabosco, Lau Cheuk Lung, Hylson Netto e Miguel Correia. Aceito como *short-paper* na *International Conference on Computational Science*, em março de 2016. Esse artigo não foi apresentado nem publicado. Relação com a tese: o artigo proporcionou o estudo de arquiteturas híbridas e componentes confiáveis. A arquitetura de armazenamento proposta nesta tese (Seção 6.1) é híbrida: tolera faltas de parada nos provedores de metadados e faltas bizantinas nos provedores de dados. A proposta de coordenação de metadados integrada ao Kubernetes (Seção 5.2) faz uso de um componente confiável (tolerante a faltas de parada).

7. *Evaluating Raft in Docker on Kubernetes.* Caio Oliveira, Lau Cheuk Lung, Hylson Netto e Luciana Rech. Apresentado no *WorkShop on Cloud Computing*, evento integrante da *XIX International Conference on Systems Science*, em setembro de 2016. Publicado no *Advances in Intelligent Systems and Computing* 2016, Springer. Relação com a tese: o Raft possui muitas implementações. Ao modificar uma implementação simples do Raft para ser executada no Kubernetes, foi possível vislumbrar o projeto da incorporação de coordenação de metadados na modalidade de serviço (Seção 3.2.5). Um artigo está submetido ao SBRC 2017 apresentando a proposta de incorporar coordenação de metadados ao Kubernetes via serviço. Esta tese propõe (Seção 5.2) incorporar coordenação de metadados no Kubernetes via integração.

8.4 TRABALHOS FUTUROS

No estágio atual das contribuições obtidas nesta tese, são elencadas a seguir possibilidades de trabalhos futuros.

No contexto da antecipação de pedidos (AdP, Seção 4.1), pode-se investigar a possibilidade de aplicar a AdP em outros contextos. Por exemplo, se for possível antecipar fases de protocolos como o PBFT

(Seção 2.1.4), pode-se melhorar o desempenho percebido por clientes.

O RafeStore foi desenvolvido para melhorar a eficiência da operação de escrita de dados. Conforme os resultados (Seção 7.2), pode-se obter melhores resultados com o uso do *upload* simultâneo de múltiplas partes de um dado. Provedores de nuvem permitem o envio de múltiplas partes do arquivo simultaneamente. Torna-se oportuna uma investigação para encontrar a quantidade de partes nas quais o dado deve ser particionado. Possivelmente essa quantidade tem relação direta com o tamanho do arquivo.

Sobre a coordenação de metadados, a extensão do DORADO para o domínio de tolerância a faltas bizantinas é um desafio. Isso porque o isolamento entre *containers* ainda não é tão seguro quanto nas máquinas virtuais tradicionais. Camadas adicionais de segurança (como máquinas virtuais) podem ser necessárias para proteger o *container* de atores maliciosos (BURNS et al., 2016).

O DORADO pode ser implementado em outros sistemas como Apache Mesos (Apêndice A, item 33) e Docker Swarm (Apêndice A, item 37). Esses ambientes utilizam ZooKeeper e Consul, respectivamente, para manter informações de estado do *cluster*. ZooKeeper e Consul têm funcionalidades semelhantes ao *etcd* (Seção 2.2.6). O ZooKeeper pode ser usado para obter números sequenciais, por meio da *flag* "sequential", que provê aos *znodes* nomes únicos (HUNT et al., 2010). Esse recurso poderia ser usado para desenvolver um protocolo de ordenação de pedidos do tipo acordo entre destinatários (DÉFAGO; SCHIPER; URBÁN, 2004).

O DORADO pode ser modificado para que as eleições sejam realizadas sem usar memória compartilhada. Assim, o Raft não seria mais necessário para garantir a corretude das eleições. O Kubernetes pode ser usado para executar eleição de líder (Apêndice A, item 59).

REFERÊNCIAS

ABU-LIBDEH, H.; PRINCEHOUSE, L.; WEATHERSPOON, H. RACS: a case for cloud storage diversity. In: ACM. **Proceedings of the Symposium on Cloud computing**. New York, NY, USA: ACM, 2010. p. 229–240.

AGRAWAL, D.; KRISHNASWAMY, V. Using multiversion data for non-interfering execution of write-only transactions. In: **Proc. of the International Conference on Management of Data**. New York, NY, USA: ACM, 1991. p. 98–107.

AILIJIANG, A.; CHARAPKO, A.; DEMIRBAS, M. Consensus in the cloud: Paxos systems demystified. In: **25th International Conference on Computer Communication and Networks (ICCCN)**. USA: IEEE, 2016. p. 1–10.

ARMBRUST, M. et al. A view of cloud computing. **Communications of the ACM**, ACM, New York, NY, USA, v. 53, n. 4, p. 50–58, 2010.

ATTIYA, H.; BAR-NOY, A.; DOLEV, D. Sharing memory robustly in message-passing systems. **Journal of the ACM (JACM)**, ACM, v. 42, n. 1, p. 124–142, 1995.

ATTIYA, H.; WELCH, J. **Distributed computing: fundamentals, simulations, and advanced topics**. Hoboken, New Jersey: John Wiley & Sons, 2004.

AVIZIENIS, A.; KELLY, J. P. Fault tolerance by design diversity: Concepts and experiments. **Computer**, IEEE, v. 17, n. 8, p. 67–80, 1984.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE transactions on dependable and secure computing**, IEEE, v. 1, n. 1, p. 11–33, 2004.

BENZ, S.; SOUSA, L. P. de; PEDONE, F. Stretching multi-ring paxos. In: ACM. **Proceedings of the 31st Annual ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2016. p. 492–499.

BERNSTEIN, D. Containers and cloud: From LXC to Docker to Kubernetes. **IEEE Cloud Computing**, IEEE, v. 1, n. 3, p. 81–84, 2014.

BERNSTEIN, D. et al. Blueprint for the intercloud-protocols and formats for cloud computing interoperability. In: IEEE. **Internet and Web Applications and Services, 2009. ICIW'09. Fourth International Conference on**. Washington, DC, USA: IEEE Computer Society, 2009. p. 328–336.

BERNSTEIN, P. A.; GOODMAN, N. Multiversion concurrency control—theory and algorithms. **ACM Transactions on Database Systems (TODS)**, ACM, v. 8, n. 4, p. 465–483, 1983.

BESSANI, A. et al. DepSky: Dependable and secure storage in a cloud-of-clouds. **Transactions on Storage**, ACM, New York, NY, USA, v. 9, n. 4, p. 12:1–12:33, nov. 2013. ISSN 1553-3077.

BESSANI, A. et al. SCFS: a shared cloud-backed file system. In: **USENIX Annual Technical Conference (ATC)**. Philadelphia, PA: USENIX Association, 2014. p. 169–180.

BESSANI, A.; SOUSA, J.; ALCHIERI, E. E. State machine replication for the masses with BFT-SMaRt. In: IEEE. **Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on**. USA: IEEE, 2014. p. 355–362.

BESSANI, A. N.; LUNG, L. C.; FRAGA, J. da S. Extending the UMIOP specification for reliable multicast in CORBA. In: **International Symposium on Distributed Objects and Applications**. Berlin, Heidelberg: Springer, 2005. p. 662–679.

BOLOSKY, W. J. et al. Paxos replicated state machines as the basis of a high-performance data store. In: **Symposium on Networked Systems Design and Implementation (NSDI)**. Boston, MA: USENIX Association, 2011. p. 141–154.

BUDHIRAJA, N. et al. The primary-backup approach. **Distributed systems**, v. 2, p. 199–216, 1993.

BURNS, B. et al. Borg, Omega, and Kubernetes. **Communications of the ACM**, ACM, New York, NY, USA, v. 59, n. 5, p. 50–57, 2016.

CACHIN, C.; DOBRE, D.; VUKOLIĆ, M. Separating data and control: Asynchronous BFT storage with 2t+1 data replicas. In:

Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems. Paderborn, Germany: Springer International Publishing, 2014. p. 1–17.

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. **Introduction to reliable and secure distributed programming**. 2nd. ed. Berlin, Germany: Springer Science & Business Media, 2011.

CACHIN, C.; HAAS, R.; VUKOLIC, M. Dependable storage in the intercloud. **IBM Research**, v. 3783, p. 1–6, 2010.

CALDER, B. et al. Windows azure storage: A highly available cloud storage service with strong consistency. In: ACM. **Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2011. p. 143–157.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. **ACM Transactions on Computer Systems (TOCS)**, ACM, New York, NY, USA, v. 20, n. 4, p. 398–461, 2002.

CASTRO, M.; LISKOV, B. et al. Practical byzantine fault tolerance. In: **Third Symposium on Operating Systems Design and Implementation (OSDI)**. New Orleans, Louisiana, USA: USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999. v. 99, p. 173–186.

CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: an engineering perspective. In: ACM. **Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing**. New York, NY, USA: ACM, 2007. p. 398–407.

CHEN, F.; KOUFATY, D. A.; ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In: ACM. **ACM SIGMETRICS Performance Evaluation Review**. New York, NY, USA: ACM, 2009. v. 37, n. 1, p. 181–192.

CHO, B.; AGUILERA, M. K. Surviving congestion in geo-distributed storage systems. In: **Proceedings of the USENIX Annual Technical Conference**. Boston, MA: USENIX, 2012. p. 439–451.

CHUN, B.-G. et al. Attested append-only memory: Making adversaries stick to their word. In: ACM. **ACM SIGOPS**

Operating Systems Review. New York, NY, USA: ACM, 2007. v. 41, n. 6, p. 189–204.

CODD, E. F. A relational model of data for large shared data banks. **Communications of the ACM**, ACM, New York, NY, USA, v. 13, n. 6, p. 377–387, 1970.

CORREIA, M. Clouds-of-clouds for dependability and security: geo-replication meets the cloud. In: SPRINGER. **European Conference on Parallel Processing**. Berlin, Heidelberg: Springer, 2013. p. 95–104.

COULOURIS, G. F.; DOLLIMORE, J.; KINDBERG, T. **Distributed systems: concepts and design**. 5. ed. Boston, Massachusetts, USA: Pearson Education, 2012.

COUTO, R. S. et al. Latência versus sobrevivência no projeto de centros de dados geograficamente distribuidos. In: **Simposio Brasileiro de Redes de Computadores e Sistemas Distribuidos (SBRC)**. Porto Alegre: Sociedade Brasileira de Computacao - SBC, 2014.

CUI, H. et al. Paxos made transparent. In: ACM. **Proceedings of the 25th Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2015. p. 105–120.

DECANDIA, G. et al. Dynamo: amazon's highly available key-value store. **ACM SIGOPS Operating Systems Review**, ACM, New York, NY, USA, v. 41, n. 6, p. 205–220, 2007.

DÉFAGO, X.; SCHIPER, A.; URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 36, n. 4, p. 372–421, 2004.

DESWARTE, Y.; BLAIN, L.; FABRE, J.-C. Intrusion tolerance in distributed computing systems. In: IEEE. **IEEE Computer Society Symposium on Research in Security and Privacy**. Oakland, CA: IEEE, 1991. p. 110–121.

DETTONI, F. et al. Byzantine fault-tolerant state machine replication with twin virtual machines. In: **Proceedings of the IEEE Symposium on Computers and Communications (ISCC)**. USA: IEEE, 2013. p. 398–403.

DOBRE, D.; VIOTTI, P.; VUKOLIĆ, M. Hybris: Robust hybrid cloud storage. In: **Proceedings of the 5th annual ACM Symposium on Cloud Computing**. New York, NY, USA: ACM, 2014. p. 1–14.

DOLEV, D.; STRONG, H. R. Authenticated algorithms for byzantine agreement. **SIAM Journal on Computing**, SIAM, USA, v. 12, n. 4, p. 656–666, 1983.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. **Journal of the ACM (JACM)**, ACM, New York, NY, USA, v. 35, n. 2, p. 288–323, 1988.

EASTLAKE 3RD, D.; JONES, P. **US Secure Hash Algorithm 1 (SHA1)**. USA: RFC Editor, 2001. IETF Network Working Group, RFC 3174.

ELMASRI, R.; NAVATHE, S. **Sistemas de Banco de Dados**. 6. ed. São Paulo, SP: Pearson Education, 2011.

FELBER, P.; NARASIMHAN, P. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. **Transactions on Computers**, IEEE, USA, v. 53, n. 5, p. 497–511, 2004.

FELTER, W. et al. An updated performance comparison of virtual machines and linux containers. In: IEEE. **International Symposium on Performance Analysis of Systems and Software (ISPASS)**. Penn's Landing, Philidelphia, USA: IEEE, 2015. p. 171–172.

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. **Journal of the ACM (JACM)**, ACM, New York, NY, USA, v. 32, n. 2, p. 374–382, 1985.

FRAGA, J.; POWELL, D. A fault-and intrusion-tolerant file system. In: **Proceedings of the 3rd international conference on computer security**. Dublin, Ireland: North-Holland,Amsterdam, Neth, 1985. v. 203, p. 203–218.

GIFFORD, D. K. Weighted voting for replicated data. In: ACM. **Proceedings of the seventh ACM symposium on Operating systems principles**. New York, NY, USA: ACM, 1979. p. 150–162.

GILBERT, S.; LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. **ACM SIGACT News**, ACM, New York, NY, USA, v. 33, n. 2, p. 51–59, 2002.

GILBERT, S.; LYNCH, N. Perspectives on the CAP theorem. **Computer**, v. 45, n. 2, p. 30–36, Feb 2012.

GOLDBERG, R. P. Survey of virtual machine research. **Computer**, IEEE, USA, v. 7, n. 6, p. 34–45, 1974.

GUERRAOUI, R.; YABANDEH, M. Independent faults in the cloud. In: ACM. **Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware**. New York, NY, USA: ACM, 2010. p. 12–17.

HADZILACOS, V.; TOUEG, S. **A modular approach to fault-tolerant broadcasts and related problems**. Ithaca NY, May 1994.

HAMMING, R. W. Error detecting and error correcting codes. **Bell System technical journal**, Wiley Online Library, v. 29, n. 2, p. 147–160, 1950.

HERLIHY, M. Wait-free synchronization. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 13, n. 1, p. 124–149, 1991.

HERLIHY, M.; LUCHANGCO, V.; MOIR, M. Obstruction-free synchronization: Double-ended queues as an example. In: IEEE. **Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on**. Providence, RI, USA: Institute of Electrical and Electronics Engineers Inc., 2003. p. 522–529.

HERLIHY, M. P.; WING, J. M. Linearizability: A correctness condition for concurrent objects. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, New York, NY, USA, v. 12, n. 3, p. 463–492, 1990.

HU, W.; YANG, T.; MATTHEWS, J. N. The good, the bad and the ugly of consumer cloud storage. **ACM SIGOPS Operating Systems Review**, ACM, New York, NY, USA, v. 44, n. 3, p. 110–115, 2010.

HUNT, P. et al. Zookeeper: Wait-free coordination for internet-scale systems. In: **USENIX Annual Technical Conference**. Philadelphia, PA: USENIX Association, 2010. v. 8, p. 9.

KRAWCZYK, H. Secret sharing made short. In: SPRINGER. **Annual International Cryptology Conference**. Berlin, Heidelberg: Springer, 1993. p. 136–146.

LAKSHMAN, A.; MALIK, P. Cassandra: a decentralized structured storage system. **ACM SIGOPS Operating Systems Review**, ACM, New York, NY, USA, v. 44, n. 2, p. 35–40, 2010.

LAMPORT, L. On interprocess communication. **Distributed Computing**, Springer, Berlin, Heidelberg, v. 1, n. 2, p. 86–101, 1986. ISSN 1432-0452.

LAMPORT, L. The part-time parliament. **ACM Transactions on Computer Systems (TOCS)**, ACM, New York, NY, USA, v. 16, n. 2, p. 133–169, 1998.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, New York, NY, USA, v. 4, n. 3, p. 382–401, 1982.

LEACH, P.; MEALLING, M.; SALZ, R. **A Universally Unique IDentifier (UUID) URN Namespace (RFC 4122)**. Jul 2005. 1–32 p. IETF Request For Comments.

LEAVITT, N. Will NoSQL databases live up to their promise? **Computer**, IEEE, v. 43, n. 2, p. 12–14, 2010.

LISKOV, B.; RODRIGUES, R. Tolerating byzantine faulty clients in a quorum system. In: IEEE. **26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)**. USA: IEEE, 2006. p. 34–34.

LUNG, L. C. et al. Experiências com comunicação de grupo nas especificações *Fault Tolerant* CORBA. In: **Simpósio Brasileiro de Redes de Computadores**. Belo Horizonte - MG: Sociedade Brasileira de Computacao - SBC, 2000. p. 1–14.

MAHAJAN, P. et al. Depot: Cloud storage with minimal trust. **ACM TOCS**, ACM, New York, NY, USA, v. 29, n. 4, p. 12:1–12:38, 2011. ISSN 0734-2071.

MALKHI, D.; REITER, M. Byzantine quorum systems. **Distributed Computing**, Springer-Verlag, Berlin, Heidelberg, v. 11, n. 4, p. 203–213, 1998. ISSN 0178-2770.

MARANDI, P. J.; PRIMI, M.; PEDONE, F. Multi-Ring Paxos. In: IEEE. **IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)**. USA: IEEE, 2012. p. 1–12.

MARANDI, P. J. et al. Ring Paxos: A high-throughput atomic broadcast protocol. In: IEEE. **Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on**. USA: IEEE, 2010. p. 527–536.

MARTIN, J.-P.; ALVISI, L.; DAHLIN, M. Minimal byzantine storage. In: **Proceedings of the 16th International Conference on Distributed Computing**. Berlin, Heidelberg: Springer, 2002. p. 311–325.

MAZIERES, D.; SHASHA, D. Building secure file systems out of byzantine storage. In: ACM. **Proceedings of the twenty-first annual symposium on Principles of distributed computing**. New York, NY, USA: ACM, 2002. p. 108–117.

MELL, P.; GRANCE, T. The NIST definition of cloud computing. **Communications of the ACM**, v. 53, n. 6, p. 50, 2010.

MELLIAR-SMITH, P. M.; MOSER, L. E.; AGRAWALA, V. Broadcast protocols for distributed systems. **Transactions on Parallel and Distributed Systems**, IEEE, USA, v. 1, n. 1, p. 17–25, Jan 1990. ISSN 1045-9219.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux Journal**, Belltown Media, Houston, TX, USA, v. 2014, n. 239, p. 2, 2014.

NARASIMHAN, P.; MOSER, L. E.; MELLIAR-SMITH, P. M. Exploiting the internet inter-ORB protocol interface to provide CORBA with fault tolerance. In: **COOTS**. Portland, Oregon, USA: USENIX Association, 1997. v. 97, p. 6–6.

NETTO, H. V. et al. RafStore: Armazenamento confiável, rápido e geo-replicado em provedores de nuvens. In: **Simpósio Brasileiro de Redes de Computadores**. Vitória, ES: Sociedade Brasileira de Computacao - SBC, 2015. p. 39–52.

OLIVEIRA, C. et al. Evaluating Raft in Docker on Kubernetes. In: **Advances in Intelligent Systems and Computing**. Berlin, Heidelberg: Springer, 2016.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: **USENIX Annual Technical Conference**. San Diego, CA, USA: USENIX Association, 2014. p. 305–320.

PACHECO, L. et al. Globalfs: A strongly consistent multi-site file system. In: **35th Symposium on Reliable Distributed Systems (SRDS)**. USA: IEEE, 2016. p. 147–156.

PÂRIS, J.-F.; LONG, D. D. Voting with regenerable volatile witnesses. In: **Data Engineering, 1991. Proceedings. Seventh International Conference on**. USA: IEEE, 1991. p. 112–119.

PATTERSON, D. A.; GIBSON, G.; KATZ, R. H. **A case for redundant arrays of inexpensive disks (RAID)**. New York, NY, USA: ACM, 1988.

PEINL, R.; HOLZSCHUHER, F.; PFITZER, F. Docker cluster management for the cloud - survey results and own solution. **Journal of Grid Computing**, Springer, Berlin, Heidelberg, p. 1–18, 2016.

PLANK, J. S.; SIMMERMAN, S.; SCHUMAN, C. D. **Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2.** University of Tennessee, 2008.

RABIN, M. O. Efficient dispersal of information for security, load balancing, and fault tolerance. **Journal of the ACM (JACM)**, ACM, New York, NY, USA, v. 36, n. 2, p. 335–348, 1989.

REED, B.; JUNQUEIRA, F. P. A simple totally ordered broadcast protocol. In: ACM. **proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware**. New York, NY, USA: ACM, 2008. p. 2.

RENESSE, R. V.; SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In: **Symposium on Operating Systems Design and Implementation (OSDI)**. Boston, MA, USA: USENIX Association, 2004. v. 4, p. 91–104.

RIVEST, R. The MD5 message-digest algorithm. IETF Network Working Group, RFC 1321, p. 1–21, April 1992.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. **Communications of the ACM**, ACM, New York, NY, USA, v. 21, n. 2, p. 120–126, 1978.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. **ACM Computing Surveys**, v. 22, n. 4, p. 299–319, 1990.

SHAMIR, A. How to share a secret. **Communications of the ACM**, ACM, New York, NY, USA, v. 22, n. 11, p. 612–613, 1979.

SILVA, M. R. X. et al. Tolerância a faltas bizantinas usando registradores compartilhados distribuídos. In: **Simpósio Brasileiro de Redes de Computadores**. Brasília,DF: Sociedade Brasileira de Computacao - SBC, 2013.

SILVA, M. R. X. et al. BAMcast-byzantine fault-tolerant consensus service for atomic multicast in large-scale networks. In: IEEE. **2013 IEEE Symposium on Computers and Communications (ISCC)**. USA: IEEE, 2013. p. 000324–000329.

SIMONS, B. An overview of clock synchronization. In: **Fault-Tolerant Distributed Computing**. Berlin, Heidelberg: Springer, 1990. p. 84–96.

SMITH, D. M. The cost of lost data. **Journal of Contemporary Business Practice**, Graziadio Business Review, Los Angeles, CA, v. 6, n. 3, p. 113–119, 2003.

SOLTESZ, S. et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: **ACM SIGOPS Operating Systems Review**. New York, NY, USA: ACM, 2007. v. 41, n. 3, p. 275–287.

SOTOMAYOR, B. et al. Virtual infrastructure management in private and hybrid clouds. **IEEE Internet computing**, IEEE, USA, v. 13, n. 5, p. 14–22, 2009.

SOUSA, J.; BESSANI, A. Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines. In: IEEE. **Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on**. USA: IEEE, 2015. p. 146–155.

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems**. USA: Prentice-Hall, 2007.

TERRY, D. B. et al. Consistency-based service level agreements for cloud storage. In: **Proceedings of the 24th Symposium on Operating Systems Principles (SOSP)**. Farmington, Pennsylvania, USA: USENIX Association, 2013. p. 309–324.

VERMA, A. et al. Large-scale cluster management at Google with Borg. In: ACM. **European Conference on Computer Systems**. New York, NY, USA: ACM, 2015. p. 18.

VERONESE, G. S. et al. Efficient byzantine fault-tolerance. **Transactions on Computers**, IEEE, USA, v. 62, n. 1, p. 16–30, 2013.

VOGELS, W. Eventually consistent. **Communications of the ACM**, ACM, New York, NY, USA, v. 52, n. 1, p. 40–44, 2009.

WANG, X.; YU, H. How to break MD5 and other hash functions. In: SPRINGER. **Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)**. Berlin, Heidelberg: Springer, 2005. p. 19–35.

WANG, Y.; ALVISI, L.; DAHLIN, M. Gnothi: Separating data and metadata for efficient and available storage replication. In: **the USENIX Annual Technical Conference**. Boston, MA: USENIX, 2012. p. 413–424.

WEATHERSPOON, H.; KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In: SPRINGER. **International Workshop on Peer-to-Peer Systems**. Berlin, Heidelberg: Springer, 2002. p. 328–337.

WU, Z. et al. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In: **Proceedings of the 24th Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2013. p. 292–308.

XAVIER, M. G. et al. Performance evaluation of container-based virtualization for high performance computing environments. In: **21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. USA: IEEE, 2013. p. 233–240.

APÊNDICE A – Citações de Endereços da Internet

Os endereços a seguir contém notícias que são referenciada ao longo esta tese. Todas as URL foram visitadas no dia 21 de fevereiro de 2017. A página `http://www.hylson.com/urls-tese.html` disponibiliza os endereços abaixo.

1. Preços de provedores de armazenamento em nuvem estão se tornando mais baratos.

   `https://techcrunch.com/2014/08/20/cloud-storage-is-eating-alive-traditional-storage`

2. Redução de preços na Amazon.

   `https://aws.amazon.com/blogs/aws/amazon-s3-price-reduction`

3. Outra notícia sobre redução de preços na Amazon.

   `https://aws.amazon.com/blogs/aws/aws-update-new-m3-features-reduced-ebs-prices-reduced-s3-prices`

4. Alteração de preços na nuvem da Google.

   `https://www.theregister.co.uk/2014/03/25/google_price_slash`

5. Alteração de preços na nuvem Microsoft Azure.

   `https://www.theregister.co.uk/2014/12/05/google_microsoft_what_the_heck_do_they_do_in_cloud`

6. Blog do CoreOS, sobre a implementação de *containers* rocket.

   `https://coreos.com/blog/rocket`

7. Agendador de recursos que pode ser usado no Apache Mesos.

   `https://mesosphere.github.io/marathon`

8. Página do *etcd*.

   `https://coreos.com/etcd`

9. Proposta de federação de *clusters* Kubernetes.

   `https://github.com/kubernetes/kubernetes/blob/master/docs/proposals/federation.md`

10. Federação de *Clusters* Kubernetes: manual de operação.

    `https://kubernetes.io/docs/admin/federation/`

11. Bancos de Dados mais utilizados.

    https://db-engines.com/en/ranking

12. Serviço Google API Engine indisponível em todos os locais por 18 minutos..

    https://status.cloud.google.com/incident/compute/16007

13. IBM pretende usar ICStore (Seção 3.1.2).

    https://www.pcworld.com/article/2070340/ibm-lays-plans-to-be-a-cloud-storage-broker.html

14. Código-fonte do protótipo do RafeStore.

    https://www.hylson.com/rafestore15

15. *Timeout* de escrita no serviço Amazon S3.

    https://https://forums.aws.amazon.com/message.jspa?messageID=215367

16. *Timeout* de escrita no serviço Azure Blog Storage.

    https://https://blogs.msdn.microsoft.com/cellfish/2012/01/18/avoid-timeout-when-uploading-large-blobs-to-azure/

17. Site do Flocker, que replica estados em *containers* no k8s.

    https://https://clusterhq.com/flocker/introduction/

18. Participação de cidadãos na criação de leis.

    https://https://www12.senado.leg.br/ecidadania/principalideia

19. Proposta para definir que os políticos receberão salário mínimo.

    https://https://www12.senado.leg.br/ecidadania/visualizacaoideia?id=52952

20. Site da Amazon EC2.

    https://https://aws.amazon.com/ec2/

21. Site do provedor linode.

    https://www.linode.com

22. Site do serviço Amazon S3.

    https://aws.amazon.com/s3

23. Serviço de armazenamento do provedor Microsoft Azure.
`https://azure.microsoft.com/en-us/services/storage/`

24. Consistência de atualização de dados no Amazon S3.
`https://aws.amazon.com/s3/faqs/`

25. Consistência de listagem de objetos no Amazon S3.
`https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel`

26. Virtualização em nível de sistema no Linux - LCX.
`https://linuxcontainers.org`

27. Estimador de custos para serviços em nuvem.
`https://www.planforcloud.com/`

28. Preços do serviço Amazon S3.
`https://aws.amazon.com/s3/pricing/`

29. Site do algoritmo de consenso Raft.
`https://raft.github.io/`

30. Site da linguagem Go.
`https://golang.org/`

31. Biblioteca JEC (Java), que implementa *erasure codes (Seção 2.2.4)*.
`https://github.com/cloud-of-clouds/depsky/tree/master/JEC`

32. Site do DepSky (Seção 3.1.3).
`https://cloud-of-clouds.github.io/depsky/`

33. Site do Apache Mesos.
`https://mesos.apache.org`

34. Site do Banco de Dados ReDis.
`https://redis.io/`

35. API da Amazon S3 que permite *upload* em múltiplas partes.
`https://docs.aws.amazon.com/AmazonS3/latest/dev/uploadobjusingmpu.html`

36. API da Azure que permite *upload* em múltiplas partes.

    https://docs.microsoft.com/pt-br/rest/api/storageservices/fileservices/Put-Block-List

37. Site do Docker Swarm.

    https://www.docker.com/products/docker-swarm

38. A aplicação que usa Flocker deve tratar acesso concorrente a volume externo.

    https://flocker-docs.clusterhq.com/en/latest/faq/index.html#can-more-than-one-container-access-the-same-volume

39. Amazon S3: limite de criação de *buckets* por conta.

    https://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html

40. Airbnb: dados sobre residências chinesas em aluguel devem ser hospedados em *data center* da China!.

    https://www.airbnb.com.br/home/terms-of-service-event?eluid=6&euid=05acfd5c-9ec4-4fd6-b90c-23e7273a0052

41. Site que contém imagens do Docker.

    https://hub.docker.com

42. Repositório com código-fonte da coordenação de metadados integrada ao Kubernetes.

    https://github.com/hvescovi/learnGo

43. Interrupção em serviços da Amazon tornam o NetFlix indisponível.

    https://www.forbes.com/sites/anthonykosner/2012/06/30/amazon-cloud-goes-down-friday-night-taking-netflix-instagram-and-pinterest-with-it

44. Interrupção na nuvem Microsoft Azure.

    https://www.theregister.co.uk/2014/08/18/azure_outage

45. Interrupção de 48 horas programada.

    https://www.theregister.co.uk/2015/01/05/verizon_cloud_plans_48hour_maintenance_outage

46. Disponibilidade de provedores de nuvem, primeiro semestre de 2014.

    https://www.cloudendure.com/blog/cloud-service-availability-q2-results-whos-up

47. Interrupções em provedores de nuvem, em 2014.

    https://www.theregister.co.uk/2015/01/16/microsoft_worst_cloud_uptime_2014

48. Serviço Google API Engine é interrompido devido a ataques.

    https://www.theregister.co.uk/2015/02/20/expired_router_cache_send_google_cloud_engine_titsup

49. DDoS no provedor RackSpace.

    https://www.theregister.co.uk/2014/12/24/rackspace_restored_after_ddos_takes_out_dns

50. Site Photopoint.com é desativado sem avisar aos usuários.

    https://www.cnet.com/news/victims-of-lost-files-out-of-luck

51. Nuvem Nirvanix é desativada.

    https://www.computerweekly.com/opinion/Nirvanix-failure-a-blow-to-the-cloud-storage-model

52. Provedor HP Helion é desativado.

    https://venturebeat.com/2015/10/21/hp-is-officially-shutting-down-its-helion-public-cloud-in-january-2016

53. Serviço de acesso a múltiplos provedores em nuvem.

    https://multcloud.com

54. Regiões com *data centers* da Amazon.

    https://aws.amazon.com/about-aws/global-infrastructure/

55. Regiões com *data centers* da Azure.

    https://azure.microsoft.com/en-us/regions/

56. *Amazon Elastic File System* - Amazon EFS.

    https://aws.amazon.com/efs/

57. Persistir dados de *containers* com o Amazon EFS.

    https://aws.amazon.com/blogs/compute/using-amazon-efs-to-persist-data-from-amazon-ecs-containers/

58. Raft usado no *etcd*.

    https://github.com/coreos/etcd/tree/master/raft

59. Uso do Kubernetes para realizar eleição de líder.

    https://blog.kubernetes.io/2016/01/simple-leader-election-with-Kubernetes.html

60. Arquitetura do Kubernetes.

    https://github.com/kubernetes/kubernetes/blob/master/docs/design/architecture.md

61. Emulador de WAN.

    https://wiki.linuxfoundation.org/networking/netem

62. Acordo de nível de serviço do Amazon S3.

    https://aws.amazon.com/s3/sla/

63. Rússia bloqueia LinkedIn por 24hs.

    http://www.bbc.com/news/technology-38014501