RafStore: Armazenamento Confiável, Rápido e Geo-Replicado em Provedores de Nuvens

Hylson Netto[1,2], Tulio Alberton Ribeiro[1], Lau Cheuk Lung[1], Miguel Correia[3], Aldelir Fernando Luiz[2]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina

[2]Campus Blumenau – Instituto Federal de Educação, Ciência e Tecnologia Catarinense

[3]INESC-ID – Instituto Superior Técnico – Universidade de Lisboa – Portugal

{hylson.vescovi,tulio.ribeiro,aldelir.luiz}@posgrad.ufsc.br,
lau.lung@ufsc.br, miguel.p.correia@tecnico.ulisboa.pt

Resumo. *Nos últimos tempos, o armazenamento distribuído de dados tem tido grande notoriedade em termos de investigação, em face à crescente oferta de provedores de armazenamento em nuvens. Aplicações que lidam com armazenamento distribuído em nuvens devem considerar aspectos de confiabilidade e disponibilidade. Este trabalho apresenta o RafStore, um protocolo que provê consistência regular, disponibilidade e tolerância a faltas bizantinas. Além disso, o desempenho do RafStore é melhor do que protocolos anteriores e os custos são reduzidos com o uso de menos provedores de dados, para aplicações nas quais o dado a ser armazenado não depende do dado armazenado previamente.*

Abstract. *Lately, distributed data storage has been gaining a lot of attention due the advent of a variety of cloud storage offerings. Applications that deal with distributed storage in clouds have to balance aspects such as reliability and availability. We present RafStore a protocol that provides regular consistency, availability and Byzantine fault tolerance. Additionally, RafStore's performance is better than previous works and costs are reduced by using fewer providers to store data, for applications in which the data to be stored does not depend on the data stored.*

1. Introdução

O paradigma introduzido pelo conceito de Computação em Nuvem (*Cloud Computing*) [NIST 2011] tem impelido no surgimento de uma série de facilidades para a realização de computações através da Internet. Uma das facilidades mais exploradas pelas aplicações na Internet é o armazenamento de dados em nuvem. De um modo geral, aplicações da Internet lidam com clientes que estão dispersos geograficamente, razão pela qual, elas tendem a armazenar dados de maneira distribuída, a fim de manter dados próximos dos clientes no intuito de prover melhor desempenho acerca das operações do cliente. Provedores de armazenamento estão disponíveis em nuvens, trazendo os benefícios de infraestruturas elásticas e custos sob demanda. O número de provedores de nuvens públicas tem aumentado, para armazenamento e para processamento de dados; assim, é possível disponibilizar aplicações e dados na Internet hospedando dados e programas em provedores de nuvens. Provedores passivos, como o Amazon S3 e o Google Cloud Datastore, são ideais para armazenar dados pois possuem custos reduzidos e mecanismos de acesso padronizados, o que torna a mudança de provedor fácil sob a perspectiva do cliente. Estes provedores passivos de armazenamento de dados seguem o modelo de Software como serviço (SaaS).

Os provedores de nuvens são distribuídos geograficamente em *data centers*, de modo que as aplicações precisam escolher um *data center* para armazenar seus dados. É evidente que para se obter um melhor desempenho, os dados devem permanecer próximos às aplicações. Uma estratégia bastante comum empregada para esta finalidade é a replicação geográfica de dados (i.e., geo-replicação), de modo a permitir que as aplicações obtenham os dados do *data center* mais próximo ao qual a aplicação está localizada. Por outro lado, a literatura aponta para o fato de que a geo-replicação introduz alguns problemas; o teorema CAP [Brewer 2012] declara que é impossível alcançar consistência, disponibilidade e particionamento de redes simultaneamente em dados replicados sobre redes de longa distância, como a Internet. Desde o advento deste anúncio – em 2002 – muitos trabalhos têm focado no equilíbrio destas dimensões com o objetivo de satisfazer demandas das aplicações.

Em se tratando de dados, uma característica de particular interesse em sistemas de armazenamento é a relação entre valores atuais e novos valores. No caso, se um dado precisa considerar o valor atual para criar um novo valor, pode-se considerar que novos valores deste dado são *dependentes* de valores anteriores. Esta característica implica inclusive que o dado seja atualizado de uma maneira transacional para evitar leituras sujas. Por exemplo, para um contador compartilhado o valor atual precisa ser lido e incrementado, e então o novo valor pode ser escrito. De outro modo, se um dado pode ser escrito no sistema sem depender do valor anterior, a operação de escrita é semelhante a um *blind-write* [Agrawal and Krishnaswamy 1991], ou uma operação de somente-escrita. Para fins de brevidade, vamos denominar este tipo de dado como dado de *conteúdo independente*. Por exemplo, em um fórum, quando um usuário publica um tópico, outros usuários podem adicionar comentários sobre o tópico. Estes comentários podem estar relacionados ao tópico, ou não apresentar nenhuma relação com o tópico - por exemplo, um usuário oportunista pode enviar um texto que contém propaganda sobre um assunto qualquer. Além disso, para alguns fóruns não há problema se novas mensagens aparecem entre a leitura do usuário e o envio do novo comentário – uma *postagem suja*. A solução proposta no âmbito deste artigo é baseada em aplicações que manipulam dados com *conteúdo independente*.

Sistemas de armazenamento baseados na replicação de dados de *conteúdo independente* buscam obter bom desempenho sem sacrificar a consistência. Um exemplo concreto[1] é o Facebook, que utiliza replicação primário-secundário [Budhiraja et al. 1993] sobre diferentes regiões geográficas. No caso, as réplicas secundárias visam atender requisições de leitura, de modo a reduzir a latência entre países, por exemplo, de 70ms para 2ms; nesse ínterim, requisições de escrita são redirecionadas para uma réplica primária. Como esta arquitetura (*escreve no primário e lê de um secundário próximo*) poderia levar um usuário a não visualizar a sua própria publicação, uma estratégia é usada para garantir que um escritor sempre visualizará suas próprias publicações: depois que um usuário publica algo, todas as requisições relacionadas à publicação do usuário são direcionadas à réplica primária por algum tempo. Este procedimento proporciona um equilíbrio entre desempenho e consistência, mas se a réplica primária falhar enquanto estiver replicando os dados para as réplicas secundárias, estados inconsistentes podem surgir, já que depois do redirecionamento temporário de requisições de leitura o usuário poderá não visualizar o valor que escreveu. Não obstante, sistemas baseados na replicação primário-secundário não são passíveis de tolerar faltas de natureza arbitrária (i.e., bizantinas) [Lamport et al. 1982].

[1] http://www.podc.org/podc2012-report/

Na literatura alguns trabalhos discorrem sobre armazenamento distribuído com semântica confirmável [Martin et al. 2002] e tolerância a faltas, embora nenhum deles verse especificamente sobre dados de *conteúdo independente*. O DepSky [Bessani et al. 2013] armazena dados em múltiplos provedores de nuvens usando quóruns tradicionais [Malkhi and Reiter 1998], sendo que ele contacta $3f + 1$ provedores de dados em cada passo de comunicação – de acordo com o número mínimo de repositórios necessários para tolerar faltas bizantinas em armazenamento [Martin et al. 2002] – e coleta $2f + 1$ respostas do quórum, onde f é o número máximo de faltas toleradas. No SPANStore [Wu et al. 2013], clientes podem escolher o nível de consistência desejado, o que influencia diretamente no número de provedores nos quais o dado será armazenado. À vista disso, para uma consistência forte, o SPANStore contacta $2f + 1$ provedores e aguarda por $f + 1$ respostas corretas. De outro modo, se apenas a consistência eventual é suficiente, $f + 1$ provedores são contactados, e apenas uma primeira resposta é requerida para concluir o passo de comunicação, enquanto os f restantes provedores são atualizados em segundo plano. O MDStore [Cachin et al. 2014] tolera faltas bizantinas e reduz o número de réplicas de dados de $3f+1$ para $2f+1$. E o Hybris [Dobre et al. 2014] é o único sistema tolerante a faltas bizantinas que tenta armazenar dados em apenas $f + 1$ provedores, utilizando um *timeout* para avaliar o sucesso da operação; se o tempo estipulado é excedido, f provedores adicionais são contactados. É digno de nota que todos estes trabalhos, exceto o Hybris, contactam incondicionalmente f provedores a mais que o número mínimo de confirmações requeridas.

Neste ensejo, com vista para uma melhor utilização dos recursos e um melhor desempenho, este artigo apresenta o RafStore, um protocolo confirmável de armazenamento distribuído que provê consistência regular, disponibilidade de dados e tolerância a faltas bizantinas para aplicações que lidam com dados de *conteúdo independente*. No presente trabalho argumenta-se que é possível prover o armazenamento de dados de *conteúdo independente*, pela contatação de apenas $f + 1$ provedores de dados, para armazenar dados nestes $f + 1$ provedores. Pelo uso da técnica de antecipação de pedidos, o desempenho do RafStore é superior à abordagem tradicional de quóruns, pois o avanço entre fases do protocolo é acelerado. Além disso, os custos são reduzidos pois apenas $f + 1$ provedores de dados são requeridos em cenários previsíveis e de operação correta.

O restante do trabalho está organizado da seguinte forma: a Seção 2 aborda os trabalhos relacionados à proposta; na Seção 3 é definido o modelo de sistema; a Seção 4 detalha a especificação do protocolo proposto; na Seção 5 é apresentada a validação do RafStore; e, a Seção 6 apresenta discussões sobre os resultados. Por fim, a Seção 7 conclui o artigo.

2. Trabalhos relacionados

Na literatura, alguns trabalhos discorrem sobre armazenamento distribuído, tolerância a faltas e níveis de consistência [Bessani et al. 2013, Wu et al. 2013, Cachin et al. 2014, Wang et al. 2012, Cho and Aguilera 2012, Dobre et al. 2014]. O Depsky [Bessani et al. 2013] provê armazenamento distribuído em múltiplas nuvens tolerando faltas bizantinas. No DepSky, a consistência dos dados é garantida a partir de sistema de quóruns [Malkhi and Reiter 1998] e *erasure codes* [Rabin 1989] são usados para reduzir a quantidade de dados armazenados em cada provedor. Apenas um subconjunto de provedores efetivamente armazenam o dado, porém todos os provedores são contactados para realizar o armazenamento, o que resulta em um consumo extra de recursos de rede. Este comportamento segue um limite inferior definido para sistemas de armazenamento

bizantino [Martin et al. 2002], em que $3f + 1$ réplicas são necessárias para garantir que o sistema tolere até f faltas bizantinas. Como exemplo, se for necessário tolerar uma falta, quatro réplicas serão necessárias para compor o sistema.

Uma abordagem mais flexível é realizada pelo SPANStore [Wu et al. 2013], que armazena dados em provedores de nuvens, de modo que cada aplicação pode escolher o nível de consistência desejado. Se uma consistência forte (p. ex.: confirmável) for necessária, o SPANStore contacta $2f+1$ provedores e um quórum de $f+1$ respostas corretas é suficiente para assegurar que o dado está corretamente armazenado. De outro modo, se uma consistência mais fraca for suficiente para a aplicação (p. ex.: consistência eventual), um quórum de $f+1$ provedores é contactado para armazenar os dados, e apenas a primeira resposta correta é suficiente para que a aplicação conclua a operação de escrita; os f provedores restantes serão atualizados de maneira assíncrona, sem confirmações para o cliente de que o dado está corretamente armazenado. Além disso, o SPANStore mantém uma tabela de latências entre os provedores, atualizada a cada hora, com a finalidade de ajudar na recomendação ao usuário sobre em quais provedores o mesmo deve armazenar, a fim de cumprir metas de latência máxima. Note que o SPANStore protege o dado contra faltas de parada, e seu objetivo principal é reduzir custos do dado armazenado.

O protocolo MDStore [Cachin et al. 2014] permite obter uma redução de custos, a partir do armazenamento dos metadados em $3f + 1$ provedores e dos dados em apenas $2f + 1$ provedores, ao invés de $3f + 1$. Embora reduza os custos, aspectos de desempenho não são abordados no MDStore, e a quantidade de $2f + 1$ réplicas de dados é anunciada como um limite inferior para sistemas que armazenam dados e toleram faltas bizantinas. O MDStore usa um serviço de metadados para oferecer armazenamento com semântica linearizável [Herlihy and Wing 1990], enquanto aplicações de *conteúdo independente* geralmente são bem atendidas com consistência regular [Lamport 1978], que é mais fraca do que a semântica linear, mas forte o suficiente para aplicações como redes sociais.

Outro protocolo particularmente interessante é o Gnothi [Wang et al. 2012], o qual replica metadados em todos os provedores enquanto dados são replicados em um subconjunto de provedores preferidos. Quando réplicas tornam-se indisponíveis, réplicas adicionais são contactadas para salvar dados, garantindo que em todas as solicitações ao menos $f + 1$ provedores de dados confirmem a operação de escrita. Um aspecto que é digno de nota é que o Gnothi tolera apenas faltas por parada, mas provê armazenamento linearizável.

O Vivace [Cho and Aguilera 2012] replica dados em provedores remotos e provê consistência forte. Durante congestionamentos de rede, metadados são tornados os menores possíveis e trafegam de maneira priorizada na rede, para permitir a continuidade da operação. O Vivace lida com objetos simples de leitura e escrita, e objetos de leitura, modificação e escrita. Para objetos de leitura e escrita, a linearização é implementada com *write-backs*, que são responsáveis por propagar o último *timestamp* para uma maioria de réplicas. Este passo de comunicação pode ser eliminado com uma otimização denominada paralelização de pedidos: se uma maioria de réplicas não contiver o último *timestamp*, a etapa de *write-back* é disparada de maneira pró-ativa paralelamente à etapa de leitura de dados. Outra otimização no Vivace sugere que o cliente mantenha um histórico de latências dos provedores remotos, a fim de contactar os provedores que possuam menores latências.

E finalmente, o Hybris [Dobre et al. 2014] consiste em um sistema de armazenamento chave-valor que armazena metadados em provedores de nuvens privadas ativas con-

fiáveis e distribui os dados em provedores de nuvens públicas passivas que são até certo ponto (i.e., o limite f de faltas) confiáveis. Os dados são escritos em $f + 1$ provedores no caso normal – sob o atendimento de um *timeout*; e no caso de esgotamento do tempo definido, um total de $2f + 1$ provedores são contactados para efetivar a escrita. Ele tolera faltas de parada nos provedores de metadados e faltas bizantinas nos provedores de dados. Hybris suporta múltiplos escritores e leitores, e é linearizável. A comunicação entre clientes e o serviço de metadados é parcialmente síncrona [Dwork et al. 1988]; entre clientes e nuvens públicas é assíncrona. O serviço de metadados é implementado por máquinas de estado replicadas; provedores de nuvens são eventualmente consistentes e os dados são versionados. Para garantir uma forte consistência dos dados, Hybris permite que clientes inscrevam-se no serviço de metadados para receberem notificações em caso de atualizações de metadados. Isto é útil no caso em que leitores estão lendo dados concorrentemente com escritas.

Uma breve comparação acerca dos trabalhos relacionados demonstra que o MDStore alcança redução de custos, o SPANStore provê uma maneira de saber quais provedores são os mais rápidos e o DepSky obtém melhor desempenho a despeito de um protocolo com maior custo. Hybris tenta reduzir custos utilizando menos provedores de dados, mas torna necessária a utilização de um *timeout*. Neste sentido, à luz dos trabalhos relacionados o protocolo proposto neste trabalho visa: (*i*) reduzir o número de provedores necessários para armazenar os dados, de modo que, para metadados, $3f + 1$ provedores ainda serão necessários; (*ii*) tolerar faltas bizantinas ao tempo que oferece semântica regular; (*iii*) prover maior desempenho sem a utilização de *timeouts* por meio de uma paralelização parcial de requisições. A proposta deste artigo considera latência dos provedores uma informação significativa que o cliente pode considerar para obter melhor desempenho no protocolo de armazenamento. Por fim, o protocolo proposto visa preencher a lacuna de obter simultaneamente, a redução de custos e melhoria de desempenho, bem como pela evocação de réplicas adicionais sob demanda, similar ao que ocorre no Gnothi.

3. Modelo de Sistema

Para a especificação do protocolo proposto, considera-se um ambiente de sistema distribuído assíncrono, em que clientes e provedores de armazenamento estão conectados por uma rede, onde os canais de comunicação são confiáveis (p. ex.: as mensagens enviadas finarão por serem recebidas). Clientes podem ler e escrever dados nos provedores de armazenamento via operações simples como *get* e *put*, respectivamente. Um esquema chave-valor referencia os dados, e na maioria dos casos espera-se que haja apenas um escritor por vez (SWMR – *single writer, multiple readers*) [Lamport 1986]; entretanto, um esquema de baixa contenção [Bessani et al. 2013] pode ser empregado para tolerar múltiplos escritores em provedores de nuvem passivos. Os provedores de armazenamento são entidades passivas, de maneira que não podem executar código ou comunicar-se entre si.

Em relação aos dados manipulados no âmbito do protocolo, estes são considerados como dados de *conteúdo independente*, isto é, para todos os valores escritos, não existe relação de processamento com valores anteriores do dado. Isto significa que uma operação de escrita pode ser executada sem uma leitura prévia do dado para compor o novo valor, de forma semelhante a uma escrita direta, ou *blind-write* [Agrawal and Krishnaswamy 1991]; isto simplifica a especificação e aumenta o desempenho do protocolo. No que concerne às faltas, adota-se um modelo híbrido de falhas, de modo que os clientes podem falhar de maneira arbitrária e exibir comportamento bizantino enquanto estiverem lendo dados –

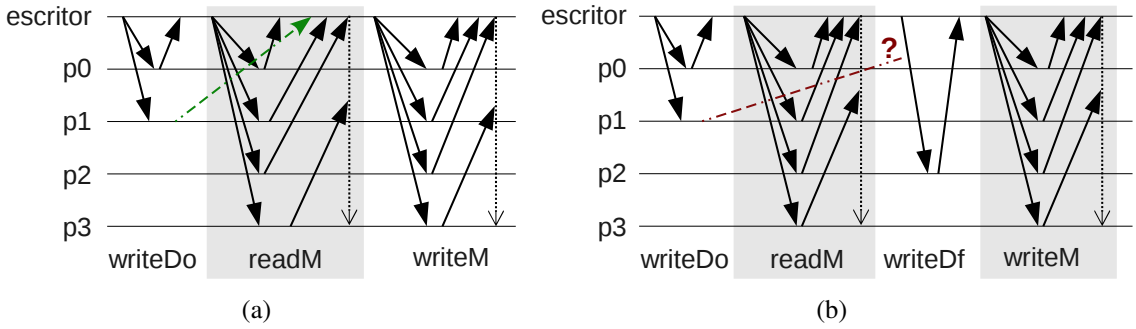

Figura 1. Protocolo RafStore: operação de escrita em (a) execução normal e (b) execução com faltas ou em condições desfavoráveis de rede; configuração com f=1 e n=4.

visto que estes clientes não estarão modificando o estado do sistema; entretanto clientes podem falhar somente por parada quando estiverem escrevendo dados: clientes maliciosos poderiam, por exemplo, escrever valores diferentes, em provedores diferentes, associados a uma mesma versão de dado, ou mesmo exaurir o espaço de versionamento utilizando valores de versão muito grandes, e não é objetivo deste trabalho tratar destas questões[2]. De outro modo, os provedores de armazenamento podem ser corretos ou apresentar comportamento bizantino. O número de provedores responsáveis por armazenar metadados é igual a $3f+1$, sendo que não mais de $f \leq \lceil \frac{n-1}{3} \rceil$ provedores podem falhar simultaneamente, durante uma janela de vulnerabilidade do sistema.

4. O Protocolo RafStore

O RafStore consiste em um protocolo cuja especificação visa prover não apenas um ambiente confiável, mas também otimizado para o armazenamento de dados de *conteúdo independente*. Neste sentido, em condições normais de execução o armazenamento ocorre em $f+1$ provedores de armazenamento, ou, de outro modo, em $2f+1$ provedores de armazenamento quando ocorrem faltas ou quando a rede não apresenta condições favoráveis, conforme explicação detalhada adiante[3]. O RafStore tolera faltas bizantinas a partir do uso de $3f+1$ provedores para armazenar metadados. Para um melhor entendimento, a Figura 1 ilustra o padrão de comunicação adotado para operações de escrita no RafStore.

O funcionamento do protocolo se dá da seguinte maneira, o cliente inicia a escrita de dados (fase *writeDo* da figura 1(a)) por meio de uma solicitação a $f+1$ provedores de dados, para que armazenem o dado. Quando a primeira resposta de um provedor de dados chega ao cliente, a próxima fase (*readM*) é iniciada de maneira especulativa, em que os metadados são buscados junto a provedores que armazenam os metadados – este aspecto consiste na técnica de antecipação de pedidos mencionada na Seção 1. Quando são obtidas respostas suficientes (i.e., $2f+1$) da fase *readM* pelo escritor, uma verificação crucial ocorre: se todas as $f+1$ respostas concernentes à fase *writeDo* foram obtidas dos provedores de dados, os novos metadados podem ser escritos e assim a fase *writeM* é iniciada, conforme ilustrado na figura 1(a). Se ao término da fase *readM* não foi obtido um número suficiente de respostas – conforme ilustra a Figura 1(b) – uma fase adicional nomeada *writeDf* é executada, onde f provedores adicionais de armazenamento de dados são contactados para assegurar que, pelo menos $f+1$ provedores de armazenamento irão armazenar o dado. Note pela Figura 1(b), que tanto a última resposta da fase *writeDo* quanto a resposta da fase *writeDf* são válidas

[2]Existem maneiras de tratar estas questões, como fases adicionais de *write-back* durante operações de leitura para que clientes corretos façam a reparação dos dados [Liskov and Rodrigues 2006].

[3]O texto próximo à Figura 3 detalha estas condições de rede.

Algoritmo 1 Operação de escrita no RafStore

```
1:  procedure WRITE(key, value)
2:      uId := UUID() + "_" + hash(value)                    ▷ identificador único
3:      global c := 0                          ▷ contador de respostas válidas; ver algoritmo 3
4:      global p := ∅                          ▷ conjunto de respostas válidas; ver algoritmo 3
5:      wQuorum(uId, value, 0, f)              ▷ solicita escrita em f + 1 provedores de dados
6:      wait until c ≥ 1
7:      m := readMDData(key, n − f)                                          ▷ ler metadados
8:      if c < (f + 1) then
9:          wQuorum(key, value, f + 1, 2 ∗ f)
10:     wait until c ≥ f + 1
11:     parallel for 0 ≤ i ≤ 2 ∗ f do provider_i.cancelPending()
12:     newV := max(m[i].v : 0 ≤ i ≤ 2 ∗ f) + 1
13:     newMDData := newV + "; " + uId + "; " + p
14:     newMDData := m + newMDData
15:     mt :=⊥                                          ▷ inicializa conjunto de respostas
16:     parallel for 0 ≤ i ≤ n − 1 do
17:         mt[i] := provider_i.put("metadata"+key,newMDData)
18:     wait until |{∀i : mt[i] = "ok"}| ≥ n − f
19:     parallel for 0 ≤ i ≤ n − 1 do provider_i.cancelPending()
```

para completar o quórum de respostas esperadas na fase *writeDf*. Com a estratégia de antecipação de pedidos, é necessário ligar o dado e o metadado visto que o nome do dado armazenado não está relacionado à chave. Para atender a esta necessidade, um identificador único pode ser representado pela composição de um identificador UUID [Leach et al. 2005] aleatório e um *hash* do dado ($MD5$). Neste sentido, uma convenção para representação de metadado no RafStore é através de uma tupla (v,uId,($r_0, r_1, ..., r_f$)), em que v significa a versão do dado, uId é o identificador do dado e r_i é o i-ésimo provedor de dados. Um único arquivo de metadados contém os metadados de todas as versões do dado.

O Algoritmo 1 apresenta detalhadamente a operação de escrita do RafStore. O cliente solicita a escrita do dado com o comando *write(key, value)* (linha 1). O dado é associado a um identificador único (linha 2) e enviado a todos os provedores de dados (linha 5). Quando a primeira resposta dos provedores de dado chega ao cliente (linha 6), o cliente solicita aos provedores de metadados a leitura dos metadados (linha 7). Quando um quórum de respostas sobre a leitura de metadados é obtido, é verificado se todas as escritas de dados foram concluídas (linha 8); se esta verificação detectar que existem escritas de dados ainda não confirmadas, mais provedores de dados são contactados para armazenar o dado (linha 9). De qualquer maneira, a operação de escrita poderá avançar apenas quando pelo menos $f + 1$ provedores de dados confirmarem que o dado foi salvo (linha 10). Para minimizar possíveis desperdícios, eventuais escritas pendentes adicionais são canceladas (linha 11); esta situação ocorre quando provedores de dados adicionais são contactados (linha 9). No que segue, uma nova versão do dado é calculada (linha 12) e o metadado anterior é anexado ao novo metadado (linhas 13 e 14). Por fim, todos os provedores de metadados são contactados para armazenar o metadado (linha 17), de modo que um quórum de $2f + 1$ respostas é suficiente (linha 18) para concluir esta etapa; solicitações de escrita de metadado remanescentes são canceladas para concluir a operação de escrita (linha 19).

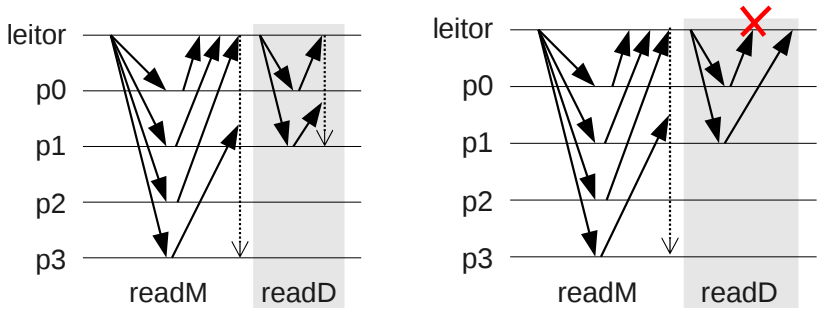

Figura 2. Protocolo RafStore: operação de leitura em (a) execução normal e (b) execução com faltas; configuração com $f = 1$ e $n = 4$.

Algoritmo 2 Operação de leitura no RafStore

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

No que segue, a operação de leitura do protocolo é apresentada na Figura 2, cuja especificação ocorre no Algoritmo 2. O cliente solicita o metadado aos provedores de metadados, a partir do fornecimento da chave (linha 2); quando um número suficiente de respostas chega ao cliente ($2f + 1$), o dado é solicitado aos provedores de dados que contém aquele dado armazenado (linha 4). Quando um dado válido é recebido de um provedor de dados (linha 5), as solicitações remanescentes de leitura de dados são canceladas (linha 6) e o valor obtido dos provedores pode ser entregue ao cliente (linha 7). Procedimentos auxiliares para o funcionamento do protocolo são especificados no Algoritmo 3. O procedimento *wQuorum* (linha 1) contacta um conjunto de provedores p_{ini} a p_{end} para armazenar um valor sob a denominação de uma chave. Na medida em que os provedores confirmam a operação de gravação (linha 4), variáveis globais (linhas 5 e 6) são atualizadas para refletirem o tamanho do quórum de respostas. A função *readMData* (linha 7) solicita aos provedores de metadados o metadado da chave desejada (linha 10). Quando a quantidade necessária de respostas corretas é alcançada (linha 13), solicitações de leitura pendentes são canceladas (linha 14) e o metadado é retornado para o cliente (linha 15).

Um aspecto relevante em se tratando do protocolo proposto, decorre do fato de que é natural supor que a operação de escrita do RafStore alcançará melhor desempenho do que em outras abordagens, visto que a quantidade de provedores de dados contactados é menor. Entretanto, o melhor desempenho estimado é condicionado à escolha de quais provedores atuarão como provedores de dados. Para tanto, considere os provedores (p_0, ..., p_3) e seus respectivos RTT's (*round-trip time*) apresentados na Figura 3. Em ambos os cenários existe uma réplica de dados com alto valor de RTT, beneficiando a tolerância a faltas catastróficas por meio da geo-replicação de longa distância. A Figura 3(a) mostra um cenário onde o provedor de dados mais lento tem RTT igual a 70ms; neste caso, quando a fase *readM* (representada pela sombra cinza) terminar, a resposta do segundo provedor de dados já terá

Algoritmo 3 Operações auxiliares do protocolo

```
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                          ▷ incrementa contador de respostas válidas
6:              p := p ∪ {i}                        ▷ adiciona resposta ao conjunto de respostas válidas

7:  function READMDATA(key, threshold)
8:      m :=⊥
9:      parallel for 0 ≤ i ≤ n − 1 do
10:         t_i := provider_i.get("metadata"+key)
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.cancel_pending()
15:     return m
```

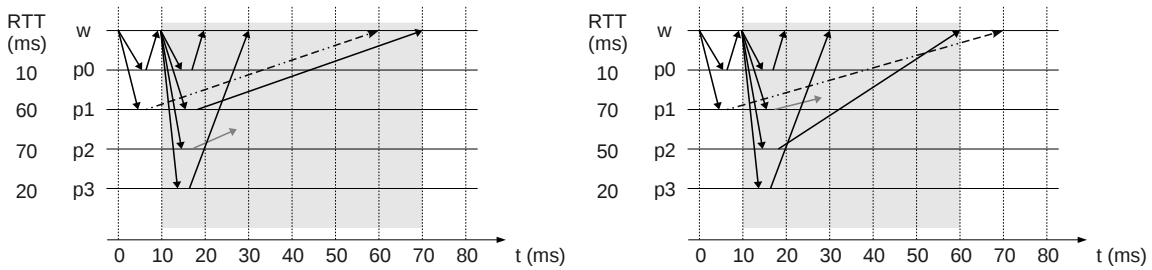

Figura 3. Exemplos de RTT: (a) execução normal e (b) condições desfavoráveis de rede, ou cenário de faltas; configuração com $f = 1$ e n=4.

chegado ao cliente, permitindo que o cliente avance para a fase *writeM* (não desenhada na figura) e conclua a operação de escrita. Entretanto, se o RTT do segundo provedor de dados for o maior entre o conjunto de provedores, a resposta do segundo provedor de dados não terá chegado ao cliente quando a fase *readM* terminar – como mostrado na Figura 3(b) – e outro provedor de dados precisará ser contactado, conforme representado na Figura 1(b).

Análises no comportamento do RafStore revelam que existe uma condição determinante que separa os cenários mostrados nas Figuras 3(a) e 3(b). Esta condição depende de três valores: a, definido na equação 1, o qual representa o provedor de dados mais lento; b, definido na equação 2, que refere-se ao provedor de dados mais rápido; e c, definido na equação 3, que denota o provedor de metadados mais lento no subconjunto dos $2f + 1$ provedores de metadados mais rápidos. RTT é o *round-trip time*, dp_i é o i-ésimo provedor de dados, mdp_i é o i-ésimo provedor de metadados e $subset(set, k)$ retorna os primeiros k elementos de set. O termo $ordered$ significa que o item anexo está ordenado. A condição que garante um melhor desempenho no protocolo RafStore é que a precisa ser menor ou igual ao valor $(b + c)$. Esta relação está expressa textualmente na Condição 1.

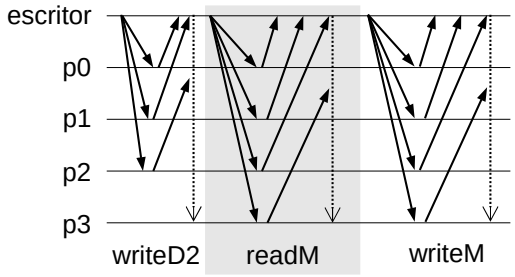

Figura 4. Modo de operação tradicional no RafStore: útil quando não há informações sobre latências dos provedores.

$$a = max(RTT_{dp_i}), 0 < i < n_{dp} - 1 \quad (1)$$

$$b = min(RTT_{dp_i}), 0 < i < n_{dp} - 1 \quad (2)$$

$$c = max(subset(orderedRTT_{mdp_j}, 2f + 1)), 0 < j < n_{mdp} - 1 \quad (3)$$

Condição 1: *a escolha dos provedores deve obedecer a regra* $a < (b + c)$. *Em outras palavras, o provedor de dados mais lento precisa ser mais lento ou tão rápido quanto a soma do provedor de dados mais rápido e o provedor de metadados mais lento no subgrupo dos* $2f + 1$ *provedores de metadados mais rápidos.*

Quando não é possível atender a Condição 1, pode-se proceder de um modo mais tradicional. A Figura 4 ilustra uma operação com quóruns tradicionais, onde um quórum de $2f+1$ provedores de dados é contactado para armazenar dados (fase *writeD2*). Quando $f+1$ respostas dos provedores de metadados chegam ao cliente, solicitações remanescentes são canceladas e as próximas duas fases (*readM* e *writeM*) são iniciadas para realizar a leitura e a escrita de metadados. O algoritmo deste modo de operação tradicional foi omitido por questões de espaço, mas é possível derivá-lo a partir dos algoritmos já apresentados.

O conhecimento prévio do RTT dos provedores é útil para orientar as escolhas de quais provedores serão responsáveis pelo armazenamento de dados e quais armazenarão apenas metadados. Algumas recomendações sugerem uma arquitetura que ofereça as condições ideais para o RafStore; esta arquitetura é apresentada pela Figura 5:

- f provedores devem ser do tipo I: estes provedores devem estar *próximos* ao cliente, para obter melhor **desempenho**. Estes provedores são replicados em quantidade igual ao número de faltas toleradas no sistema, e podem ser nuvens menos custosas ou nuvens privadas. Dados e metadados serão armazenados nestes provedores.
- ao menos um provedor deve ser do tipo II: este provedor deverá ser robusto e hospedar uma cópia de dados; deve estar *distante* para garantir a **sobrevivência** do dado.
- f provedores devem ser do tipo III: estes provedores devem estar tão distantes quanto o provedor de dados mais distante, a fim de manter válida a relação $a < (b + c)$. Além disso, estes provedores são responsáveis principalmente pelo armazenamento de metadados, mas podem armazenar dados em caso de faltas.
- não há recomendações especiais para os f provedores remanescentes, denominados do tipo IV. Estes provedores armazenam apenas metadados.

5. Experimento

De acordo com as características apresentadas na Seção 4, o RafStore deve alcançar melhor desempenho do que protocolos baseados em quóruns tradicionais quando as condições de

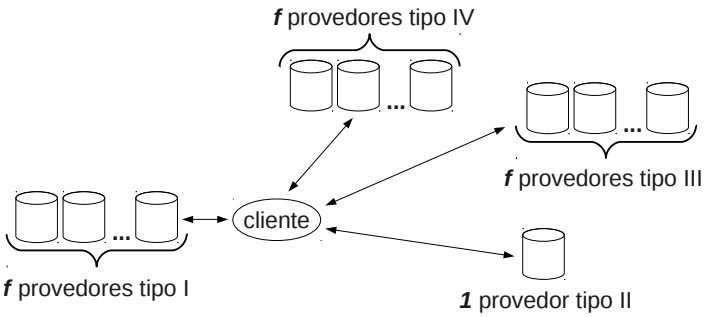

Figura 5. Clientes e provedores: arquitetura sugerida para o uso do protocolo RafStore.

rede (latências) forem favoráveis. Esta afirmação remete à primeira hipótese:

Hipótese 1: *se as latências dos provedores de dados obedecerem à Condição 1, o protocolo RafStore alcançará desempenho melhor que os protocolos de quóruns tradicionais.*

Outro ponto a ser investigado é se o tamanho do dado interfere na diferença do desempenho entre os protocolos considerados. Esta ponderação é intuitivamente verdadeira, de modo que a literatura evidencia [Padilha and Pedone 2011] que existe um valor limite até o qual operações de escrita e leitura tem desempenhos similares, e a partir do qual a diferença de desempenho será maior. Esta asserção remete à segunda hipótese:

Hipótese 2: *na medida em que o tamanho do dado aumenta, a diferença de desempenho entre os protocolos comparados também aumenta.*

Um experimento foi conduzido para avaliar as hipóteses apresentadas. O tempo de resposta percebido pelo cliente foi a informação coletada. Os testes foram executados em um sistema distribuído, com quatro computadores representando os provedores e um quinto computador representando o cliente. Neste cenário, de acordo com a regra $3f + 1$, uma falta bizantina é tolerada ($f = 1$). Os provedores foram executados em computadores com a configuração Intel i7 3.5Ghz, QuadCore, cache L3 8MB, 12GB RAM, 1TB HD. O cliente foi executado sobre um computador com configuração Intel i7 3.07GHz, QuadCore, cache L3 8MB, 16GB RAM, 2TB HD. Todas as máquinas executaram o sistema operacional Debian 7.4 Wheeze, 64 bits, kernel 3.2.54-2, com JVM Oracle JDK 1.7. Clientes e provedores foram interconectados por uma rede Ethernet 10/100 MBits.

Para verificar a Hipótese 1, as latências dos provedores serão definidas com valores de acordo com a Condição 1. Os valores escolhidos foram (0, 161, 308, 181) para os provedores p_0, ..., p_3, respectivamente. Estes valores foram obtidos de um trabalho relacionado [Terry et al. 2013] e representam latências para um cliente na China e provedores em *data centers* no Oeste dos Estados Unidos, Inglaterra e Índia, respectivamente.

Note que o RafStore pode operar em dois modos: o caso quando as latências obedecem a Condição 1 e apenas $f + 1$ provedores são contactados para armazenar dados (uso da técnica de antecipação de pedidos), ou um caso mais tradicional, onde $2f + 1$ provedores de dados são contactados. Estas duas abordagens buscam obter $f + 1$ confirmações de que provedores de dados realizaram o armazenamento. Neste sentido, o RafStore será comparado nestes dois modos de operação, em relação aos protocolos especificados no DepSky [Bessani et al. 2013]. O tamanho do dado será variado de 1KB até 1MB, por um fator multiplicativo de 4, com a finalidade de verificar a Hipótese 2. Estes valores cobrem desde o tamanho de uma frase textual grande até o tamanho típico de uma fotografia. Para simular o RTT, os computadores provedores foram configurados com emulador de rede

WAN[4]. A unidade de tempo considerada foi *ms*, provedores de metadados são definidos como p_0, ..., p_3 e provedores de dados são definidos como p_0 e p_1. A ordem de execução dos pedidos, para os diferentes tamanhos e protocolos, foi sorteada para prover aleatoriedade ao experimento. Vinte replicações do experimento foram realizadas; os dados coletados ajustam-se bem a uma distribuição normal e estão disponíveis em hylson.com/rafstore.

É digno de nota que o RafStore foi implementado em Java. Para uma comparação mais justa, os protocolos do DepSky foram reimplementados, sem otimizações, sobre a mesma base de código do RafStore. O DepSky-A realiza replicação integral de dados, enquanto o DepSky-CA requer o uso de *erasure codes* [Rabin 1989], que foram utilizados da biblioteca JEC, disponível no código do DepSky[5]. Aspectos de privacidade não foram implementados, visto que o objetivo principal do experimento é comparar o desempenho observando o número de provedores de dados e o tamanho do dado armazenado nos provedores de dados. No DepSky-CA, o uso de *erasure codes* reduz o tamanho dos dados armazenados em cada provedor de dados. O RafStore está disponível em hylson.com/rafstore.

6. Avaliação

A realização do experimento descrito na Seção 5 nos permitiu verificar alguns aspectos em relação ao RafStore. A Figura 6 apresenta as latências de operações de escrita para os diferentes protocolos comparados: DepSky-A, DepSky-CA, RafStore usando a abordagem de pedidos antecipados e RafStore usando a abordagem tradicional. Nota-se que apenas o RafStore com a abordagem de pedidos antecipados obtém melhor desempenho do que os outros métodos, para todos os tamanhos de dados considerados. Com relação à Hipótese 1, os resultados não permitem concluir que o RafStore supera, em desempenho, os protocolos de quóruns tradicionais, mesmo quando as latências dos provedores atendem à Condição 1. Entretanto, é possível afirmar que o RafStore que utiliza pedidos antecipados apresenta melhores resultados que os demais protocolos, para todos os tamanhos de dados considerados.

Sobre a relação entre o tamanho do dado e a variação de desempenho, é possível observar, por exemplo, que a diferença de desempenho entre os protocolos DepSky-A e DepSky-CA aumenta conforme o dado aumenta; porém, essa relação não ocorre continuamente entre os protocolos DepSky-CA e RafStore antecipado, pois a diferença aumenta quando o dado cresce de 64KB para 256KB, mas diminui na transição de 256KB para 1MB. Assim, considerando a Hipótese 2, não é possível afirmar que a diferença de desempenho entre os protocolos comparados aumenta de acordo com o aumento do tamanho do dado. Por fim, destaca-se que o RafStore com o uso de pedidos antecipados alcança uma latência em torno de 30% menor que o protocolo DepSky-CA, para dados maiores que 64KB; para dados com 1MB de tamanho, essa diferença diminui para cerca de 21%.

Um comportamento interessante pode ser percebido na Figura 6: o desempenho do RafStore no modo de operação tradicional comparado com o protocolo DepSky-CA. Para dados de tamanho 256KB, o RafStore possui melhor desempenho do que o DepSky-CA. Para dados de tamanho 1024KB, o DepSky-CA possui melhor desempenho do que o RafStore. Essa inversão pode ser explicada pela fragmentação de dados realizada pelo DepSky-CA: visto que fragmentos são salvos mais rapidamente do que dados integrais (pois são menores), é evidente que o DepSky-CA obtém um melhor desempenho em relação ao

[4]http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[5]http://code.google.com/p/depsky/

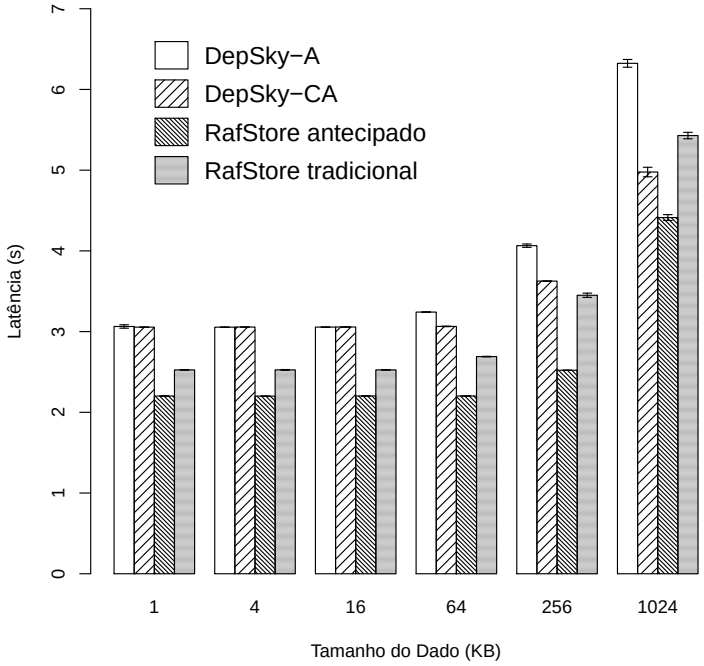

Figura 6. Latências para os métodos comparados, com variação no tamanho do dado.

RafStore, quando o tamanho dos dados aumenta. Entretanto, existem outras questões sobre o tamanho de fragmentos em relação ao tamanho integral de dados. Um micro-experimento foi conduzido utilizando o código do DepSky para comparar o tamanho dos fragmentos gerados pelo DepSky-CA, na medida em que o tamanho do dado aumenta. De acordo com os dados da Tabela 1 existe um ponto a partir do qual o tamanho do dado armazenado como fragmento (DepSky-CA) é menor do que o tamanho integral do dado. Uma discussão mais aprofundada sobre tais aspectos será desenvolvida em trabalhos futuros.

Tabela 1. Tamanho do fragmento, em bytes, para o DepSky-(A) e DepSky-(CA).

DepSky-(A)	1	512	1024	1536	2048	2560	3072	3584
DepSky-(CA)	685	941	1197	1453	1709	1965	2221	2477

7. Conclusão

Este artigo apresentou o RafStore, um protocolo de armazenamento distribuído que possui desempenho melhor que trabalhos anteriores – até 30% em alguns casos – e consome menos recursos – dispensa o uso de um provedor de nuvens para armazenar dados – se algumas condições de rede, como latência entre o cliente e os provedores, puderem ser observadas, e a aplicação fizer uso de dados de conteúdo independente. Não obstante, o protocolo do RafStore foi especificado para prover não somente geo-replicação, mas também para tolerar faltas bizantinas. Os clientes precisam escolher quais provedores e *data centers* desejam usar para armazenar dados em nuvens e o RafStore pode orientar esta escolha se desempenho e economia de recursos forem prioridades do cliente.

Agradecimentos

Miguel Correia é bolsista CAPES/Brasil (projeto LEAD CLOUDS). O trabalho foi parcialmente financiado pela FCT UID/CEC/50021/2013 e pelo CNPq 455303/2014-2.

Referências

Agrawal, D. and Krishnaswamy, V. (1991). Using multiversion data for non-interfering execution of write-only transactions. *SIGMOD Record*, 20(2):98–107.

Bessani, A., Correia, M., Quaresma, B., André, F., and Sousa, P. (2013). DepSky: Dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage*, 9(4):12:1–12:33.

Brewer, E. (2012). CAP twelve years later: How the "rules"have changed. *Computer*, 45(2):23–29.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). The primary-backup approach. In Mullender, S., editor, *Distributed systems*, pages 199–216. 2 edition.

Cachin, C., Dobre, D., and Vukoli, M. (2014). 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*, pages 1–17.

Cho, B. and Aguilera, M. K. (2012). Surviving congestion in geo-distributed storage systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 439–451.

Dobre, D., Viotti, P., and Vukolić, M. (2014). Hybris: Robust hybrid cloud storage. In *Proceedings of the 5th annual ACM Symposium on Cloud Computing*, pages 1–14.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Lamport, L. (1986). On interprocess communication. *Distributed Computing*, 1(2):77–101.

Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401.

Leach, P., Mealling, M., and Salz, R. (2005). A universally unique identifier (UUID) URN namespace (RFC 4122). IETF Request For Comments.

Liskov, B. and Rodrigues, R. (2006). Tolerating byzantine faulty clients in a quorum system. In *the 26th IEEE International Conference on Distributed Computing Systems*, pages 34–43.

Malkhi, D. and Reiter, M. (1998). Byzantine quorum systems. *Distributed Computing*, 11:203–213.

Martin, J.-P., Alvisi, L., and Dahlin, M. (2002). Minimal byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325.

NIST (2011). National institute of standards and technology definition of cloud computing. Disponível em `http://www.nist.gov/itl/cloud/`.

Padilha, R. and Pedone, F. (2011). Belisarius: BFT storage with confidentiality. In *Proceedings of the 10th IEEE International Symposium on Network Computing and Applications*, pages 9–16.

Rabin, M. O. (1989). Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348.

Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., and Abu-Libdeh, H. (2013). Consistency-based service level agreements for cloud storage. In *Proceedings of the 24th Symposium on Operating Systems Principles*, pages 309–324.

Wang, Y., Alvisi, L., and Dahlin, M. (2012). Gnothi: Separating data and metadata for efficient and available storage replication. In *the USENIX Annual Technical Conference*, pages 413–424.

Wu, Z., Butkiewicz, M., Perkins, D., Katz-Bassett, E., and Madhyastha, H. V. (2013). Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th Symposium on Operating Systems Principles*, pages 292–308.