Capítulo 1

Tolerância a Faltas e Intrusões para Sistemas de Armazenamento de Dados em Nuvens Computacionais

Hylson Vescovi Netto, Lau Cheuk Lung, Rick Lopes de Souza

Abstract

Distributed storage is an area already explored in the literature. The increasing availability of storage providers in the Internet - the clouds - creates opportunities for research on problems that arise with this global infrastructure, that has high geographical distribution and on-demand costs. Systems that works in this environment have to maintain data consistency, run over partitioned networks and achieve good performance. Systems that need to be reliable have to tolerate fault and intrusions. This minicourse presents concepts and techniques that are used in commercial cloud providers, also in scientific works that consider data storage in clouds.

Resumo

Armazenamento distribuído é uma área bastante explorada na literatura. A crescente disponibilidade de provedores de armazenamento na Internet - as nuvens - criou oportunidades de pesquisa para resolver desafios que surgiram com essa infraestrutura global, que possui alta distribuição geográfica e custos sob demanda. Sistemas que operam nesse ambiente precisam manter a consistência dos dados, trabalhar sob particionamento de rede e alcançar desempenho satisfatório. Sistemas que requerem alta confiabilidade precisam tolerar faltas e intrusões. Este minicurso apresenta conceitos básicos e técnicas que são utilizadas por provedores de nuvens comerciais, bem como por trabalhos científicos que apresentam soluções para o armazenamento de dados em nuvens.

1.1. Introdução

Armazenamento distribuído de dados é um assunto bastante explorado pela literatura. Desde registradores compartilhados [2] a sistemas de arquivo em rede, protocolos tem

sido desenvolvidos para adicionar capacidades de tolerar faltas, tornando assim os sistemas mais confiáveis. Trabalhos recentes atuam sob o novo cenário criado pela disponibilização de serviços de armazenamento em nuvens, onde existem benefícios como a utilização sob demanda (custos sob demanda) e a elasticidade de recursos. As nuvens de armazenamento podem ser categorizadas em passivas ou ativas. As nuvens passivas não tem capacidade de executar código, assemelhando-se em muito aos registradores compartilhados: possuem operações básicas de leitura e escrita. Como exemplo de nuvens passivas temos a Amazon S3[1], a Microsoft Azure Storage[2], o RackSpace Cloud Block Storage[3] e a Google Cloud Storage[4]. As nuvens ativas oferecem capacidade de processamento, onde é possível, por exemplo, realizar comunicação entre nuvens e analisar ou validar comandos antes de sua execução. Exemplos de nuvens que disponibilizam recursos de processamento são a Google App Engine[5] e a Amazon EC2[6]. A disponibilidade de armazenamento em nuvens possibilitou uma nova fase de pesquisa em armazenamento distribuído: protocolos como o DepSky [5] e o SCFS [6] tornam possível o armazenamento de dados nas nuvens de maneira confiável.

Os provedores de armazenamento de dados em nuvens disponibilizam várias formas de utilizar os serviços. A aplicação cliente pode utilizar bancos de dados relacionais, por meio de comandos SQL; pode especificar um nome (uma chave) e um valor, em um banco de dados NoSQL; ou pode mapear dados como um sistema de arquivos remoto. Cada tipo de dado possui características e custos diferentes. Estas formas de utilização de serviço referem-se ao uso de uma nuvem em modo SaaS (*Software as a Service*), por meio de API's. Uma API (*Application Programming Interface*) é uma maneira padronizada que permite programas acessarem serviços. É possível também utilizar nuvens como IaaS - *Infrastructure as a Service*, de modo que seja possível instalar o software desejado, como um gerenciador de banco de dados, e assim interagir com a nuvem por meio de protocolos mais específicos. Entretanto, o SaaS é mais apropriado no caso de utilização em grande escala, oferecendo benefícios como elasticidade e custos sob demanda. A elasticidade se refere ao fato de o próprio provedor de nuvem controlar a criação ou remoção de recursos em função da demanda da aplicação.

A diversidade de características existente entre os provedores de nuvens públicas traz consigo um benefício no que se refere à capacidade de tolerar faltas [3]. Diferentes provedores de nuvens são administrados por diferentes pessoas, com diferentes estratégias, em ambientes de rede variados. Assim, a utilização de diferentes provedores de nuvens pode aumentar a tolerância a faltas. Há trabalhos que tem por objetivo realizar o armazenamento de dados em mais de um provedor de nuvem, visando alcançar benefícios como disponibilidade total. No artigo de Bessani et al. [5], experimentos apontam que o uso de múltiplos provedores de nuvens são a única forma de garantir 100% de disponibilidade de acesso a dados no ambiente da Internet. Essa disponibilidade total inclui tolerar faltas de acesso de provedores, ainda que em poucos momentos. Mesmo os raros

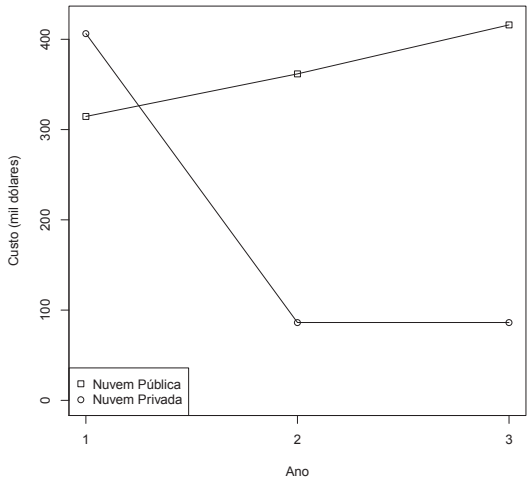
[1] http://aws.amazon.com/s3/
[2] http://azure.microsoft.com/en-us/services/storage/
[3] http://www.rackspace.com/cloud/block-storage/
[4] https://cloud.google.com/products/cloud-storage/
[5] https://developers.google.com/appengine/
[6] http://aws.amazon.com/ec2/

instantes de inacessibilidade de serviços tornam-se significativos quando o serviço é escalado em grande proporção, em função do acesso de muitos clientes. Por isso tornam-se importantes protocolos que garantam acesso contínuo aos dados.

Apesar da grande disponibilidade de provedores de serviços em nuvens públicas da atualidade, ainda existe alguma resistência em se utilizar exclusivamente nuvens públicas para, por exemplo, armazenar dados sigilosos. Há uma grande tendência em manter informações como metadados de arquivos ou controles de acesso em servidores internos às instituições; nesse caso, são apropriadas as nuvens privadas, que também possuem características como elasticidade, mas funcionam de forma dedicada a uma organização, não sendo compartilhada. Nuvens híbridas consideram a utilização simultânea de nuvens públicas e privadas. Existem provedores que já sugerem essa composição de nuvens híbridas[7], enquanto questões de confidencialidade e privacidade ainda estão sendo fonte de problemas ainda nos dias atuais - o recente caso do vazamento de fotos íntimas de celebridades[8] é mais um exemplo de consequências do simples envio de dados para as nuvens. Além de questões de segurança, o custo para criar e manter infraestruturas computacionais também pode ser um fator relativo na decisão de utilizar sistemas e dados em nuvens públicas ou privadas. Um estudo comparativo [18] apontou a relação entre a utilização de um mesmo sistema em nuvem pública e privada. Conforme mostra a figura 1.1, nuvens privadas podem fornecer um melhor benefício ao longo do tempo, apesar do maior custo inicial.

Ano	Nuvem Pública ($)	Nuvem Privada ($)
1	314.525	406.395
2	361.703	86.248
3	415.959	86.248

(a)



(b)

Figura 1.1: Valores gastos em nuvens, para uma mesma aplicação. Nuvens privadas podem apresentar melhor benefício após determinado período [18].

Uma das grandes ameaças à computação em nuvem é o controle que os provedores deste tipo de tecnologia detém sobre as comunicações e dados dos usuários com as empresas contratantes. Recentemente os Estados Unidos, por meio da agência *NSA*, espionaram milhões de ligações, comunicações e dados armazenados nos mais variados serviços de nuvem. Conclui-se então que esse tipo de mecanismo dá muitos poderes às

[7]http://www.emc.com/microsites/cio/articles/goulden/index.htm?pid=hp-goulden-article-180714
[8]http://www.latimes.com/business/la-fi-celebrity-hack-20140901-story.html

empresas de telecomunicações e provedores de nuvem que monitoram as atividades dos seus usuários. As empresas Google, Microsoft, Apple, Facebook, entre outras [19], são exemplos de empresas que tem parceria com o governo americano e que fornecem dados para espionagem. A seguir, serão elencados diversos aspectos referentes à segurança na computação em nuvens.

Localização dos dados: normalmente, os servidores dos provedores de nuvem estão localizados em outros países, como os Estados Unidos, no qual possuem rígidos controles de dados para controles internos e fazem com que os dados das corporações de outras empresas fiquem expostos a interesses destes países [20]. O usuário final normalmente não sabe onde estão armazenados seus dados e, com isso, terá dificuldades de buscar por seus direitos caso seus dados sejam expostos devido às diferentes leis vigentes em cada país.

Segurança da rede: na computação em nuvem os dados são obtidos das estações locais dos usuários, processados e armazenados na nuvem. Todos os dados que trafegam entre o usuário e a nuvem precisam ser cifrados com o objetivo de proteger os dados com conteúdos sensíveis. Para que isso seja feito, deve-se utilizar mecanismos de cifragem como o *Secure Socket Layer* (SSL). Estes provedores de serviço de nuvem devem estar protegidos contra ataques advindos da rede, como por exemplo, ataque do homem no meio, *IP spoofing* e captura de pacotes.

Segregação de Dados: uma das principais características da computação em nuvem são os múltiplos clientes que acessam a mesma instância do software na nuvem. Como resultado deste uso compartilhado, dados de diversos usuários serão armazenados no mesmo servidor. Com isso, ataques de invasão ao espaço de outros usuários tornam-se possíveis pelo não-isolamento seguro dos dados. Estes tipos de ataques podem ser realizados inserindo códigos maliciosos no provedor de serviços. Se o provedor executá-los sem uma verificação, existe um alto potencial de sucesso no ataque. O provedor de serviços deve prover mecanismos de separação segura entre os dados dos usuários.

Autenticação e Autorização: grande parte das empresas que utilizam a computação em nuvem armazenam as informações e credenciais de seus funcionários nesses provedores de serviços. Com isto, os dados ficam fora da proteção da empresa, tornando assim os dados dos funcionários vulneráveis a possíveis ataques. Uma das alternativas seria a utilização de parte destes serviços para os servidores internos da empresa. Desta forma, alguns dados sensíveis e autenticações poderiam estar protegidos física e logicamente.

Confidencialidade dos Dados: Quando indivíduos, empresas ou governos utilizam a computação em nuvem para armazenar seus dados, questões sobre a privacidade e confidencialidade são discutidas. Pela natureza distribuída da nuvem, os dados dessas organizações são compartilhadas em diversos servidores que são de propriedade externa e que são operados por terceiros. Existem diversos serviços de nuvem que armazenam dados sensíveis, como sistemas governamentais, de saúde, judiciários e sistemas pessoais. Algumas das implicações relacionadas a privacidade são:

- A preocupação com o sigilo dos dados não se restringe a dados pessoais, mas também de empresas e órgãos governamentais;

- Os direitos sobre a privacidade e confidencialidade dos dados muda conforme os tipos e categorias de dados que o provedor de conteúdo fornece para a nuvem;
- O armazenamento de dados pessoais e empresariais feito em nuvens pode gerar consequências negativas para a imagem com relação à privacidade dos dados;
- A localização dos servidores tem influência direta sobre os efeitos da privacidade e obrigações legais para aqueles provedores que fornecem serviços e armazenam dados;
- As informações armazenadas na nuvem podem estar localizadas em mais de um local, podendo assim estar sob jurisdições diferentes;
- Algumas leis podem obrigar os provedores de nuvem a fornecer dados de seus usuários alegando investigações sobre crimes ou outros motivos;

Falhas: servidores dos provedores de nuvem podem falhar por diversas razões e parar de oferecer seus serviços. Um sistema que preza pela segurança deve ser tolerante a faltas, mantendo o funcionamento mesmo que em menor capacidade. Essa ameaça é contínua nos provedores de nuvem, haja visto que o usuário não detém controle sobre o funcionamento total do serviço contratado. Por esse motivo, deve-se utilizar uma arquitetura diferenciada, que possa manter o funcionamento do sistema em casos de falha. Uma das alternativas que pode-se adotar é a utilização de múltiplos provedores de nuvem, garantindo assim uma maior heterogeneidade no fornecimento dos serviços e minimizando o impacto de possíveis falhas dos sistemas.

Ao final deste minicurso, espera-se que o leitor possa compreender os conceitos e técnicas que permitem a utilização de sistemas de armazenamento em nuvens, tanto em artigos científicos, quanto em descrições de produtos comerciais. A seguir temos um pequeno exemplo de texto que envolve os conceitos abordados neste minicurso[9]. O trecho transcrito refere-se ao funcionamento do Facebook[10], e foi apresentado por Harry Li em 2012, no evento PODC[11]:

> Facebook tem mais de um milhão de usuários, e usa *cache* distribuído em memória capaz de processar um bilhão de operações por segundo. Sua prioridade máxima é garantir uma experiência rápida, confiável e consistente ao usuário. Eles usam uma arquitetura geo-distribuída de regiões mestre e escravo, onde cada região tem seu próprio conteúdo web, *cache* e *clusters* de banco de dados. Uma região escravo atende apenas pedidos de leitura de dados e usa MySQL com replicação para sincronizar seus bancos de dados com o primário, reduzindo uma latência de acesso entre países de 70ms para 2ms. Embora o Facebook garanta apenas consistência eventual, eles usam um pequeno truque para garantir que um usuário possa ler suas próprias escritas: se o usuário recentemente atualizou dados, os consecutivos pedidos deste usuário são redirecionados para a região primária durante algum tempo.

[9]Caso o leitor não entenda o texto, este trecho pode ser lido novamente após o minicurso, sob a expectativa de sua compreensão.

[10]http://www.facebook.com

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

1.2. Replicação e Tolerância a faltas

Tolerância a faltas é uma área de pesquisa que tem por objetivo tornar um sistema capaz de continuar funcionando a despeito de problemas que possam ocorrer no mesmo, de forma transparente ao usuário. Normalmente é estabelecido um limiar f que significa o número de faltas que podem ser toleradas no sistema; os recursos necessários para manter esse funcionamento transparente do sistema geralmente são definidos em função desse parâmetro f. Consequentemente, quanto mais faltas deseja-se que o sistema tolere, mais recursos são necessários.

Dentre as possíveis faltas que podem ocorrer em um sistema, destaca-se a falta denominada *crash*, também chamada falta de parada, onde um elemento do sistema, por algum motivo, interrompe definitivamente a interação com o usuário. Uma questão conhecida e bastante relevante em sistemas assíncronos é a dificuldade em diferenciar se essa falta de interação deve-se a uma falha total no elemento ou a uma eventual lentidão na rede. Em sistemas síncronos essa detecção é baseada em restrições temporais que orientam a verificação sobre o funcionamento do elemento: após um certo tempo sem interagir (*timeout*), considera-se que o elemento não funciona mais - sofreu um *crash*. Para tolerar este tipo de falta, redundância é a técnica mais utilizada: a falha de um elemento não compromete o sistema porque existem outros elementos que podem realizar as atividades necessárias.

Quando os componentes do sistema distribuído apresentam desvios de funcionamento além da simples interrupção de interação, é necessário utilizar técnicas mais elaboradas para garantir o funcionamento do sistema: é preciso tolerar faltas arbitrárias, ou bizantinas. Essas faltas podem ocorrer por quaisquer motivos, e sua detecção geralmente envolve critérios de maioria: os elementos são replicados, sendo necessário que mais da metade dos elementos apresente respostas iguais para que o resultado seja confiável. Alguns exemplos de faltas bizantinas são: modificações no sistema por invasores, substituição de respostas por valores antigos, interrupção de funcionamento de elementos (*crash*) e acessos indevidos por operadores internos do sistema. A partir de agora, será utilizado o termo BFT (*Byzantine Fault Tolerance*) para referenciar a expressão ”tolerância a faltas bizantinas”.

Replicação é a maneira mais prática de obter redundância em sistemas. A criação de réplicas implica na existência de mais de um provedor de serviço, trazendo vantagens para o cliente: *i*) o desempenho do sistema poderá ser melhor devido à utilização de réplicas que estiverem mais próximas ao cliente, resultando assim em um tempo de resposta menor; *ii*) a disponibilidade do serviço replicado é maior, pois a existência do serviço em diversos locais permite um maior conjunto de opções ao cliente, sobre qual réplica escolher para utilizar o serviço; *iii*) em sistemas replicados, geralmente tolera-se a perda ou o mau funcionamento de uma ou mais réplicas, agregando assim a tolerância a faltas como uma característica do sistema. Naturalmente, existem custos decorrentes da replicação de serviços ou recursos. Porém, dados os benefícios apresentados, considera-se a replicação como uma das maneiras mais efetivas para o provimento de tolerância a faltas em sistemas computacionais. O armazenamento de dados em provedores de nuvens também utiliza-se da replicação para aumentar a durabilidade do dado, bem como a disponibilidade, o desempenho no tempo de resposta, entre outras características.

Dentre as diversas maneiras de realizar a replicação, duas são mais conhecidas e utilizadas: a replicação passiva e a replicação ativa. A replicação passiva consiste em utilizar uma estratégia do tipo primário-secundário [10], de maneira que o cliente interage apenas com uma réplica principal, e esta réplica atualiza as outras réplicas secundárias. O termo ”passivo” refere-se ao fato de o cliente interagir apenas com uma réplica principal, e o sistema atualizar as demais réplicas passivamente, no sentido de não exigir a participação ativa do cliente para tal operação (o cliente não precisa solicitar que cada réplica secundária seja atualizada). A atualização entre a réplica primária e as secundárias pode ocorrer em tempo síncrono ou assíncrono, dependendo dos requisitos do sistema. Caso a atualização das réplicas secundárias seja síncrona, o cliente deverá aguardar até que as réplicas secundárias sejam atualizadas. Se a atualização das réplicas secundárias for assíncrona, o tempo de resposta do sistema será igual ao tempo de resposta da réplica primária, que após enviar a resposta ao cliente, poderá iniciar a atualização das demais réplicas. A figura 1.2a ilustra um sistema com replicação passiva: clientes acessam uma réplica primária, que atualiza as réplicas secundárias. Na replicação ativa (figura 1.2b), o cliente interage diretamente com todas as réplicas (ou um subconjunto delas). Pode-se aguardar pela resposta de todas as réplicas, ou de um subconjunto de respostas, dependendo do modelo do sistema. As setas possuem tracejado diferenciado para explicitar a comunicação de cada cliente com as réplicas.

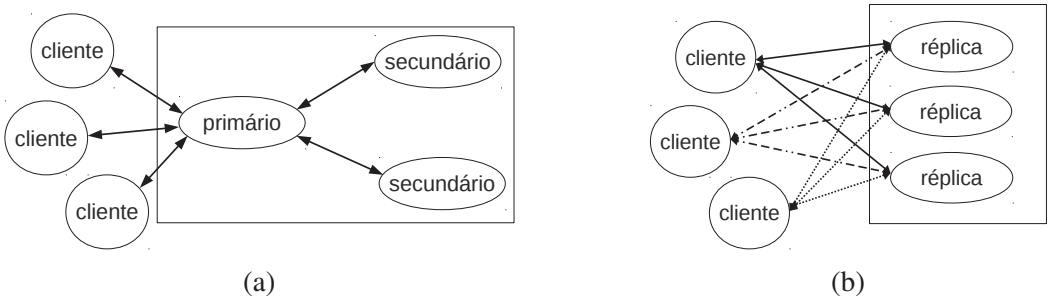


Figura 1.2: Replicação (a) passiva e (b) ativa [13].

Considerando o aspecto de tolerância a faltas, a replicação passiva é mais suscetível a falhas do que a replicação ativa, pois o cliente interage apenas com a réplica primária. Quando esta réplica primária sofre faltas de parada, o cliente inicia o contato com uma réplica secundária, que será considerada pelo cliente a nova réplica primária. Em geral esta verificação de falta de parada é realizada pelo cliente com a utilização de um temporizador (*timeout*). Dessa maneira, consideramos que na replicação passiva sob condições síncronas de tempo, tolera-se faltas de parada utilizando $f+1$ réplicas, onde f é o número máximo de faltas toleradas. Considerando, por exemplo, $f = 1$, temos duas réplicas, uma sendo primária e outra secundária. Quando a primária deixa de responder ao cliente, após um período de tempo específico (*timeout*), considera-se que ocorreu uma falta no sistema, sendo essa falta tolerada devido à presença da réplica secundária, que assume a função de responder ao cliente. Na replicação passiva não é possível tolerar faltas bizantinas, pois a réplica primária poderia, por exemplo, realizar o armazenamento primário incorretamente, mas responder ao cliente que a operação ocorreu com sucesso. Ao mesmo tempo, essa réplica primária poderia não encaminhar a solicitação à réplica secundária, deixando

a réplica secundária desatualizada; quando a réplica primária manifestar comportamento incorreto, o cliente acionaria a réplica secundária, que estaria desatualizada e não poderia responder corretamente. Por isso no modelo de replicação passiva apenas faltas de parada são toleradas, e apenas em condição de tempo síncrono.

Na replicação ativa, todas as réplicas são solicitadas a executar o pedido do cliente. Dessa maneira, é possível que o cliente consiga executar a operação se ao menos uma réplica funcionar corretamente. Esse fato traduz-se na definição de que na replicação ativa as faltas de parada são toleradas com a existência de $f + 1$ réplicas, sendo f o número máximo de faltas toleradas, independentemente do modelo de tempo (síncrono ou assíncrono).

Para uma tolerância a faltas mais robusta - a tolerância a faltas bizantinas - a quantidade necessária de réplicas varia em função do modelo de tempo considerado. Para sistemas síncronos, é possível tolerar faltas bizantinas dispondo de $2f + 1$ réplicas. Essa garantia existe porque caso uma réplica não responda no tempo máximo especificado (*timeout*), aquela réplica será considerada incorreta. Se todas as réplicas responderem no tempo máximo, mas alguma réplica apresentar valor diferente das demais, é possível aplicar um critério de maioria para verificar qual a réplica que está apresentando valor incorreto. Isso deve-se ao fato de que a partir de $2f + 1$ réplicas é possível obter $f + 1$ respostas iguais provenientes de réplicas diferentes. Por exemplo, considerando três réplicas no sistema ($f = 1$), é possível obter duas respostas iguais, eliminando a necessidade de a terceira resposta estar correta.

No caso de replicação ativa e sistema assíncrono (ou parcialmente síncrono), serão necessárias $3f + 1$ réplicas para que o sistema tolere faltas bizantinas. Isso deve-se ao fato de que, da mesma maneira que no modelo síncrono, deve haver uma maioria de respostas corretas ($f + 1$) para garantir a correção da resposta. Subtraindo estas respostas corretas ($f + 1$) do total de réplicas ($3f + 1$), restam $2f$ réplicas, das quais f podem ter comportamento incorreto e f podem apresentar atraso de rede (alta latência de resposta). Independente da origem do problema, o progresso do protocolo deve ocorrer. Por isso, a tolerância a faltas bizantinas inclui como possibilidade o mau funcionamento de réplicas, bem como o mau funcionamento da rede, e apesar destes problemas, fornece a resposta correta ao cliente, respeitando o limite de faltas toleradas no sistema.

Replicação de máquinas de estado: os protocolos que permitem tolerar faltas bizantinas geralmente consideram dois tipos de estratégias para realizar a replicação: os quóruns [26] e a replicação de máquinas de estado (RME) [32]. Na RME, um cliente envia a solicitação às réplicas e ocorre uma comunicação entre essas réplicas com o objetivo de estabelecer uma ordem de execução das solicitações. Considerando que vários clientes estarão efetuando requisições ao sistema, é necessário que as réplicas estabeleçam uma ordem na execução dos pedidos, de tal maneira que todas as réplicas executem os pedidos na mesma ordem, mantendo assim o estado de todas as réplicas da mesma maneira. O protocolo PBFT [12] é um dos mais conhecidos na realização de tolerância a faltas bizantinas utilizando RME, e seus passos de comunicação estão ilustrados na figura 1.3. O cliente envia uma solicitação *R* a todas as réplicas (etapa *I*); uma das réplicas é definida previamente como líder, cuja função é ordenar a solicitação *R* em relação às anteriores e reenviar esta solicitação às demais réplicas, informando uma ordem de execução *i* de-

terminada para aquela solicitação *R* (etapa *II*, denominada *pré-prepare*). No próximo passo, as réplicas que receberam a mensagem do líder trocam informações entre si, com o objetivo de certificar-se de que as outras réplicas também foram orientadas pelo líder a executar a solicitação *R* seguindo a ordem de execução *i* (etapa *III*, denominada *prepare*). Neste momento, pode-se verificar que uma réplica apresentou comportamento incorreto, ou o líder atuou de forma incorreta (por exemplo, enviando o mesmo pedido com ordens diferentes para réplicas diferentes). No caso de comportamento incorreto do líder, as réplicas que percebem esse fato solicitam uma troca de visão, operação que tem por objetivo eleger uma outra réplica para exercer o papel de líder. Quando ocorre essa mudança de visão, o protocolo precisa ser reiniciado, e nesse contexto a etapa *IV*, denominada *commit*, tem por objetivo organizar a execução de pedidos definidos antes e depois de uma possível troca de visão.

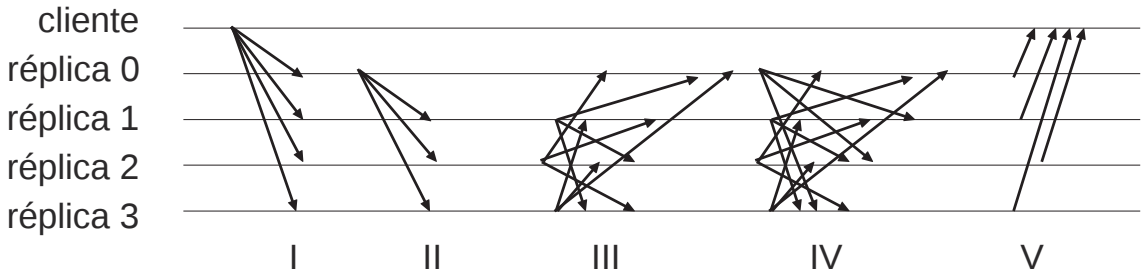


Figura 1.3: O protocolo PBFT [11].

Quóruns: os protocolos RME podem ser utilizados para realizar armazenamento de dados, entretanto essa técnica requer que as réplicas sejam entidades ativas, no sentido de comunicar-se com outras réplicas, verificar se as respostas das outras réplicas configuram maioria de respostas iguais, etc. A abordagem de quóruns [26] é capaz de trabalhar com réplicas passivas, ou seja, réplicas que não possuem capacidade de processamento, mas apenas realizem operações simples de ler e escrever dados. Inicialmente, será considerado o caso de faltas de parada, tal que $n = 2f + 1$ réplicas são suficientes para tolerar este tipo de falta; o sistema considerado será assíncrono. De maneira geral, a utilização de quóruns em sistemas de armazenamento de dados tolerantes a falta de parada ocorre da seguinte maneira: o cliente solicita a $2f + 1$ réplicas que armazenem um dado; quando $f + 1$ réplicas confirma a execução do armazenamento, o cliente obtém a confirmação da operação. Quando uma leitura de dados é solicitada por um cliente, novamente $2f + 1$ réplicas são questionadas sobre o valor do dado, e quando $f + 1$ réplicas respondem, o cliente pode obter o valor do dado. Considere que o sistema não suporta concorrência de escrita, ou seja, apenas um cliente por vez realizará a escrita de um dado. Além disso, os dados são versionados, ou seja, cada dado escrito possui um número de versão incremental, cujo maior valor define a versão mais recente do dado. Dessa maneira, a tabela 1.1 ilustra o desenvolvimento de solicitações de escrita de dados em um objeto. Considere a escrita de dados contendo os parâmetros (*valor*, *versão*), onde *valor* é o valor a ser escrito e *versão* é a versão do *valor*. Inicialmente, todas as réplicas possuem um valor nulo ($\perp$) no objeto.

operação	R_A	R_B	R_C
	$\perp$	$\perp$	$\perp$
c1, write	(X,1)	(X,1)	
c2, write		(Y,2)	(Y,2)

Tabela 1.1: Operações de escrita em um sistema de armazenamento usando quóruns.

O primeiro cliente escreveu o valor X, sob um número de versão 1, e as réplicas R_A e R_B executaram esse pedido; a réplica R_C ainda não executou o pedido por motivo de atraso de rede. Um segundo cliente escreveu o valor Y, sob o número de versão 2, sendo atendido pelas réplicas R_B e R_C. Um terceiro cliente que deseja ler o valor atual do objeto armazenado poderá obter dois conjuntos de respostas provenientes das réplicas: $(X,1),(Y,2)$, das réplicas R_A,R_B e R_A,R_C, ou o valor $(Y,2)$ das réplicas R_B,R_C. Para definir a resposta mais recente, observa-se o maior valor no número de versão. No primeiro caso, o valor Y será decidido como o valor mais recente, pois possui o maior número de versão. No segundo caso, como os números de versão são iguais, o critério de maior valor não tem efeito e o resultado retornado também será o valor Y. A propriedade de interseção dos quóruns garante o correto funcionamento apesar de possíveis falhas de parada, como as falhas consideradas no exemplo. Um dos maiores desafios na detecção de falhas de parada em sistemas assíncronos é diferenciar réplicas lentas de réplica que sofreram uma parada total. Por isso, réplicas lentas são consideradas faltosas em protocolos tolerantes a faltas que trabalham em redes assíncronas.

A interseção dos quóruns garante que sempre uma das réplicas vai possuir o valor de versão mais recente. Entretanto, se a réplica que possuir o valor mais recente for uma réplica maliciosa, a resposta retornada ao cliente poderá ser incorreta. Dessa maneira, para tolerar faltas bizantinas em sistemas que utilizam quóruns é necessário adicionar mais uma quantidade de f réplicas no sistema, garantindo então que mesmo que uma réplica apresente valores incorretos, ainda haverá outra réplica que possuirá o valor correto mais recente. Uma réplica maliciosa pode apresentar valores incorretos de duas maneiras: com versões antigas de dados, ou com versões futuras de dados. No primeiro caso, as réplicas adicionais inseridas carregam a versão mais recente do dado (a quantidade de réplicas aumenta de $2f+1$ para $3f+1$; o quórum de gravação aumenta de $f+1$ para $2f+1$), tornando portanto sem efeito o valor antigo de versão apresentado pela réplica incorreta. Para evitar que réplicas maliciosas apresentem valores de versões futuras de dados, utiliza-se a estratégia de fazer com que os escritores de dados compartilhem uma chave criptográfica, tornando assim impossível para a réplica forjar um número futuro de versão de dado. Uma outra maneira de não permitir valores maiores incorretos é utilizar a técnica de *non-skipping timestamps* [4], onde os valores obtidos são ordenados e considera-se como correto e maior valor o $(f+1)$-ésimo valor.

Paxos: por fim, é importante apresentar o Paxos [24], que pode ser considerado uma família de protocolos, dada a grande quantidade de implementações que seguem sua definição. O Paxos é um algoritmo que tem por objetivo resolver o problema de consenso, que consiste em, dado um conjunto de participantes, decidir por um valor, a partir de propostas de um ou mais participantes. O Paxos, em sua forma mais básica, baseia-se na lógica de que participantes devem propor valores até que um valor seja aceito por uma

maioria de participantes. Os participantes deste processo incorporam papéis denominados: cliente, proponente, eleitor e aprendiz. A figura 1.4 ilustra as etapas de comunicação entre os participantes. O cliente faz uma solicitação ao sistema (etapa *I*); o proponente recebe a solicitação do cliente, atribui um valor à solicitação e encaminha aos eleitores a proposta que sugere o valor atribuído (etapa *II*). Pode-se considerar um sistema onde qualquer participante pode atuar como proponente, ou pode-se realizar uma eleição para definir um líder de tal forma que apenas o líder será proponente. A eleição também é uma espécie de consenso, e o Paxos pode ser utilizado para tal fim. Os eleitores recebem propostas do proponente, verificam se irão aceitar ou não a proposta, e respondem ao proponente caso possam aceitá-la (etapa *III*). O proponente, se conseguir obter respostas dos eleitores de tal forma que seja obtida uma maioria de votos apontando o mesmo valor, informa aos eleitores que foi possível decidir por um valor (etapa *IV*). Os eleitores, ao receberem a informação de que foi possível decidir por um valor, verificam se aquela decisão ainda é válida para eles; em caso positivo, retornam uma confirmação ao proponente informando que o valor foi aceito por eles, e repassam o valor decidido aos aprendizes (etapa *V*). Os aprendizes, após receberem um valor definido (e aceito) pelos eleitores, realizam a tarefa solicitada e então retornam o resultado ao cliente (etapa *VI*).

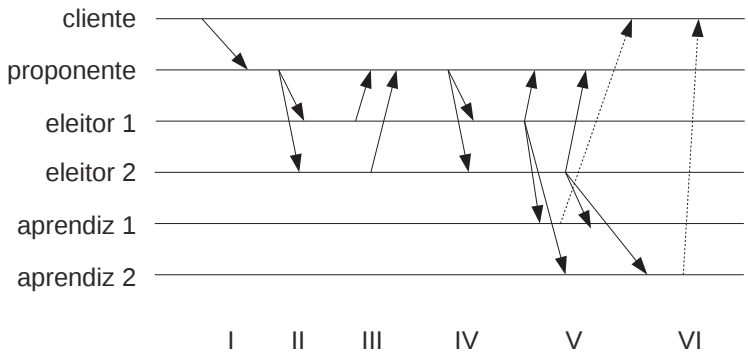


Figura 1.4: Etapas do algoritmo Paxos básico [24].

Existem alguns detalhes que fazem do Paxos um protocolo altamente flexível e versátil. O eleitor, quando recebe uma proposta de número *n* (etapa *II*), verifica se *n* é maior do que o maior número de proposta recebido até então. Caso esta verificação se confirme, o eleitor retorna ao proponente uma mensagem que representa uma ”promessa” de que ignorará futuras propostas com números menores do que *n* (etapa *III*). Informações sobre ”promessas” anteriormente feitas ou propostas anteriormente aceitas também são enviadas ao proponente, juntamente com a ”resposta-promessa”. Quando o proponente recebe uma quantidade de respostas dos eleitores de tal forma que essa quantidade configura uma maioria de respostas, o proponente pode definir o número de sua proposta; geralmente essa decisão usa o critério do maior valor. Após decidir o número da proposta, o proponente envia aos eleitores uma mensagem (etapa *IV*) informando que a proposta foi aceita por uma maioria de eleitores, e informa também qual foi o valor definido para a proposta. Quando o eleitor recebe uma mensagem de proposta aceita, ele deverá aceitar essa decisão somente se não prometeu a ninguém que iria aceitar somente propostas com valor maior que *n*. Caso esta situação se confirme, o eleitor registra o valor definido e envia aos aprendizes a solicitação e o valor aceito (etapa *V*). Essa mensagem é também enviada ao proponente.

O algoritmo Paxos é bastante utilizado em provedores de nuvens. A nuvem Microsoft Azure menciona que ”Um algoritmo de consenso, semelhante ao Paxos, é utilizado para manter as réplicas consistentes.”[12]. É possível identificar características do Paxos na descrição detalhada de como a nuvem Azure proporciona tolerância a faltas em seus bancos de dados:

> O banco de dados SQL da Microsoft Azure mantém múltiplas cópias de todos os bancos de dados em diferentes nós localizados em subsistemas físicos independentes, como diferentes servidores e roteadores. A todo tempo, os bancos de dados são mantidos em três réplicas: uma primária e duas secundárias. O mecanismo de replicação usa um esquema de confirmação baseado em quóruns onde dados são escritos na réplica primária e em uma réplica secundária antes que a transação possa ser confirmada. Se a réplica primária falhar, a falha é detectada e o sistema aciona a réplica secundária. Em caso de perda física da réplica, o sistema cria uma nova réplica automaticamente. Portanto, existem sempre ao menos duas réplicas de banco de dados que possuem consistência transacional.

1.3. Segurança: confidencialidade e fragmentação

Esta seção contém técnicas utilizadas para prover confidencialidade aos dados. É apresentada uma alternativa ao gerenciamento de chaves tradicional: o compartilhamento secreto (*Secret Sharing* - SS), onde é possível distribuir uma chave entre participantes, ao invés de manter a custódia da chave sob um elemento único ou centralizado. Será também abordada a técnica de criptografia baseada em identidade (*Identity Based Encryption* - IBE), que possibilita a construção de uma chave pública a partir de um conjunto de caracteres conhecido, como um nome ou um e-mail.

Esta seção inclui também a fragmentação de dados, uma operação que auxilia na proteção do dado com a estratégia de fragmentação, replicação e espalhamento (*Fragmentation, Replication and Scattering* - FRS), bem como auxilia na economia de espaço de armazenamento por meio da técnica de *erasure codes*, amplamente utilizada por provedores de nuvens comerciais. O particionamento de dados será também apresentado como estratégia para permitir um melhor desempenho de aplicações, bem como melhorar a capacidade de tolerância a faltas dos sistemas. Por fim, serão apresentados alguns tipos de consistência e suas características em relação aos dados considerados.

1.3.1. Compartilhamento secreto

O compartilhamento secreto [33] (*secret sharing - SS*) é uma técnica que permite codificar um segredo e distribuir as partes (também chamadas compartilhamentos) entre participantes, de maneira que seja necessária uma quantidade mínima de compartilhamentos para que o segredo possa ser restaurado. Geralmente especifica-se um esquema do tipo (m,k), onde m é o número total de partes e k é o número mínimo de partes necessárias para reconstruir o segredo.

[12]http://azure.microsoft.com/blog/2012/07/30/fault-tolerance-in-windows-azure-sql-database/

Em sistemas de armazenamento, é comum a utilização da técnica de SS para gerenciar uma chave criptográfica por meio da distribuição da chave em forma de compartilhamentos. Assim, para decifrar um conteúdo armazenado, é necessário reunir uma quantidade mínima de provedores que possuam compartilhamentos, possibilitando a restauração da chave simétrica que poderá decifrar o conteúdo ofuscado.

Como exemplo, seguem comandos para criar compartilhamentos secretos no Linux Debian 7. O programa selecionado foi a biblioteca *libgfshare-bin*. Considere o arquivo *senha.txt*, listado abaixo, contendo o texto *senha secreta*. Os compartilhamentos serão criados no esquema $(3,2)$: dentre três compartilhamentos, quaisquer dois deles são suficientes para restaurar o segredo. Seguem os comandos:

```
$ ls -l
-rw-r--r-- 1 friend friend 14 Sep  2 09:07 senha.txt
$ gfsplit -n 2 -m 3 senha.txt
$ ls -l
-rw-r--r-- 1 friend friend 14 Sep  2 09:07 senha.txt
-rw-r--r-- 1 friend friend 14 Sep  2 10:40 senha.txt.014
-rw-r--r-- 1 friend friend 14 Sep  2 10:40 senha.txt.104
-rw-r--r-- 1 friend friend 14 Sep  2 10:40 senha.txt.233
```

Os compartilhamentos possuem o mesmo tamanho do segredo[13], porém seu conteúdo não tem qualquer característica ou proximidade com o segredo. Para restaurar o conteúdo, bastam quaisquer dois compartilhamentos. Após a restauração, pode-se averiguar que não há diferenças entre o arquivo restaurado e o arquivo original.

```
$ gfcombine -o senhaRestaurada.txt senha.txt.014 senha.txt.104
$ ls -l
-rw-r--r-- 1 friend friend 14 Sep  2 10:42 senhaRestaurada.txt
-rw-r--r-- 1 friend friend 14 Sep  2 09:07 senha.txt
-rw-r--r-- 1 friend friend 14 Sep  2 10:40 senha.txt.014
-rw-r--r-- 1 friend friend 14 Sep  2 10:40 senha.txt.104
-rw-r--r-- 1 friend friend 14 Sep  2 10:40 senha.txt.233
$ diff senha.txt senhaRestaurada.txt
$
```

É importante mencionar que os compartilhamentos são restaurados a despeito de corrupções que possam ter sido realizadas nos mesmos. Assim, para garantir a integridade, é necessário proteger o compartilhamento, por exemplo, com assinatura digital. Outra maneira é dispor de uma quantidade suficiente de compartilhamentos de tal forma que seja possível restaurar corretamente o segredo mesmo que uma quantidade de compartilhamentos esteja corrompida. No sistema de armazenamento Belisarius [27], a garantia de integridade dos compartilhamentos é realizada exigindo-se que o sistema contenha $2f+2$ compartilhamentos, de forma que seja sempre possível verificar a correção de $f+2$ compartilhamentos. A figura 1.5a mostra o caso onde os três compartilhamentos gerados estão corretos, e a combinação entre quaisquer dois compartilhamentos irá gerar o segredo correto. Já no caso da figura 1.5b, existe um compartilhamento incorreto, fazendo

[13] Apesar de o texto ser formado por 13 caracteres, é possível ver na listagem o tamanho 14. O fato é que existe o caractere de término de linha, ou *carriage return*, de código igual a $0A$, no Linux; no Windows, o término de linha é representado por dois caracteres adicionais: *line feed* e *carriage return*, respectivamente, $0D$ e $0A$, ou \n e \r.

com que seja possível restaurar o segredo de três maneiras diferentes. Neste caso, não é possível identificar qual é o compartilhamento incorreto. Dispondo de $2f + 2$ compartilhamentos, conforme mostra a figura 1.5c, é possível obter $f + 2$ alinhamentos (conjuntos de $f + 1$) de compartilhamentos que levarão ao mesmo segredo; assim, é possível identificar qual é o compartilhamento incorreto.

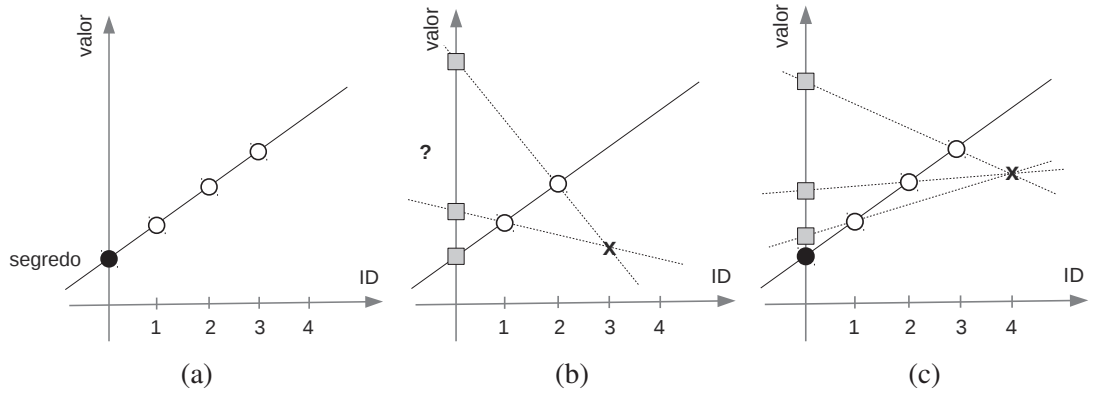


Figura 1.5: (a) compartilhamentos corretos; (b) um compartilhamento incorreto, não é possível saber qual é o segredo correto; (c) é possível identificar qual o compartilhamento incorreto [27].

1.3.2. Criptografia Baseada em Identidade

A criptografia baseada em identidade (Identity Based Cryptography - IBE) foi proposta por Shamir [34] com o objetivo de se utilizar conjuntos de caracteres, denominados identidades, como chaves públicas. Desta forma, não haveria a necessidade de se calcular aleatoriamente um número e vinculá-lo a uma pessoa por meio de um certificado digital. Isto significa que pode-se obter a chave pública de um indivíduo sem consultar nenhum repositório ou verificar certificados, basta ter o conhecimento do identificador relacionado. Um exemplo seria a utilização de um e-mail para a chave pública de um indivíduo.

Entretanto, a proposta de Shamir concebia somente a assinatura baseada em identidade e não havia como utilizá-la nas aplicações reais para o ciframento baseado em identidade. Em 2001 Boneh e Franklin [8] propuseram o primeiro esquema viável matematicamente de IBE, fazendo com que houvesse uma renovação no interesse dos pesquisadores e da indústria por novas propostas de uso do IBE. A proposta de Boneh baseia-se no uso do emparelhamento bilinear e possui as mesmas características da proposta de Shamir.

A IBE baseia-se no uso de uma Autoridade de Confiança (AC) para que esta emita as chaves privadas referentes às identidades utilizadas como chaves públicas. Dessa forma, um usuário que tem um identificador *ID* deve se comunicar com o sua AC e se autenticar. Caso essa autenticação seja feita de maneira correta, a AC envia por meio de um canal seguro a chave privada *CPriv* relacionada com o identificador *ID*. A IBE pode ser descrita com base nos seguintes passos, segundo Boneh:

- **Inicialização do Sistema:** Tem como entrada um parâmetro de segurança e retorna um conjunto de parâmetros públicos e uma chave mestra. Entre os parâmetros

públicos, deve-se conter a descrição do espaço finito que será utilizado para uma mensagem e uma descrição do espaço finito de um texto cifrado. Os parâmetros públicos serão publicamente divulgados enquanto a chave mestra será guardada e conhecida apenas pela AC.

- **Extração da Chave Privada:** Tem como entrada o conjunto de parâmetros públicos, a chave mestra e um identificador arbitrário *ID* e retorna uma chave privada. O identificador *ID* é um conjunto de caracteres arbitrário que se utiliza como chave pública.
- **Cifragem:** Tem como entrada os parâmetros públicos, o identificador *ID* e uma mensagem. O método retorna um texto cifrado.
- **Decifragem:** Recebe-se como entrada um texto cifrado e uma chave privada. O método retorna o texto em claro.

1.3.3. Fragmentação de dados

Considerando os sistemas de armazenamento de dados, a replicação integral de um dado implica em custos proporcionais à quantidade de réplicas utilizadas. Além disso, problemas comuns como uma pequena corrupção em parte do dado tornam a simples replicação custosa, visto que essa corrupção pode comprometer todo o restante da informação. Dessa maneira, foram criadas técnicas de fragmentação de dados, com o objetivo de permitir um isolamento de partes do dado e consequentemente o tratamento diferenciado de cada parte, ao invés de tratar o dado como um todo. Nesta seção duas estratégias serão apresentadas: a fragmentação com replicação e espalhamento, e a fragmentação com redundância.

1.3.3.1. Fragmentação, replicação e espalhamento

Um dos primeiros trabalhos que sugeriram a fragmentação de dados foi o artigo de Fraga [17], que estabeleceu o termo ”tolerância a intrusão”. A seguir, Deswarte [15] utilizou a técnica em um sistema juntamente com estratégias para o gerenciamento de acesso, consolidando a técnica FRS (*fragmentation, replication and scattering*). Os dados considerados pelo FRS são arquivos. De uma maneira geral, o processo ocorre conforme a figura 1.6. O arquivo original é convertido em fragmentos, cada fragmento é replicado e a seguir os fragmentos são espalhados pelas réplicas. As questões de quantidades de fragmentos e réplicas tem sido investigadas deste então, no sentido de definir quais os valores ideais para cada nível de tolerância desejado.

Em detalhe, conforme mostra a figura 1.7a, um arquivo é fragmentado em páginas (no artigo esta operação é denominada particionamento); eventualmente é necessário adicionar um complemento ao arquivo (*padding*) (etapa *I*), para que as páginas tenham tamanhos iguais (etapa *II*). A seguir, conforme ilustra a figura 1.7b, cada página é cifrada com uma chave simétrica (etapa *III*) e então a página cifrada é fragmentada (etapa *IV*). A distribuição dos fragmentos deve ocorrer em ordem aleatória para diminuir a chance de observadores na rede (ataques *eavesdropping*) identificarem a ordem dos fragmentos durante o envio.

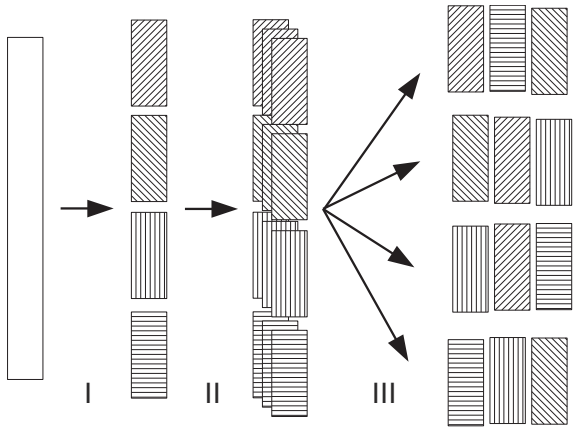


Figura 1.6: Visão geral do processo FRS [15].

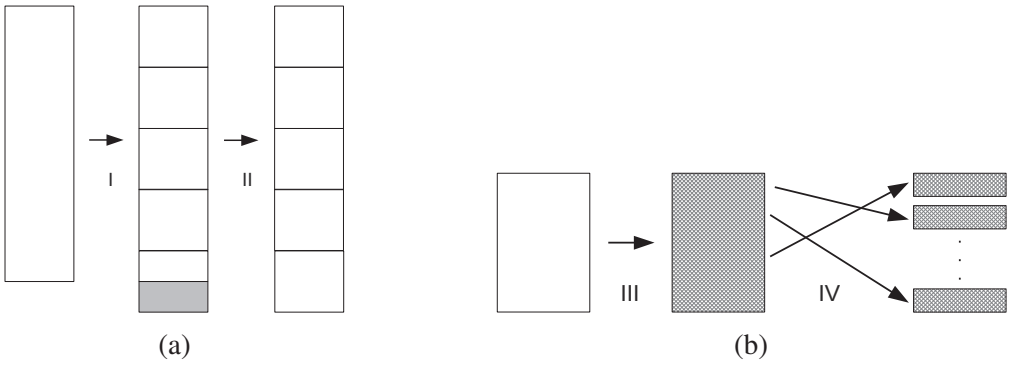


Figura 1.7: (a) Preenchimento complementar para igualar o tamanho da última página; (b) Cifragem e fragmentação da página [15].

A técnica FRS possui como premissa a replicação de fragmentos para garantir a redundância do dado. É possível fazer várias construções de acordo com o número de replicação dos fragmentos e o número de servidores de armazenamento disponíveis. Por exemplo, considere a fragmentação de um dado em três fragmentos, conforme ilustração na figura 1.8. O dado é particionado (1); a seguir, é inserido um complemento para igualar o tamanho da última página às demais páginas(2). Cada página então é cifrada (3), fragmentada (4) e replicada (5). Os fragmentos podem ser agrupados dois a dois e enviados a três provedores diferentes (6). O conjunto de fragmentos obtidos de quaisquer dois provedores é suficiente para restaurar o dado.

A técnica FRS garante confidencialidade ao dado; entretanto, possui um fator de ocupação de espaço igual ao dobro do espaço requerido pelo dado. Esta quantidade equivale a uma replicação integral. A próxima técnica (fragmentação com redundância) realiza operação de fragmentação semelhante à FRS, ocupando bem menos espaço, porém sem garantir confidencialidade ao dado.

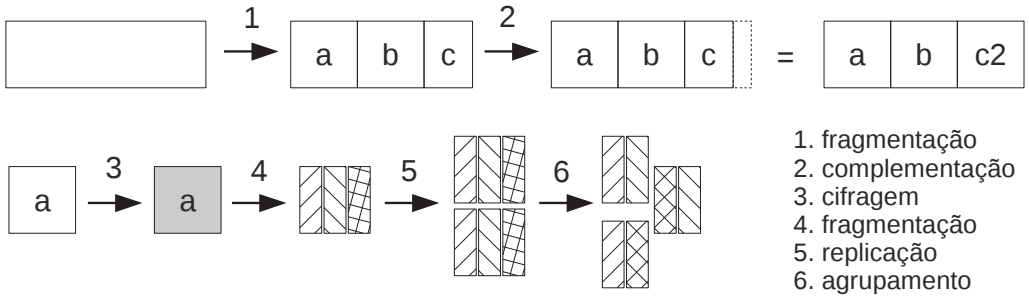


Figura 1.8: Exemplo de aplicação da técnica FRS: replicação em três provedores.

1.3.3.2. Fragmentação com redundância

Os códigos de correção de erro (CRC)[14], foram criados para tolerar pequenas corrupções em dados. De forma geral, algumas informações eram agregadas ao dado original, tornando possível a recuperação de perdas de dados. Esta técnica era inicialmente utilizada para corrigir perdas em sinais de comunicação; posteriormente, foi adaptada para ser utilizada em armazenamento de dados, permitindo a correção de dados em repouso.

A correção de erros foi utilizada para prover redundância a dados, em uma técnica denominada RAID [29] (*Redundant Array of Inexpansive Disks*), que tinha por objetivo permitir a realização de armazenamento confiável utilizando dispositivos econômicos de baixa confiabilidade. Os primeiros discos rígidos não eram dispositivos confiáveis; logo, o RAID foi criado para permitir o uso de vários discos, de modo que a substituição de discos pudesse ser feita de maneira a não interromper o funcionamento do sistema nem implicar 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. A técnica de dispersão de dados [30], também chamada de fragmentação com redundância, foi criada para prover correção de erros e redundância a dados em dispositivos distribuídos por uma rede de computadores. O procedimento consiste em dividir o dado em k fragmentos e criar m fragmentos adicionais de tal forma que qualquer conjunto de k fragmentos possibilite a reconstrução do dado. A configuração dos parâmetros geralmente é definida por (k,m), sendo que k é o número de fragmentos originais e m é o número de fragmentos adicionais ou redundantes.

A utilização de fragmentação com redundância adiciona tolerância a faltas ao armazenamento de dados. Entretanto, é importante notar que esta técnica de distribuição de dados apenas tem sentido quando são utilizados no mínimo três provedores de armazenamento. A figura 1.9 ilustra o processo de codificação dos fragmentos e de restauração do dado, para o caso de uma codificação $(2,1)$. Inicialmente, o dado é dividido em dois fragmentos (I); a seguir, um fragmento redundante é criado (II), a partir do conteúdo dos outros dois fragmentos. Os três fragmentos são então distribuídos em diferentes provedores de armazenamento (III). Para restaurar o dado, dois fragmentos quaisquer são obtidos dos provedores (IV), sobre os quais é realizado um processamento para então recuperar o dado original (V).

[14]Richard W. Hamming, 1950. Error detecting and error correcting codes.

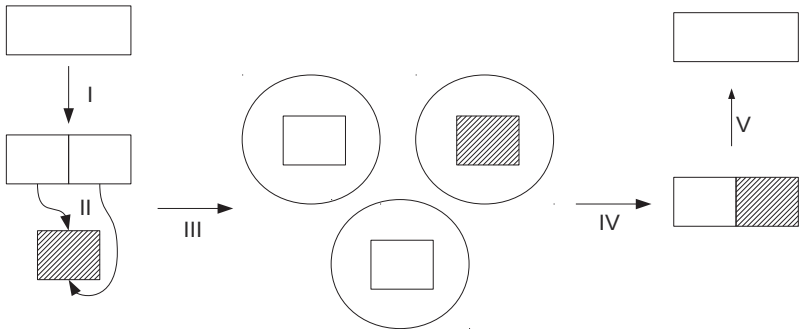


Figura 1.9: Codificação e restauração de dado em uma operação de *erasure code* (2,1).

Para fins de demonstração, seguem exemplos da fragmentação com redundância utilizando a ferramenta Jerasure[15], versão 1.2. O sistema operacional utilizado foi o Debian 7.0. Considerando o diretório onde o arquivo *.tar* foi salvo, seguem de forma resumida os comandos para a instalação deste software:

```
tar -xvf Jerasure-1.2A.tar
cd Jerasure-1.2A/Examples
make
```

A partir de então, os comandos *encoder* e *decoder* realizam as tarefas de fragmentar e restaurar o dado, respectivamente. Considere um diretório contendo os seguintes arquivos:

```
$ ls -lh
total 40M
-rwxr-xr-x 1 friend friend  69K Sep  1 10:21 decoder
-rw-r--r-- 1 friend friend  13K Sep  1 10:24 documento.pdf
-rwxr-xr-x 1 friend friend  69K Sep  1 10:21 encoder
-rw-r--r-- 1 friend friend 3.4M Jun 16 20:44 slides.pdf
-rw-r--r-- 1 friend friend  37M Sep  1 10:23 video.mp4
```

Para realizar a criação dos fragmentos, será utilizado o comando *encoder*. Os parâmetros 2 e 1 são relativos à quantidade de fragmentos originais e à quantidade de fragmentos redundantes, respectivamente. Os demais parâmetros são específicos do Jerasure; para verificar detalhes, consulte a documentação disponível no relatório técnico[16] da biblioteca Jerasure. Após a codificação, é criado o diretório *Coding*, que contém os fragmentos e metadados sobre a operação:

```
$ ./encoder slides.pdf 2 1 reed_sol_van 8 1024 1000
$ ls -lh Coding/
total 5.1M
-rw-r--r-- 1 friend friend 1.7M Sep  1 10:40 slides_k1.pdf
-rw-r--r-- 1 friend friend 1.7M Sep  1 10:40 slides_k2.pdf
-rw-r--r-- 1 friend friend 1.7M Sep  1 10:40 slides_m1.pdf
-rw-r--r-- 1 friend friend   54 Sep  1 10:40 slides_meta.txt
```

[15]Disponível em http://web.eecs.utk.edu/ plank/plank/www/software.html.

[16]http://web.eecs.utk.edu/ plank/plank/papers/CS-08-627.pdf

Note que os fragmentos possuem tamanho igual à metade do arquivo original. Isto deve-se ao fato de que o tamanho dos fragmentos é estabelecido em função do número de fragmentos originais especificado (no exemplo, $k = 2$). O tamanho dos fragmentos, portanto, é igual ao tamanho do arquivo original divido pelo parâmetro k. Os metadados contém informações necessárias durante a restauração do dado:

```
$ cat Coding/slides_meta.txt
slides.pdf
3534834
2 1 8 1024 65536
reed_sol_van
0
54
```

Um dos fragmentos será excluído para que o funcionamento da ferramenta seja aferido. Para restaurar o arquivo original, é utilizado o comando *decoder*. O arquivo original será restaurado dentro da pasta *Coding*:

```
$ rm Coding/slides_k2.pdf
$ ./decoder slides.pdf
$ ls -lh Coding/
total 6.8M
-rw-r--r-- 1 friend friend 3.4M Sep  1 10:44 slides_decoded.pdf
-rw-r--r-- 1 friend friend 1.7M Sep  1 10:40 slides_k1.pdf
-rw-r--r-- 1 friend friend 1.7M Sep  1 10:40 slides_m1.pdf
-rw-r--r-- 1 friend friend   54 Sep  1 10:40 slides_meta.txt
```

Pode-se verificar se o arquivo original está igual ao arquivo restaurado; no caso do comando abaixo, nenhuma saída foi mostrada, indicando que os arquivos são idênticos. Adicionalmente, podem ser obtidos resumos dos arquivos para outra verificação de integridade.

```
$ diff slides.pdf Coding/slides_decoded.pdf
$ md5sum slides.pdf
020dd875eacfed1f3a8e9da2fefb8efe  slides.pdf
$ md5sum Coding/slides_decoded.pdf
020dd875eacfed1f3a8e9da2fefb8efe  Coding/slides_decoded.pdf
```

1.3.4. Particionamento de dados

A replicação de dados pode ser aprimorada considerando-se características específicas das aplicações. Por exemplo, em caso de dados replicados em regiões geográficas distantes, nem sempre é necessário que os dados estejam presentes em todas as réplicas, isto é, nas réplicas que estão próximas e distantes (em termos de latência de acesso) da aplicação. Por exemplo, em um sistema de cadastro de clientes, o registro de dados dos clientes pode estar armazenado em provedores próximos às regiões nas quais os referidos clientes residem. Isto significa, por exemplo, armazenar clientes de nacionalidade Brasileira em um provedor do Brasil, e clientes de nacionalidade Japonesa em provedores do Japão. Naturalmente, os dados dos provedores podem ser replicados em outros provedores, mas a replicação pode ser feita de maneira assíncrona, como comentado na seção 1.2. Separar os dados desta maneira significa *particionar* os dados. No exemplo do cadastro de clientes, a separação de registros consiste em um particionamento horizontal de dados. No

caso deste exemplo, o critério de particionamento foi a localização geográfica do dado. O particionamento horizontal pode feito de diversas outras maneiras [31]: *round-robin*, *hashing* e faixa de valores.

Com o objetivo de distribuir as informações de forma mais homogênea, o particionamento horizontal pode seguir uma ordem denominada *round-robin*: requisições sequentes armazenam dados em partições sequentes. Considerando um sistema de requisições ordenadas (sequência linearizável) de armazenamento de dados, a primeira requisição de dados armazenará o dado na partição 0, a segunda requisição armazenará o dado na partição 1, e assim por diante. O pedido R_i armazenará o dado na partição $P_{(i \ mod \ n)}$, sendo que n é o número total de partições. O particionamento horizontal pode também ser definido de uma maneira fixa, com faixas de valores. Um exemplo simples consiste na regra de que clientes cujo primeiro nome inicia pela letra que está na faixa entre A e L serão armazenados em um determinado provedor (partição); e clientes de nome iniciando pela letra M a Z serão armazenados em outra partição. Ainda relativo ao particionamento horizontal, outra forma bastante utilizada de selecionar a partição na qual o dado será armazenado é através de *hashing*. Neste caso, é gerado um *hash* do dado, e de acordo com este *hash* é decidido em qual partição o dado será armazenado. Pode-se considerar que o particionamento por *hashing* é uma espécie de particionamento de faixa de valores, sendo que o valor considerado é o *hash* do dado, ao invés de algum outro campo específico.

Outra maneira de atenuar o esforço de replicação é a separação de campos específicos em um conjunto de dados. Por exemplo, o cadastro de clientes considerado poderia ter um campo chamado *CPF* que faria referência ao cadastro de pessoa física, informação vital para qualquer cidadão brasileiro. Os números de CPF do cadastro de clientes deveriam ser armazenados em provedores brasileiros, para que o acesso por brasileiros seja beneficiado em termos de latência. Este tipo de separação de dados é considerado um particionamento vertical. Campos específicos podem ser armazenados em partições diferentes.

1.3.5. Consistência

A consistência dos dados armazenados em um local x refere-se a visão que os leitores têm do valor de x quando ocorrem leituras e escritas de dados em x por diversas réplicas ou processos. Lamport [22] define o nível de consistência sequencial; Tanenbaum [35] apresenta essa definição de forma pragmática: considerando que operações são executadas em diferentes réplicas, "...qualquer sequência de operações de leitura e escrita é válida, mas todas as réplicas executam a mesma sequência de leituras e escritas". A figura 1.10 demonstra exemplos com e sem consistência sequencial.

Na figura 1.10a, a réplica $P1$ escreve o valor a em x; em seguida, $P2$ escreve o valor b em x. As réplicas $P3$ e $P4$ lêem o valor b, e em um segundo momento, o valor a é lido por essas réplicas. Pode-se inferir que a escrita de b foi efetivada antes da escrita de a. Este fato não configura problema, visto que a sequência de operações observada pelas réplica $P3$ e $P4$ foi a mesma. Essa igualdade de sequência de operações entre réplicas configura a consistência sequencial. A sequência observada pelas réplicas $P3$ e $P4$ será: $W(x)b > R(x)b > Wx(a) > R(x)a$. Na figura 1.10b, as réplicas $P3$ e $P4$ observam os valores a e b em ordens diferentes, não possuindo portanto uma consistência sequencial.

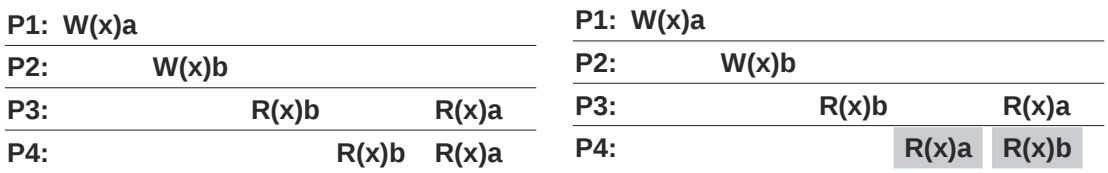


Figura 1.10: (a) Consistência sequencial; (b) Consistência não-sequencial.

A consistência causal [22] trata de operações que possuem uma relação de causalidade potencial. Tanenbaum [35] enuncia este nível de consistência de forma objetiva: "Se uma operação b é causada ou influenciada por uma operação anterior a, então a causalidade requer que todos os executores de operações executem a antes de b". Uma observação fundamental neste tipo de consistência é que operações de escrita concorrentes não podem estabelecer relação causal com leituras consecutivas. A figura 1.11 apresenta situações onde a causalidade é obedecida e violada, respectivamente. Na figura 1.11a, as réplicas $P1$ e $P2$ realizam escrita sobre objeto x. As escritas são consideradas concorrentes, pois não há relação causal entre as mesmas: nenhuma das escritas utiliza o valor anterior do objeto para a composição do novo valor que está sendo escrito. Sendo assim, como a ordem das escritas não importa, na consistência causal, a ordem das leituras também não importa. As duas sequências observadas pelas réplicas $P3$ e $P4$, respectivamente: $W(x)b > R(x)b > W(x)a > R(x)a$; e $W(x)a > R(x)a > W(x)b > R(x)b$ obedecem à relação causal entre as operações de leitura e escrita realizadas.

A figura 1.11b mostra a inserção de uma operação de leitura antes de uma escrita, na mesma réplica, o que torna a operação de escrita $W(x)b$ dependente da operação de leitura $R(x)a$; essa, por sua vez, é dependente da operação $W(x)a$. Sendo assim, a única sequência admissível que mantém a causalidade de operações é: $W(x)a > R(x)a > W(x)b > R(x)b$. Esta sequência está sendo observada pela réplica $P4$, mas a réplica $P3$ observa uma sequência que viola a causalidade entre as operações de escrita dos valores b e a. A causalidade entre estas operações de escrita existe porque a escrita de b depende potencialmente do valor de a, devido ao fato de que b pode resultar de um processamento envolvendo o valor de a.

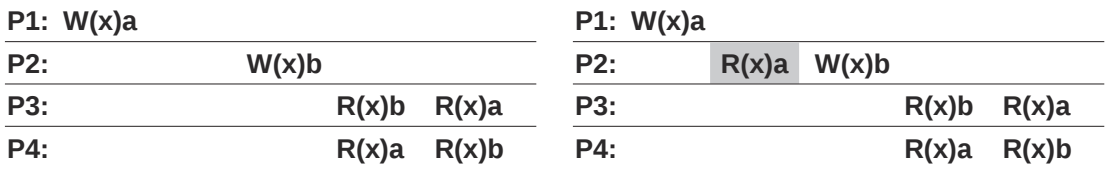


Figura 1.11: (a) Consistência causal; (b) Consistência não-causal.

Considerando a situação de concorrência entre escritas, Lamport [23] definiu três comportamentos possíveis quando leituras de dados ocorrem com escritas concorrentes:

- Na semântica segura, é possível afirmar que uma leitura não-concorrente com escritas retornará o último valor gravado; no caso de leitura simultânea com escritas,

apenas é possível afirmar que será retornado um valor válido para o local x. Por exemplo, se é possível armazenar em x valores de 0 a 9, então é possível afirmar que o valor retornado durante uma leitura concorrente com uma ou mais escritas, na semântica segura, será igual a um valor que estará localizado entre 0 e 9.

- A semântica regular inclui as características da semântica segura, e adicionalmente define que leituras realizadas de forma concorrente com uma ou mais escritas apenas possuem duas possibilidades de valor de retorno: ou irão retornar o valor anterior ao momento em que começaram as escritas, ou irão retornar o valor de uma das escritas que está sendo realizada. É possível que diferentes leitores obtenham diferentes valores.
- Na semântica atômica estão presentes as características da semântica regular, com uma restrição adicional: as escritas concorrentes, sob o ponto de vista dos leitores, tornam-se sequenciais. Como exemplo, considere que o sistema possui um valor a armazenado no local x. A partir do momento em que o sistema entregar para um leitor L o valor b, dado que b é um valor mais recente do que a, nenhum outro leitor poderá receber mais o valor a; todos os leitores subsequentes a L receberão o valor b ou outro valor mais atual do que b.

A necessidade de desempenho e a alta distribuição geográfica de sistemas proporcionada pelas redes de longa distância, como a Internet, motivaram a criação da semântica eventual. Vogels [36] apresenta alguns níveis níveis de consistência observados por aplicações, caracterizando por fim a consistência eventual:

- consistência forte: possui definição equivalente à semântica forte, de Lamport [23].
- consistência fraca: o sistema não garante que, após um escritor A concluir a operação, leitores subsequentes observarão o valor escrito por A. Diversas condições precisam ser observadas para que os leitores tenham garantia de que observarão um valor atual. Há um conceito de janela de inconsistência que se refere a um intervalo de tempo entre o momento da última atualização e o momento no qual é possível garantir que o valor retornado para um leitor é um valor atual.
- consistência eventual: é uma consistência fraca cuja janela de inconsistência pode ser definida por características do sistema, como latência do meio de comunicação, carga do sistema, número de réplicas, etc.

O nível de consistência utilizado pelas aplicações depende dos requisitos da confiabilidade que a informação precisa ter, em relação ao tempo de propagação da informação, ou a atualidade da informação. O provedor de armazenamento em nuvem Amazon S3 utiliza consistência de dados eventual para seus dados[17], sendo este nível de consistência suficiente para o Dropbox armazenar seus dados [16] nessa nuvem de armazenamento.

1.3.5.1. Teorema CAP

Dadas as diversas possibilidades de configurações que sistemas podem apresentar em relação a consistência, disponibilidade (em termos de latência) e particionamento (no sentido de desconexão de redes), foi criado o teorema CAP [9] (*Consistency, Availability and Partition*), que estabelece relações entre estes itens. Afirma-se que em sistemas

[17]http://aws.amazon.com/s3/faqs/

que compartilham dados em rede, não é possível atingir simultaneamente as máximas unidades destas grandezas. Em geral, aplicações priorizam duas características, em detrimento da terceira. Várias interpretações podem retratar a relação entre as três variáveis. Particionamento pode ser considerado como um limite de tempo de comunicação: se uma consistência não for alcançada em um determinado tempo, pode-se dizer que houve uma partição no sistema. Partições podem ser detectadas com *timeouts*; na existência de partições, o sistema pode limitar operações - por exemplo, não permitir que escritas sejam realizadas no sistema. Neste caso, a consistência está sendo priorizada. Outra possibilidade é armazenar as operações que deverão ser efetivadas quando uma posterior etapa de recuperação for executada, na ocasião do desaparecimento da partição. Assim, a disponibilidade é priorizada, permitindo que a escrita sempre possa ocorrer, permitindo no entanto que leitores nem sempre obtenham a versão mais recente do dado.

1.4. Estado da arte

Nesta seção serão comentados alguns dos trabalhos mais relevantes que abordam armazenamento de dados em provedores de nuvens.

1.4.1. SPANStore

SPANStore [37] é um sistema que permite armazenar um dado em múltiplos provedores de serviços em nuvem tendo como objetivo reduzir os custos associados à gravação e leitura do dado. A figura 1.12 apresenta a arquitetura do SPANStore. Nos *data centers* estão localizados os serviços de armazenamento e as aplicações. Cada *data center* contém uma máquina virtual (VM) contendo o SPANStore, uma VM contendo a aplicação e um serviço de armazenamento.

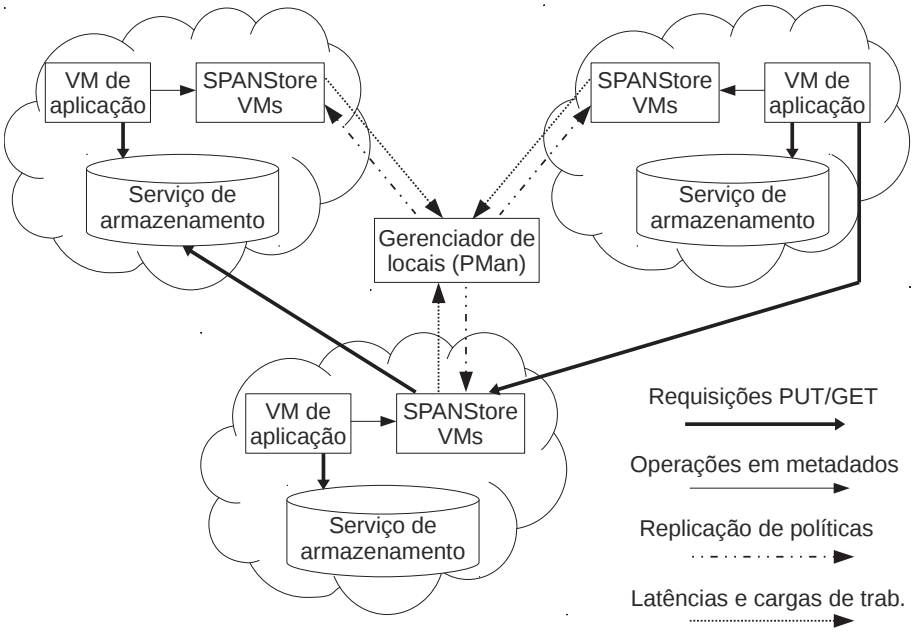


Figura 1.12: Visão geral do SPANStore [37]

A otimização de custos é alcançada com a geração de uma recomendação para a aplicação sobre quais são os provedores que devem ser utilizados para realizar o ar-

mazenamento. Para produzir a recomendação são necessários dois conjuntos de informações: informações mantidas pelo SPANStore e informações de necessidades específicas da aplicação. O SPANStore precisa ser informado com os valores praticados pelos provedores de serviço; na prática, são tabelas de preços. Esta informação pode ser obtida diretamente dos sites de provedores. Para se ter uma noção dos valores tarifados, a tabela 1.2[18] mostra informações sobre as cobranças realizadas para transferir dados a partir de instâncias de máquinas virtuais da Amazon EC2. As informações de custo utilizadas são: preço por byte armazenado, preço por PUT e GET e hora de utilização de máquina virtual (VM). Além disso, para cada par de *data centers*, obtém-se o preço ($/GB) por transferência de dados (também denominado custo da rede) que é determinado pelo preço de envio de dados a partir do *data center* de origem.

Destino		**Valor ($/GB**
EC2 mesma zona	endereço IP privado	0
	IP público	0.01
EC2 em outra região AWS		0.02
Para a Internet	Primeiro GB/mês	0
	Até 10TB/mês	0.12
	Próximos 40TB/mês	0.09
	Próximos 100TB/mês	0.07
	Próximos 350TB/mês	0.05
	Próximos TB/mês	consultar

Tabela 1.2: Valores de transferência de dados a partir de instância Amazon EC2

Ainda relativo ao conjunto das informações mantidas pelo SPANStore, existe 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*. Estas informações são coletadas por um componente do sistema nomeado *Placement Manager* (PMan), denominado Gerenciador de Locais na figura 1.12. Este componente periodicamente solicita[19] a cada *data center* as latências de comunicação entre o *data center* e os outros *data centers* com os quais ele possui comunicação, e também a frequência de acesso dos objetos na VM por diferentes clientes. Essas informações são atualizadas a cada hora. Especificamente, as informações de latência são as seguintes: medidas de *PUT*, *GET* e *pings* entre VMs e serviços de armazenamento, ou entre VMs e outras VMs em outros *data centers*. Foram utilizados *data centers* nos provedores Amazon, Microsoft Azure e Google Cloud.

O outro conjunto de informações que o SPANStore utiliza para gerar a recomendação para a aplicação é específico de cada cliente, sendo por este fornecido, e consiste nas seguintes definições: nível de consistência desejado (eventual ou forte), número máximo de falhas a tolerar, latência máxima desejada e uma fração de *requests* que devem possuir latência menor que a latência máxima especificada. A confecção da melhor recomendação pode ser considerada como um problema de otimização, e o SPANStore

[18] Valores anotados em 22/08/2014, para um *data center* da Irlanda.

[19] A solicitação é feita à VM do SPANStore existente no *data center.*

utiliza um algoritmo que utiliza técnicas de programação inteira (também denominada programação linear) para resolver esta questão.

Quando a consistência definida pela aplicação é eventual, é possível replicar os dados de maneiras diversas, conforme mostra a figura 1.13. Os itens $A..F$ são *data centers*, o cliente está localizado no *data center A* e a operação considerada é a escrita de dados. Na figura 1.13a, a confirmação da escrita é feita apenas no *data center A*, e a atualização do dado é enviada do *data center A* para todos os outros *data centers* em segundo plano (protocolos de *gossip*). Este cenário garante menor latência, pois o *data center* mais próximo é atualizado e a resposta é retornada ao cliente, enquanto as atualizações são enviadas aos outros *data centers*. Em termos de tolerância a faltas, confirma-se que uma réplica armazenou o dado, e as outras f réplicas são atualizadas assincronamente. Não é garantido que as outras f réplicas terão sucesso na operação, entretanto isto é uma premissa intrínseca da semântica eventual e da consideração de que $f+1$ réplicas armazenarão o dado.

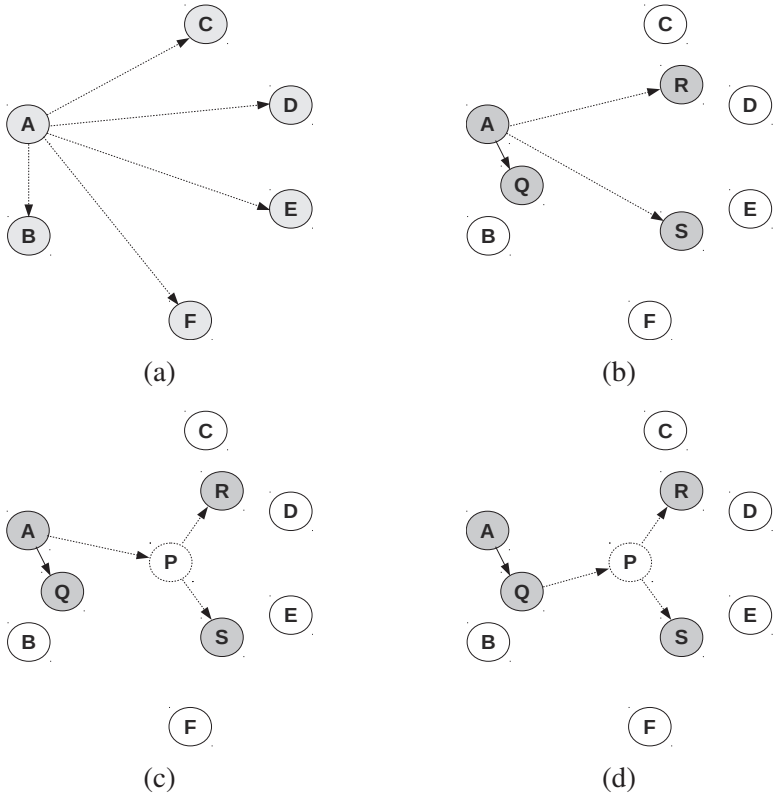


Figura 1.13: Replicação e consistência no SPANStore: (a) replicação total minimiza latência; (b) replicação parcial reduz custos; (c) delegação de escrita reduz custos de rede; (d) delegações múltiplas podem reduzir ainda mais os custos.

A figura 1.13b ilustra uma replicação parcial: nem todos os *data centers* são solicitados a armazenar o dado. Os *data centers* menos custosos são escolhidos para realizar o armazenamento de dados. O *data center* mais próximo ao cliente (neste caso, o A) pode sequer ser escolhido, devido às métricas de custo (será mais barato armazenar em Q do que em A, e a latência de Q está no limite definido pela aplicação). Na figura 1.13c ocorre

um encaminhamento (*relay*) de pedido de escrita: ao invés do cliente (localizado em *A*) solicitar o armazenamento diretamente aos *data centers* *R* e *S*, o pedido é encaminhado ao nó *P* para que o mesmo faça o armazenamento nos vizinhos *R* e *S*. Uma outra situação ilustrada na figura 1.13d pode ocorrer se, de acordo com os custos calculados, o nó *Q* além de armazenar o dado também possuir um custo de rede melhor do que o custo de rede entre os pontos *A* e *P*. Nesse caso, o nó *Q*, após realizar o armazenamento, encaminha o pedido ao nó *P*; o nó *P* não armazena o dado, mas apenas encaminha o pedido aos nós *R* e *S*.

As figuras 1.14, 1.15 e 1.16 ilustram[20] alguns cenários com uma série de detalhes que permitirão compreender melhor como o SPANStore toma algumas decisões. Os custos apresentados são denominados custo da rede. A figura 1.14 demonstra o armazenamento de dados utilizado quando a consistência exigida pela aplicação é forte. Neste caso, não pode ser utilizada delegação de pedidos, pois é necessário ter a confirmação de cada *data center* sobre o sucesso da operação de armazenamento. O custo de rede total neste caso será igual a $0.75/GB, correspondente ao envio do pedido para os três provedores.

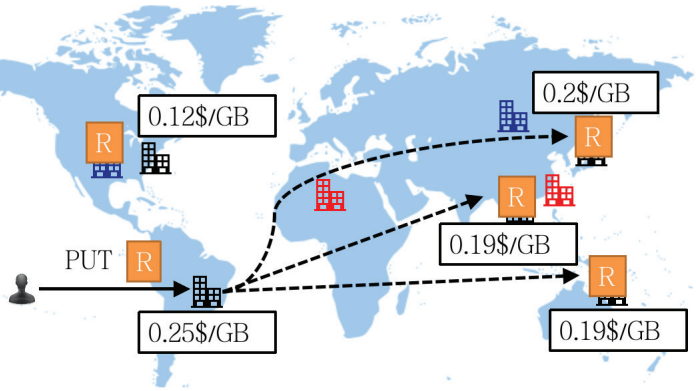


Figura 1.14: Escrita de dados, consistência forte [37].

Quando uma aplicação requer consistência eventual pode ser mais econômico delegar a transferência de dados a outro *data center* que possua menores taxas de envio de dados. Por exemplo, na figura 1.15 o pedido foi delegado ao provedor que possui custo de rede igual a $0.12/GB. Nesse caso, o valor do envio de dados para os provedores será igual a $0.25 + 3 * 0.12$, totalizando $0.61/GB, sendo este um valor menor do que $0.75/GB. Um inconveniente desta estratégia é que a transferência de responsabilidade de execução do pedido pode provocar atrasos que excederão as latências definidas no SLO (*Service Level Objective*), definido pela aplicação.

Para delegar a execução de um pedido e manter a latência dentro do SLO, pode-se solicitar a um provedor destinatário distante que realize duas tarefas: execute o pedido localmente e também solicite a outros *datacentes* que executem o pedido. Neste caso, um *data center* atua simultaneamente como provedor, quando executa o pedido, e cliente, quando solicita a outros *data centers* que executem o pedido. A figura 1.16 apresenta esse cenário; o custo de transferência será igual a $0.25 + 2 * 0.19$, totalizando $0.63/GB.

[20]Estas três figuras foram obtidas de slides dos autores do SPANStore, e estão disponíveis em http://sigops.org/sosp/sosp13/program.html

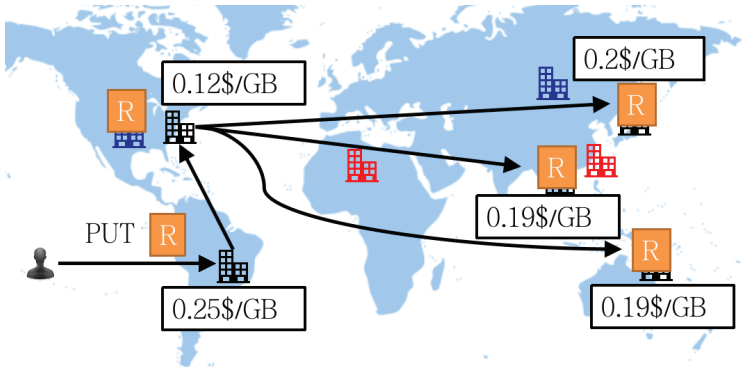


Figura 1.15: Escrita de dados, consistência fraca, *relay* da escrita [37].

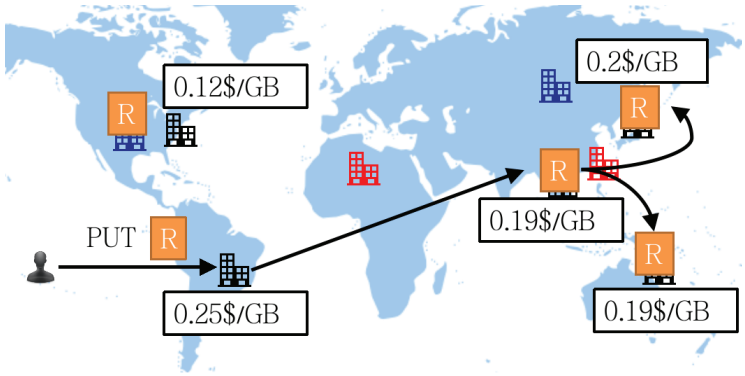


Figura 1.16: Solicitação de execução e delegação de pedido a um provedor [37].

O SPANStore apresenta considerações de custos em armazenamento nos provedores de nuvens. Ainda que exista resistência para realizar armazenamento de dados sensíveis (metadados, senhas, documentos confidenciais) em provedores de nuvens devido à falta de confiança dos clientes na garantia de confidencialidade dos seus dados, uma questão bastante relevante na utilização de serviços de armazenamento remoto é o custo. Ocorrem alterações nos valores praticados pelos provedores[21], o que pode reconfigurar o conjunto de provedores menos custosos, sob o ponto de vista do cliente. Existem ferramentas que permitem estimar gastos em provedores de nuvens[22], 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 é elaborada para a aplicação.

Sobre a tolerância a faltas, aponta-se o elemento PMan como um ponto fraco do sistema: é um elemento único (não-replicado) que, em caso de corrupção de dados (invasão maliciosa) pode apresentar respostas incorretas, afetando assim as recomendações geradas para as aplicações. Apesar de o PMan ser um elemento centralizado, ele não afeta significativamente o desempenho do sistema, pois a periodicidade com a qual o PMan é executado é grande: ele é executado a cada hora.

[21] http://techcrunch.com/2014/08/20/cloud-storage-is-eating-alive-traditional-storage/
[22] http://www.planforcloud.com

1.4.2. Augustus

O Augustus [28] é um sistema de armazenamento chave-valor tolerante a faltas bizantinas que proporciona escalabilidade através de particionamento e permite realizar operações complexas de uma maneira mais simples com a utilização de transações de curta duração (*minitransações*). As transações de curta duração surgiram em um sistema chamado Sinfonia [1], com o objetivo de abstrair o gerenciamento de concorrências e falhas no acesso a dados. No Augustus, elas recebem um identificador único, que é construído a partir de uma análise (*digest*) do conteúdo das operações na transação. Dessa forma, para garantir a unicidade, toda transação deve conter uma operação *nop* tendo como parâmetro um número aleatório. A vantagem das transações de curta duração utilizadas no Augustus é que elas diminuem a chance de haver conflitos de *locks* em dados, devido ao tamanho reduzido da transação. A figura 1.17 mostra uma visão geral do Augustus, onde diferentes clientes executam transações em uma ou mais partições.

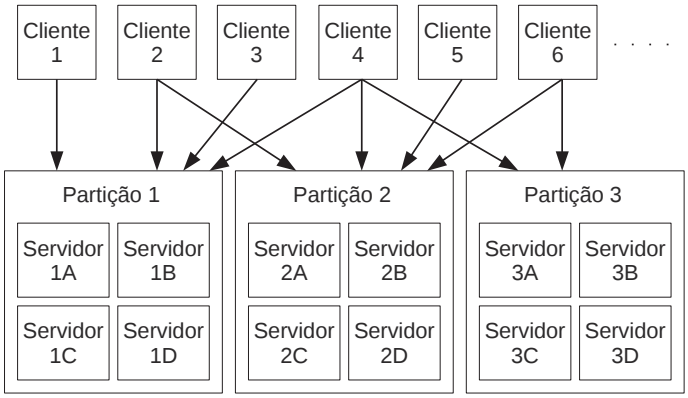


Figura 1.17: Clientes executam transações em partições [28].

As fases do protocolo estão ilustradas na figura 1.18. Inicialmente, o cliente envia a transação às partições envolvidas (passo *I*), via *multicast*. Na fase de execução (passo *II*) cada partição executa um protocolo de *atomic multicast* tolerante a faltas para ordenar as requisições; no caso do Augustus, é utilizado o PBFT [12].

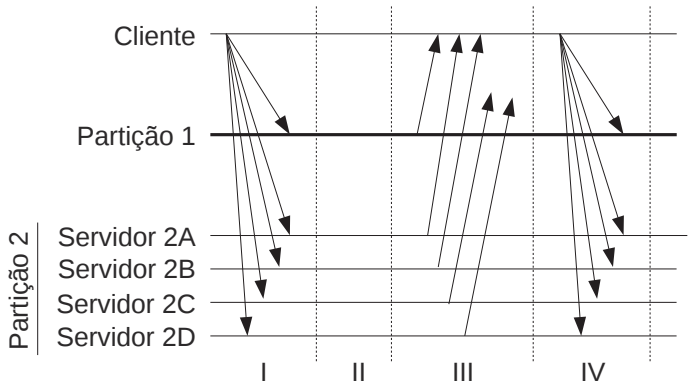


Figura 1.18: Etapas da execução global de uma transação [28].

Para executar a requisição, cada réplica na partição segue o seguinte fluxo de execução: *i*) tenta obter os *locks* dos recursos contidos na transação; *ii*) caso obtenha

sucesso, a transação é executada e assume um estado *pendente*; caso não obtenha sucesso, a transação passa ao estado *terminada* (figura 1.19); *iii*) se a transação for executada mas não obtiver sucesso (por exemplo, uma comparação que resultou em desigualdade), a mesma é abortada e passa para o estado *terminada*.

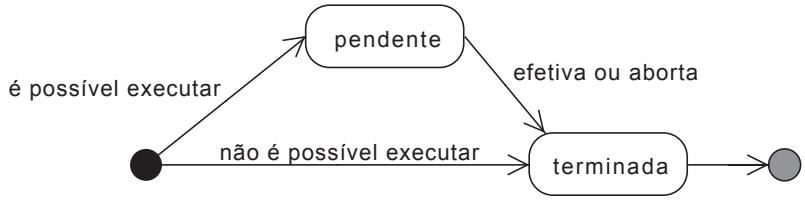


Figura 1.19: Estados possíveis a uma transação.

No passo *III*, são geradas respostas (votos) assinadas (chave privada) para o cliente: a réplica envia um *commit* ao cliente, caso tenha executado a transação com sucesso; caso contrário, envia um *abort*. Quando o cliente recebe $f_g + 1$ respostas iguais[23] das réplicas na partição g, e estas respostas confirmam a execução da transação, o cliente confecciona um certificado contendo um parecer *commit* referente ao voto da partição; caso contrário, é criado um certificado com o parecer *abort*. Quanto o cliente obtém certificados do tipo *commit* para todas as partições envolvidas na operação, considera-se que a transação foi efetivada globalmente (em todas as partições envolvidas). Na última etapa (passo *IV*), o cliente envia todo o conjunto de certificados que recebeu às partições envolvidas. Quando uma réplica recebe um certificado referente a uma transação que se encontra no estado "pendente", a transação é efetivada ou abortada, respectivamente, de acordo com o parecer do certificado recebido (*commit* ou *abort*); os *locks* são liberados e o estado da transação é atualizado para "terminada". Em réplicas que possuem transações já terminadas, o recebimento de certificados não tem efeito sobre a réplica.

Para ilustrar uma execução bem sucedida no Augustus, considere a figura 1.20 e a seguinte sequência de passos: 1) o cliente envia a solicitação a duas partições envolvidas; 2) cada partição executa o protocolo de ordenação de mensagens entre suas réplicas; 3) cada partição executa a transação solicitada; 4) as partições retornam ao cliente o estado "pendente" relativo ao pedido solicitado; 5) o cliente confecciona certificados das execuções bem sucedidas nas partições; 6) o cliente envia todos os certificados a todas as partições envolvidas; 7) as partições efetivam as transações e o protocolo termina.

Uma operação pode requerer mais de uma transação para realizar uma tarefa. A tabela 1.3 apresenta um algoritmo de *Compare-and-Swap* usando transações de curta duração no Augustus. São exibidos o código de duas transações e o código executado pelo cliente. Apesar de nas transações exemplificadas não constar a operação *nop* com um número aleatório, ela é necessária para aumentar a garantia de unicidade da transação. Inicialmente, o cliente tenta realizar a leitura do dado por meio da transação de leitura (linha 18). A falha na leitura significa que não foi possível obter o bloqueio necessário. Nesse caso, é possível repetir essa transação até que esse bloqueio seja alcançado. Após ler o valor, a aplicação modifica o dado (linha 20) e então submete a

[23]Note que cada partição possui $3f + 1$ réplicas, que são capazes de fornecer um quorum de $2f + 1$ respostas; este quorum deve possuir no mínimo $f + 1$ respostas iguais para caracterizar uma resposta correta.

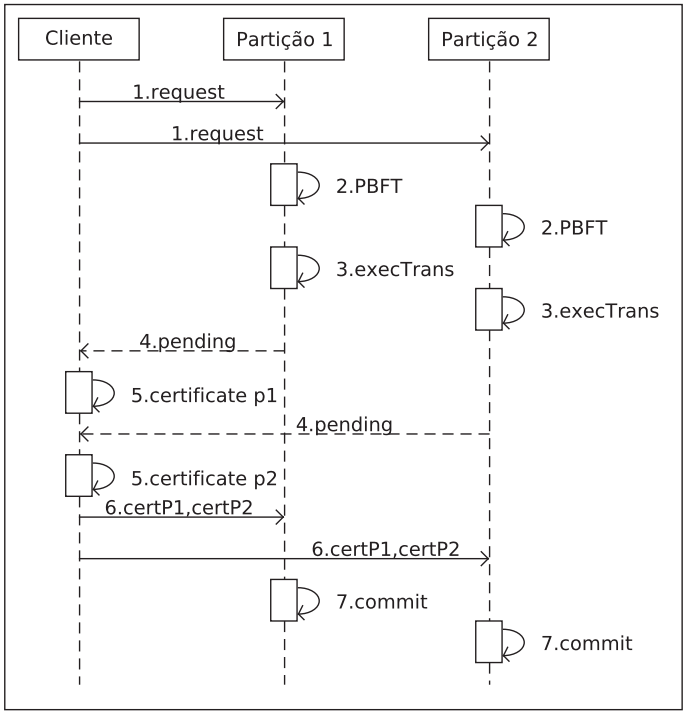


Figura 1.20: Execução bem sucedida de uma solicitação em duas partições.

transação de gravação ao Augustus (linha 21). O protocolo segue conforme o resultado da execução da transação. A gravação pode falhar quando não é possível obter o bloqueio para gravação do dado, ou quando o valor lido já foi modificado por outra aplicação. Nesse caso, será necessário executar novamente todo o código da aplicação.

Se uma transação for executada em ordens diferentes nas partições, ao menos uma das partições envolvidas enviará um *abort*, o que fará com que todas as outras partições também abortem a execução da transação. Se um cliente falhar durante a execução e deixar transações pendentes (não executarem o passo *IV* na figura 1.18), clientes corretos subsequentes atuam no sentido de solicitar o cancelamento das transações conflitantes, mediante a apresentação de certificados que mostram a situação de conflito nas outras partições. O cliente é responsável por definir o esquema de particionamento. Quando o esquema é *hashing*, quaisquer partições podem ser envolvidas. Para um esquema baseado em valor, uma análise nas operações da transação define as partições envolvidas.

O banco de dados Cassandra[24] adicionou o recurso de minitransações em sua versão 2.0 do software, permitindo, por exemplo, a não execução de um comando no caso de uma verificação falhar. O código[25] mostrado na tabela 1.4 representa uma inserção que apenas será executada se a verificação de pré-existência do dado for bem sucedida. Sem a verificação, o comando de inserção funcionaria como uma atualização, sobrescrevendo o valor antigo. Outro exemplo está na tabela 1.5: reiniciar uma senha. A senha apenas será alterada caso o campo *reset_token* possua o valor especificado.

[24]http://cassandra.apache.org/

[25]O código foi obtido no site http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0. Datastax é uma empresa que distribui comercialmente o Cassandra.

Tabela 1.3: Minitransação de alteração atômica de valor (compare-and-swap)

```
1:  function MINITRANSACTIONALREAD(key)
2:      if tryReadLock(key) then
3:          return read(key);
4:      end if
5:  end function

6:  function MINITRANSACTIONALWRITE(key, old, new)
7:      if tryWriteLock(key) then
8:          actual ← read(key)
9:          if actual == old then
10:             write(key, new);
11:             return true;
12:         end if
13:     end if
14:     return false;
15: end function
                                          ▷ Código da aplicação cliente
16: old ← null;
17: while old == null do
18:     old = miniTransactionalRead(x);
19: end while
20: new ← old + " modified";
21: success ← miniTransactionalWrite(x, old, new);
22: if success then
23:     state ← "pending"; vote ← "commit";
24: else
25:     state ← "terminated"; vote ← "abort";
26: end if
```

1.4.3. Depot

Depot[25] é um sistema de armazenamento chave-valor que alcança confiabilidade máxima usando a premissa de não confiar em nenhum nó. O Depot incorpora uma característica de sistemas *peer-to-peer*: a capacidade de comunicar-se com outros clientes; dessa maneira, uma aplicação pode permanecer operante mesmo se todas as réplicas do sistema estiverem comprometidas.

A arquitetura do Depot é mostrada na figura 1.21: um conjunto de servidores compõem um volume (uma partição). No exemplo da figura, cada partição contém uma faixa de chaves de um cliente (a estratégia de particionamento é definida pelo cliente). Qualquer servidor no volume pode ser contactado pelo cliente: os demais servidores recebem as atualizações de acordo com a forma de replicação considerada, que é transparente ao Depot. A figura 1.21 apresenta três exemplos de estratégias de replicação no interior das partições: *chain*, *mesh* e *star*. Caso todos os servidores fiquem indisponíveis, os clientes podem comunicar-se e compartilhar dados entre si. Clientes e servidores utilizam

Tabela 1.4: Exemplo de minitransação no Cassandra: inserção condicional.

```
INSERT INTO USERS (login, email, name, login_count) VALUES
('jbellis', 'jbellis@datastax.com', 'Jonathan Ellis', 1) IF NOT EXISTS
```

Tabela 1.5: Exemplo de minitransação no Cassandra: atualização condicional.

```
UPDATE users SET reset_token = null, password = 'newpassword'
WHERE login = 'jbellis' IF reset_token = 'some-generated-reset-token'
```

o mesmo protocolo, sendo assim denominados *nós* do sistema. Os dados replicados nos servidores também ficam armazenados nos clientes; existe um coletor de lixo que remove versões anteriores dos dados, segundo uma política de versionamento.

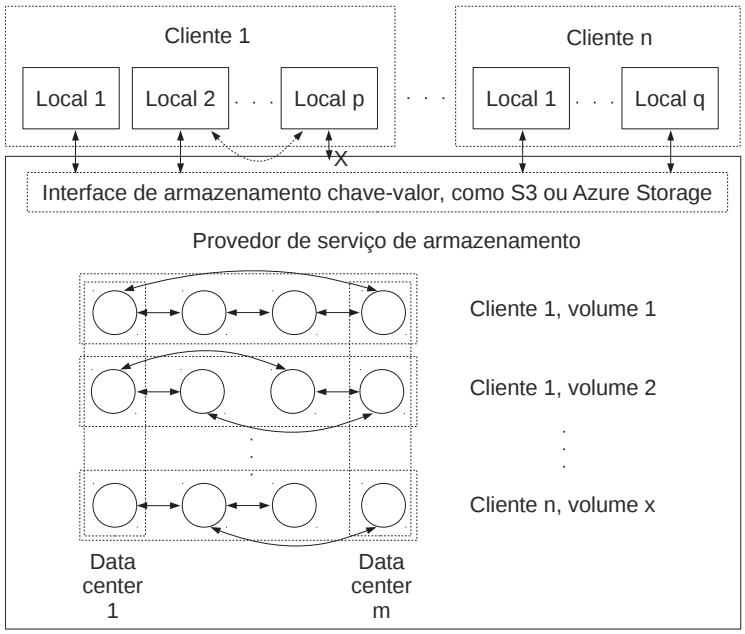


Figura 1.21: Arquitetura do Depot [25]

Para armazenar um dado no Depot, a primeira ação do cliente é armazenar o dado localmente. Em seguida, o cliente propaga a atualização para as réplicas segundo um protocolo padrão de troca de *logs* (*gossip*), em segundo plano de execução[26]. A leitura de um dado inicia-se com o cliente enviando para um servidor primário a chave k e a última versão de k que o cliente conhece (o cliente também participa da troca de *logs*, sendo eventualmente informado de atualizações de dados que conhece). Se o valor da última versão conhecida pelo servidor for o mesmo valor informado pelo cliente, o dado é retornado para o cliente. Toda chave possui associada um histórico de *hashes* dos valores anteriores. O cliente verifica se os *hashes* dos valores anteriores a k correspondem aos que ele conhece; em caso positivo, o dado é entregue à aplicação. Caso contrário, o cliente envia ao servidor primário as informações que possui sobre o dado buscado. O servidor

[26]Uma otimização de implementação utiliza de forma otimista um servidor primário (geralmente o mais próximo), ao invés de iniciar uma troca de *logs* com um conjunto de réplicas.

deverá responder com as atualizações que o cliente não possui e também o conjunto de dados mais recente que o servidor possui. Tanto na escrita quanto na leitura de dados, caso o servidor primário não responda ao cliente, outro servidor será buscado; caso nenhum servidor responda, o cliente interage com outros clientes para realizar o armazenamento ou a busca de dados. Operações cliente-a-cliente são mais rápidas do que comunicações entre clientes e servidores, pois dispensam a troca de *logs*.

Duas escritas *A* e *B* são concorrentes logicamente se a escrita *A* não aparece no histórico da escrita *B*, e vice-versa. Se essas escritas atualizarem o mesmo objeto, elas podem ser conflitantes. Escritas são ordenadas por um relógio lógico (*timestamp*); em caso de escritas com o mesmo *timestamp*, o identificador do nó serve como critério de desempate. Diversas condições certificam a efetivação de uma escrita: assinatura digital, *timestamp* e *hash* do histórico; leituras sujas e escritas concorrentes são detectadas com estas informações. Quando servidores detectam que estão fora de sincronia, é iniciada uma troca de *logs*.

A corrupção de dados pode ser resolvida com a recuperação de versões anteriores do dado. A coleta de lixo é feita pelos nós quando estes recebem uma lista de candidatos a serem descartados (*Candidate Discard List* - CDL) assinada por todos os clientes. Uma ação de um nó faltoso para comprometer a consistência causal é realizar uma escrita divergente (com valores diferentes para nós diferentes), provocando assim um *fork*. A figura 1.22 ilustra esse cenário.

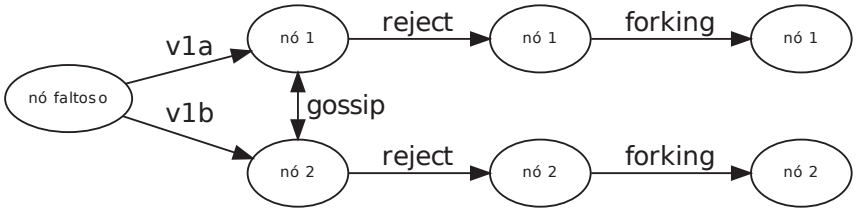


Figura 1.22: Operação *forking*: conversão de faltas em concorrência.

Para resolver essa inconsistência, cada nó considerará ambas as escritas como válidas, tornando-as concorrentes localmente; essa estratégia é denominada *forking*. No exemplo da figura 1.22, os nós trocarão informação sobre o dado a ser gravado (*gossip*); após a verificação de que os *hashes* dos históricos de valores (que incluem o valor atual) são diferentes, cada nó rejeitará a efetivação do novo valor. Após identificar que houve uma bifurcação no histórico de valores (*fork*), cada nó criará um novo dado contendo os dois valores propostos, realizando assim o *forking* que permite a coexistência de mais de um valor para uma mesma chave.

Quando uma leitura for solicitada em um nó que possui um conjunto de valores para uma determinada chave, o valor retornado será o conjunto das escritas concorrentemente realizadas. A aplicação é quem deve tomar a decisão de como proceder quanto à diversidade de valores, para realizar a convergência dos valores diferentes. Algumas estratégias sugeridas para as aplicações são: filtragem (exemplo: considerar o valor que provém do nó de maior identificador) ou substituição (exemplo: realizar uma computação dos valores retornados e utilizar este resultado). Nós faltosos são excluídos do sistema mediante provas de comportamento errôneo (*proofs of misbehavior* - POM). Essas provas

tem por base as escritas assinadas, a partir das quais é possível verificar comportamento malicioso. Seria possível, por exemplo, verificar que o nó que iniciou a operação na figura 1.22 é um nó faltoso. Altas latências em nós são tratadas como violações no SLA (*Service Level Agreement*).

O Depot tolera faltas arbitrárias por meio do enfraquecimento da semântica de dados: qualquer dado válido é considerado. A concorrência é tratada como um fato natural: ambos os valores propostos são incluídos no conjunto de valores da chave (*fork*), e a aplicação é responsável por realizar uma operação que converta os diversos valores em um único valor (*join*). Faltas arbitrárias são reduzidas a problemas de concorrência: por meio dos históricos de valores anteriores (*hashes*), é possível detectar comportamento malicioso, e assim excluir os nós bizantinos do sistema. Os servidores no interior das partições podem ser distribuídos geograficamente, o que favorece a tolerância a faltas catastróficas.

Todo nó assina as atualizações com sua chave privada. Cada cliente escritor compartilha sua chave pública para que o dado escrito possa ser lido. Por fim, confidencialidade e privacidade não são tratados no Depot, deixando a cargo da aplicação a tarefa de ofuscar os dados. Depot também não trata questões de controle de acesso dinâmico.

Foi desenvolvida uma versão do Depot denominada Teapot, que tem por objetivo armazenar dados em nuvens simples de armazenamento, como a Amazon S3, a Microsoft Azure Storage e a Google Cloud Storage. Porém, como o Depot requer que os provedores de nuvens executem código, são apresentadas no artigo algumas funcionalidades que a nuvem deveria executar para tornar possível a utilização destas nuvens de armazenamento na implementação do Teapot. Estas funcionalidades poderiam ser executadas por nuvens ativas. Por exemplo, códigos executados em máquinas virtuais na Amazon EC2 podem acessar dados armazenados na Amazon S3 sem custos, desde que a máquina virtual esteja na mesma região do provedor de dados[27]

1.4.4. DepSky

O DepSky [5] é um sistema de armazenamento chave-valor que utiliza diversos provedores de armazenamento em nuvem (deste ponto em diante denominados apenas como provedor, por fins de brevidade) para fornecer confiabilidade e confidencialidade ao dado armazenado. Os dados são armazenados e recuperados dos provedores utilizando-se protocolos de quóruns bizantinos [26]. O DepSky é implementado sob forma de biblioteca que deve ser utilizada junto ao cliente. Dois protocolos são responsáveis por realizar o armazenamento: o DepSky-A e o DepSkyCA.

O DepSky-A realiza a replicação integral do dado. Os passos realizados para a escrita de um dado estão representada na figura 1.23. O cliente inicialmente envia o dado a todos os provedores (etapa *I*); a quantidade de provedores é definida pela regra $n = 3f + 1$, onde f é o número máximo de faltas que deseja-se tolerar no sistema, e n é o total de provedores necessários para tornar possível a tolerância a f faltas no sistema. 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 correspondentes ao

[27]http://aws.amazon.com/s3/pricing/

dado que foi armazenado (etapa *III*). Após $2f+1$ confirmações de gravação do metadado pelos provedores (etapa *IV*), considera-se a operação de escrita concluída. Para ler os dados, o cliente que usa o DepSky-A solicita a todos os provedores os metadados relativos ao dado desejado. Após obter um quorum de $2f+1$ respostas, o cliente pode solicitar a qualquer provedor a leitura do dado. Nos protocolos do DepSky, a escrita de dados é feita antes da escrita de metadados para garantir que somente após a confirmação da gravação dos dados, os metadados estarão disponíveis para consulta, por clientes que desejam ler o dado.

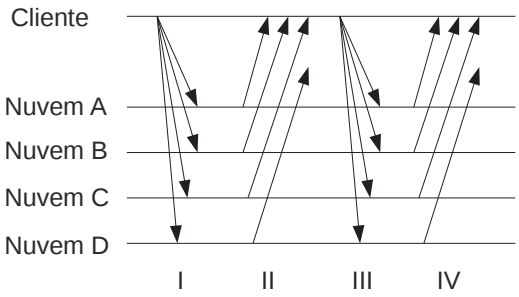


Figura 1.23: Quórum com dois *rounds*.

O protocolo DepSky-CA aplica fragmentação com redundância (seção 1.3.3.2) por meio da técnica de *erasure codes* [30] para otimizar o uso do armazenamento, reduzindo os custos de replicação. Além disso, o DepSky-CA garante a confidencialidade do dado por meio de cifragem dos dados, utilizando uma chave simétrica gerada aleatoriamente. Esta chave é distribuída entre os servidores utilizando-se a técnica de compartilhamento secreto [33] (seção 1.3.1). A fragmentação e o compartilhamento secreto são combinados seguindo uma estratégia já estabelecida [21], de forma que um bloco é composto por um fragmento do dado e um compartilhamento secreto da chave. Os blocos são os elementos distribuídos para serem armazenados nos provedores. A figura 1.24 ilustra o processo de escrita de dados nos provedores, utilizando-se o protocolo DepSky-CA. Inicialmente, o dado é cifrado (passo *I*) utilizando-se uma chave K gerada aleatoriamente. A seguir (passo *II*), o dado cifrado é fragmentado ($F1$ e $F2$) e fragmentos redundantes são gerados ($F3$ e $F4$). A chave K é codificada em compartilhamentos $S1..S4$ (passo *III*). São criados blocos de dados (passo *IV*): cada bloco agrega um fragmento e um compartilhamento. Os blocos $F1+S1$, $F2+S2$, $F3+S3$ e $F4+S4$ são enviados aos provedores. Após a escrita do dado, é realizada a escrita dos metadados, que contém o número de versão do dado e dados de verificação. A leitura de dados no DepSky-CA é realizada consultando-se blocos de dados de m provedores, tal que m é o número mínimo de fragmentos necessários para reconstruir o dado e a chave usada na cifragem.

O controle de concorrência de escrita no DepSky utiliza um mecanismo de baixa contenção: inicialmente, o escritor verifica se há um arquivo de bloqueio no formato *lock*-ID-T nos provedores, onde *ID* é o identificador de algum outro cliente que possa estar realizando a escrita, e T é uma referência tal que o tempo local t do cliente deve ser menor que $T+\delta$. O parâmetro δ é uma diferença estimada máxima entre os relógios dos outros escritores. Se um arquivo de bloqueio com este formato for encontrado, considera-se que existe outro cliente atuando como escritor; o cliente deverá aguardar por um tempo aleatório e tentar novamente realizar este procedimento. Se o escritor não

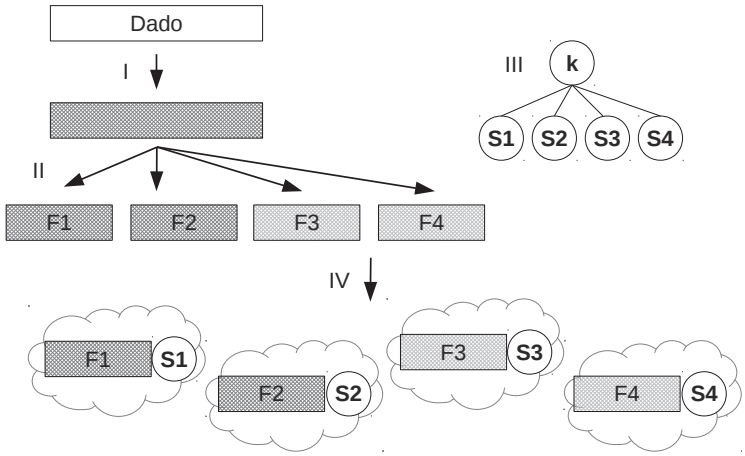


Figura 1.24: DepSky-CA: composição de técnicas na escrita de dados [5].

encontrar o arquivo segundo estas condições, ele poderá escrever o arquivo de bloqueio, definindo $T = t + tempodebloqueio$. O arquivo de bloqueio terá o formato *lock*-p-T. A seguir, o cliente busca novamente por algum arquivo de bloqueio com $t < T + \delta$. Se for encontrado algum arquivo (com exceção do arquivo escrito pelo próprio cliente), o cliente remove seus bloqueios e aguarda um tempo aleatório antes de tentar novamente a execução completa do procedimento de bloqueio.

O DepSky admite a existência de um mecanismo para compartilhamento de chaves assimétricas; no entanto, este é um dos principais desafios para garantir o sigilo de dados armazenados em provedores nuvem, quando se utiliza criptografia. O DepSky possui como ponto de vulnerabilidade a reutilização do controle de acesso fornecido pelos provedores de nuvem pública.

Foram realizados alguns testes com o Depsky, utilizando o código disponível em http://code.google.com/p/depsky/. Os testes foram feitos utilizando o armazenamento local, ao invés do armazenamento nos provedores de nuvens. O sistema operacional utilizado foi o Debian 7.0.

O arquivo obtido no site do DepSky foi o DepSky-v0.4.zip, que contém os seguintes arquivos e diretórios:

```
$ ls
bin  config  DepSky_Run.sh  dist  lib  README.txt  Run_LocalStorage.sh  src
```

Como será utilizado armazenamento local, o serviço de servidor será iniciado em um terminal específico para tal fim:

```
$ sh Run_LocalStorage.sh
```

Em um segundo terminal o cliente será ativado. Três parâmetros são necessários para executar o programa: identificador do cliente, protocolo e tipo de armazenamento. Para protocolo, utiliza-se 0 para o DepSkyA e 1 para o DepSky-CA. Para tipo de armazenamento, utiliza-se 0 para as nuvens e 1 para armazenamento local. Para acionar o cliente inicialmente com o protocolo DepSky-A, utiliza-se:


```
 $ sh DepSky_Run.sh 1 0 1
USAGE:  commands             function
       pick_du 'name' - change the container
       write 'data'   - write a new version in the selected container
       read           - read the last version of the selected container
       delete         - delete all the files in the selected container
       read_m 'num'   - read old versions written in this running. If 'num' = 0 read the last version

starting drivers...
All drivers started.
```

Seguem comandos[28] que demonstram o armazenamento e a recuperação de dados:

```
pick_du cliente
DataUnit 'cliente' selected!
write João da Silva
WRITING: João da Silva
I'm finished write -> 276 milis
read
I'm reading
I'm finished read -> 158 milis
READ RESULT = João da Silva
write Marina Lima
WRITING: Marina Lima
I'm finished write -> 241 milis
```

```
read
I'm reading
I'm finished read -> 139 milis
READ RESULT = Marina Lima
read_m 0
I'm reading
I'm finished read -> 126 milis
READ RESULT = Marina Lima
read_m 1
I'm reading
I'm finished read -> 170 milis
READ RESULT = João da Silva
```

Em um terceiro terminal, pode-se verificar os tamanhos dos dados armazenados localmente; nos arquivos de metadados existem dados binários, porém apenas os caracteres textuais foram incluídos na listagem a seguir. Note que o tamanho do arquivo *clientevalue*1001 é igual à quantidade de caracteres da frase 'João da Silva', mais um caractere de quebra de linha.

```
$ cd ds-local/cloud1/

$ ls -la
total 20
drwxr-xr-x 2 friend friend 4096 Aug 29 17:05 .
drwxr-xr-x 6 friend friend 4096 Aug 29 17:05 ..
-rw-r--r-- 1 friend friend  788 Aug 29 17:05 clientemetadata
-rw-r--r-- 1 friend friend   14 Aug 29 17:05 clientevalue1001
-rw-r--r-- 1 friend friend   11 Aug 29 17:05 clientevalue2001

$ cat clientevalue1001
João da Silva

$ cat clientemetadata
clientevalue2001
versionNumber = 2001
versionHash = 98e1d18783872be2c877837ec27801d7921646e4
allDataHash = [B@14669f26

clientevalue1001
versionNumber = 1001
versionHash = ded70e73e90c3fb8beaa443f1b1c4d5f7b52100c
allDataHash = [B@73edc5f6
```

Para a execução do protocolo DepSky-CA, utiliza-se o seguinte comando:

```
 $ sh DepSky_Run.sh 1 1 1
```

Novamente seguem comandos que demonstram a manipulação de dados:

[28]Note que a sequência está exibida em duas colunas.


```
pick_du vendedor
DataUnit 'vendedor' selected!
write Sebastião Homenish
WRITING: Sebastião Homenish
I'm finished write -> 890 milis
read
I'm reading
I'm finished read -> 172 milis
READ RESULT = Sebastião Homenish
write Suzan Croft Silva
WRITING: Suzan Croft Silva
I'm finished write -> 268 milis
read
I'm reading
I'm finished read -> 182 milis
READ RESULT = Suzan Croft Silva
read_m 1
I'm reading
I'm finished read -> 169 milis
READ RESULT = Sebastião Homenish
```

Para observar os conteúdos dos dados armazenados, foram utilizados os comandos a seguir. Note que nos metadados há informações sobre os compartilhamentos secretos (PVSS[29]) e sobre os *erasure codes* (*reed_sol_van*[30]).

```
$ cd ds-local/cloud1/

$ ls -la
total 20
drwxr-xr-x 2 friend friend 4096 Aug 30 09:45 .
drwxr-xr-x 6 friend friend 4096 Aug 30 09:44 ..
-rw-r--r-- 1 friend friend 1002 Aug 30 09:45 vendedormetadata
-rw-r--r-- 1 friend friend  685 Aug 30 09:45 vendedorvalue1001
-rw-r--r-- 1 friend friend  685 Aug 30 09:45 vendedorvalue2001

$ cat vendedorvalue1001
sr depskys.core.ECKSObject ec_bytest [BL ec_filenamet Ljava/lang/String;sk_sharet
Lpvss/Share;xpur pvss.Share Iindex[ LencryptedSharet Ljava/math/BigInteger;L
interpolationPointq~ proofcq~ proofrq~ Lshareq~ xpuq~ java.math.BigInteger bitCountI
bitLengthI firstNonzeroByteNumI lowestSetBitI signum[ magnitudeq~ xr java.lang.Number
xpsq~ xsq~ xsq~ Aqx (dados binários foram omitidos)

$ cat vendedormetadata
vendedorvalue2001
versionNumber = 2001
versionHash = 2d2ef19f772c6d67e3375c22466dc5b969389d6b
allDataHash = [B@7337e552
versionFileId = vendedorvalue2001
versionPVSSinfo = 4;2;2373168401;1964832733;1476385517
versionECinfo = 32;2 2 8 0 32;reed_sol_van;0;1;

vendedorvalue1001
versionNumber = 1001
versionHash = 3001c5b10f1d8c2ece05aa2bc9b32d46bb1deafc
allDataHash = [B@572b06ad
versionFileId = vendedorvalue1001
versionPVSSinfo = 4;2;2373168401;1964832733;1476385517
versionECinfo = 32;2 2 8 0 32;reed_sol_van;0;1;v (dados binários foram omitidos)
```

Um outro teste foi realizado com o objetivo de verificar os tamanhos dos blocos de dados utilizando-se os dois protocolos DepSky-A e DepSky-CA. Foram gerados arquivos de diversos tamanhos[31]; os conteúdos dos arquivos foram copiados e colados diante do comando *write*, na execução do cliente do DepSky.

[29]PVSS significa *public verifiable secret sharing*, e refere-se ao fato dos compartilhamentos serem verificáveis publicamente, ou seja, por entidades externas ao sistema.

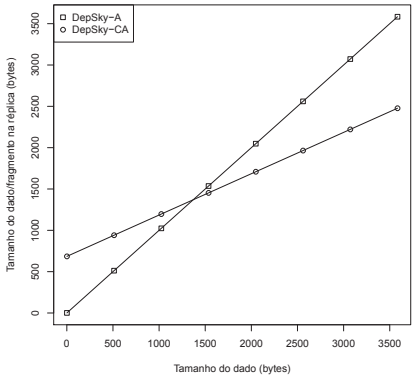
[30]*Reed Solomon* é uma técnica de geração de códigos de erro; no caso, foi especificada essa técnica para ser utilizada pelo *erasure code* na geração dos fragmentos redundantes.

[31]Como exemplo, para gerar um arquivo de 1Kb, foi usado o comando: tr -dc A-Za-z0-9 < /dev/urandom | head -c 1024 > 1k.txt

A tabela 1.25a mostra resultados do armazenamento de dados usando os protocolos DepSky-A e DepSky-CA e variando o dado de 1 bytes até 3584 bytes, com um fator incremental de 256 bytes. Naturalmente, o tamanho do bloco de dado no DepSky-A é exatamente igual ao tamanho do dado, visto que a replicação é integral. No DepSky-CA, o tamanho refere-se ao tamanho do fragmento do *erasure code*. O valor máximo do tamanho do dado igual a 3584 deve-se ao fato de a execução do cliente do DepSky, no terminal de comandos, não aceitar 4096 bytes no comando de escrita. Para testes com maiores tamanhos de dados, o DepSky deve ser usado como código de biblioteca em um programa Java. A figura 1.25b contém os dados da tabela 1.25a; é possível observar o ponto de inversão a partir do qual os dados armazenados pelo protocolo DepSky-CA apresentam vantagem, de fato, da redução de tamanho por meio do uso de fragmentos ao invés de replicar o dado integralmente.

Tamanho do dado/DepSky-A	DepSky-CA
1	685
512	941
1024	1197
1536	1453
2048	1709
2560	1965
3072	2221
3584	2477

(a)



(b)

Figura 1.25: Protocolos do DepSky e ocupação de bytes pelo dado armazenado.

1.4.5. SCFS

SCFS (*Shared Cloud-backed File System*) é uma arquitetura modular que permite acessar serviços de armazenamento chave-valor em provedores de armazenamento em nuvens (a partir deste ponto denominados provedores, para fins de brevidade) por meio da abstração de sistema de arquivos. Os provedores comerciais oferecem apenas consistência eventual, que pode não ser suficiente para alguns tipos de aplicações. O SCFS traz como principal contribuição o provimento de consistência forte utilizando provedores de consistência eventual.

Dois componentes principais compõem a arquitetura do SCFS (figura 1.26): um serviço de armazenamento (SS) e um coordenador de serviços (CA). O SS é representado por provedores passivos como a Amazon S3 e Windows Azure Storage; estes provedores contém primitivas que realizam apenas leitura e escrita de dados. O CA deve ser instanciado por servidores que tenham a capacidade de realizar processamento, como a Amazon EC2. Sistemas como o ZooKeeper[32] ou o DepSpace [7] são apropriados para realizar o serviço de coordenação que este componente requer. O cliente possui um *cache* que contém todas as informações do sistema de arquivos replicado, e possui também um agente

[32]http://zookeeper.apache.org/

do SCFS que comunica-se com o SS e com o CA.

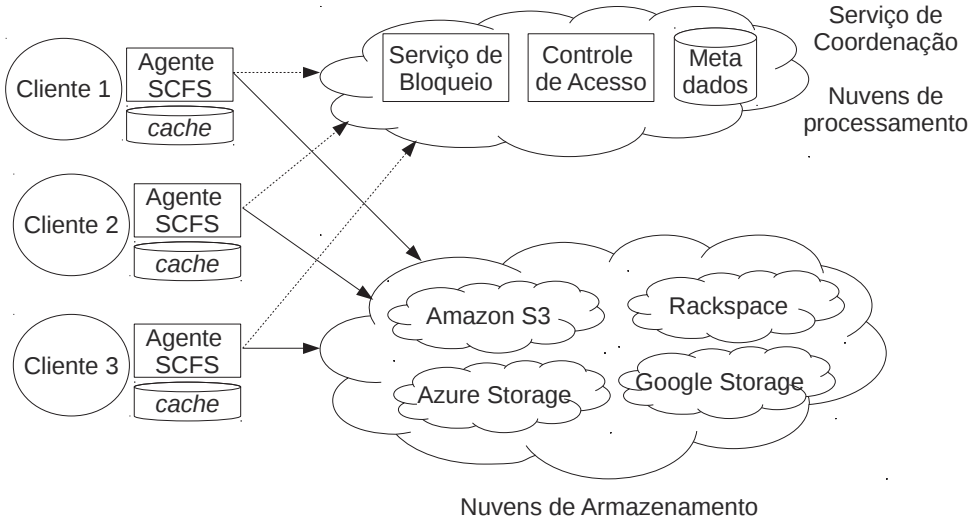


Figura 1.26: Arquitetura do SCFS [6].

O SCFS consegue prover consistência forte utilizando provedores que trabalham sob consistência eventual. Para alcançar essa característica, os procedimentos de escrita e leitura foram modificados segundo os algoritmos mostrados na figura 1.27 e representados na figura 1.28. Para realizar uma escrita de dados, considere o par chave-valor como sendo os identificadores *id* e *v*. Inicialmente, o agente gera um *hash H* de *v*; em seguida, *v* é gravado no SS, sob a chave *id+H* (concatenação de *id* e *H*). Após a gravação de *v*, *H* é gravado no CA, sob a chave *id*. Estas informações que ficam armazenadas no CA garantem a consistência forte e são chamadas de âncoras de consistência.

WRITE(id, v):	READ(id):
w1: $h \leftarrow \mathsf{Hash}(v)$	**r1:** $h \leftarrow$ CA.read(id)
w2: SS.write($id\|h, v$)	**r2:** **while** $v = null$ **do** $v \leftarrow$ SS.read($id\|h$)
w3: CA.write(id, h)	**r3:** **return** $(\mathsf{Hash}(v) = h)?v : null$

Figura 1.27: Algoritmos de escrita e leitura que alcançam consistência forte.

Para ler um dado, 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 conta com um *cache* local para otimizar as operações. Quando um usuário abre um arquivo, o agente lê o metadado (de CA), opcionalmente cria um *lock* se a operação for escrita, lê o conteúdo do arquivo (de SS) e salva este conteúdo no *cache* local. As operações de leitura e escrita (descritas a seguir) são operações locais, ou seja, operam sobre o *cache*. Quando ocorre escrita em um arquivo, é feita atualização do conteúdo em *cache* e em algumas partes do metadado (tamanho, data da última atualização). A leitura de um arquivo envolve apenas a busca do dado no *cache*, já que o conteúdo foi carregado nessa memória quando o arquivo foi aberto. Fechar um arquivo requer a

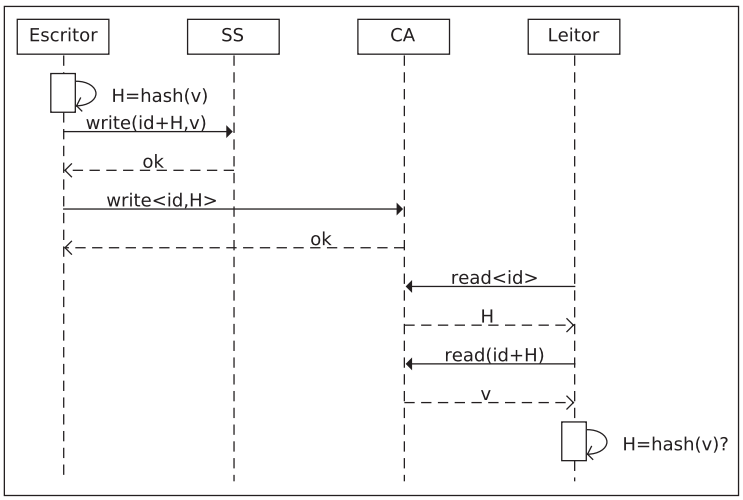


Figura 1.28: Sequência de escrita e leitura no SCFS.

sincronização dos dados e metadados. Primeiramente, o arquivo é copiado para o disco local e para as nuvens. A seguir, o metadado no *cache* é modificado, e enviado ao serviço de coordenação. Por fim, o *lock* é removido, caso o arquivo tenha sido aberto para escrita.

No SCFS, arquivos não-compartilhados são tratados de forma diferente: ao montar o sistema de arquivos, o agente SCFS obtém as entradas PNS (*Private Name Spaces*) do serviço de coordenação, e os metadados do SS, bloqueando (no CA) as PNS para evitar inconsistências, caso dois usuários estejam conectados simultaneamente com a mesma conta. Para abrir um arquivo, o usuário acessa o metadado localmente, e se necessário busca os dados dos SS. Ao fechar um arquivo modificado, dados e metadados são enviados ao SS.

Há um *garbage collector* que desativa versões antigas do dado. O sistema usa ACL's (*access control lists*) para controle de acesso. São avaliadas três versões do sistema de arquivos: bloqueante, não-bloqueante e não-compartilhado. O serviço de armazenamento pode ser o DepSky ou diretamente uma nuvem apenas. Foram usadas máquinas da Amazon EC2 para implementar a coordenação de serviços, e máquinas Amazon S3 para o serviço de armazenamento.

1.4.6. CloudSec

O CloudSec [14] é um *middleware* capaz de compartilhar documentos sigilosos utilizando provedores de armazenamento em nuvens públicas para hospedar os documentos, e módulos de segurança criptográficos[33] (MSC) - que integram uma nuvem privada - para gerenciar as chaves criptográficas. O *middleware* localiza-se em uma nuvem híbrida onde o controle de acesso é deixado a cargo da organização, que geralmente já possui um sistema de controle de acesso estabelecido. A figura 1.29 apresenta a arquitetura do CloudSec. Os gerenciadores de chaves criptográficas encontram-se instalados nos MSC, e utilizam criptografia baseada em identidades para a criação e fornecimento de chaves. Os provedores de nuvens públicas possuem o controle de acesso aos blocos de dados e

[33]Exemplo de um MSC em http://www.kryptus.com/#!asi-hsm/c11e6

são responsáveis pelo armazenamento do conteúdo dos documentos sigilosos. O usuário é responsável por editar, cifrar e decifrar os documentos. No *middleware*, o controlador contém as APIs que o usuário utiliza para manipular os documentos; o manipulador de chaves trata das operações de cifrar e decifrar, bem como a criação de chaves simétricas e assimétricas; e o manipulador de arquivos envolve o armazenamento e a recuperação de dados, interagindo com as nuvens públicas.

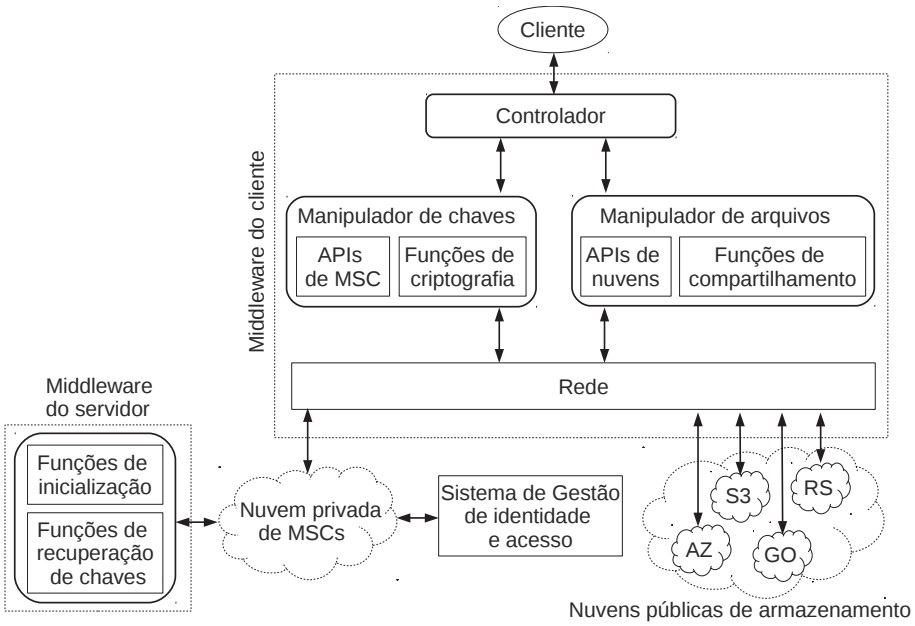


Figura 1.29: Arquitetura do CloudSec [14].

O CloudSec utiliza um mecanismo de geração de chave distribuída de maneira que nenhum dos MSC possuem a chave privada dos usuários. Ao solicitar uma chave aos MSC, é preciso que ao menos dois MSC forneçam suas respostas para que o usuário possa recuperar sua chave privada. Assim, mesmo que um MSC seja acessado indevidamente (o que é pouco provável, dadas as características inerentes desse componente), a segurança das chaves não seria comprometida, visto que um MSC não possui acesso claro às chaves privadas.

O processo de autenticação no CloudSec segue os passos ilustrados na figura 1.30. Inicialmente, para que um usuário possa ter acesso ao provedor de nuvem pública para armazenamento, o mesmo já deve ter sido cadastrado em um sistema de gestão de identidades (SGI) em uma aplicação separada. Nesse caso, os mesmos dados para autenticação serão utilizados perante os gerenciadores de chaves criptográficas (GCC). Esses irão receber os dados dos usuários (número 1 da figura) e entrarão em contato com um SGI para conferir se os dados conferem (número 2 da figura). Caso os dados estejam corretos, o SGI informará o GCC que o usuário autenticou-se corretamente e informará quais dados pode ter acesso (número 3 da figura). O GCC então irá entrar em contato com os provedores de nuvem pública para solicitar *tokens* de acesso para o usuário com as restrições necessárias (número 4 da figura). Os provedores de nuvem para armazenamento retornam tokens de acesso com as devidas restrições e um tempo de expiração (número 5 da figura). Após receber os *tokens*, o GCC então enviará os mesmos para o usuário que o uti-

lizará para ter acesso por um tempo pré-determinado de uso e com direitos limitados para acessar apenas os dados que pode ter acesso (número 6 da figura). O usuário então utilizará esse *token* para autenticar-se perante os provedores de nuvem para armazenamento e assim conseguirá ter acesso aos dados (número 7 da figura).

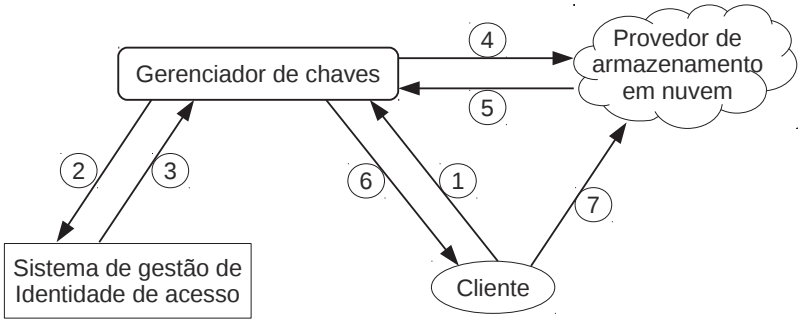


Figura 1.30: Processo de autenticação no CloudSec [14].

O SGI é compartilhado entre todos os GCCs, fazendo com que o controle de acesso não precise ser replicado. Após a autenticação, cada GCC irá emitir todos os *tokens* necessários para acessar os provedores de armazenamento em nuvem. Dessa forma, caso o usuário consiga receber corretamente os N tokens corretamente do primeiro GCC, não necessita mais receber *tokens* dos outros GCCs para se autenticar perante os provedores de nuvem para armazenamento. Cada provedor de nuvem possui uma peculiaridade com relação aos *tokens*, portanto, o GCC deverá emitir *tokens* conforme as propriedades necessárias para cada provedor de nuvem. Por exemplo, o provedor de nuvem para armazenamento da empresa Google utiliza o mecanismo OAuth 2 para a emissão do *token* de autorização; a empresa Amazon com seu serviço de armazenamento utiliza conceitos de *token* com o nome *Token Vending Machine*.

Após a obtenção do *token*, os documentos sigilosos são cifrados com a chave simétrica criada pelo usuário (aleatória). A chave simétrica é cifrada com uma chave privada que é construída a partir de um identificador *ID*, que é composto pelo nome do documento, um nome de grupo e um número de versão do documento. A chave simétrica cifrada é então codificada em compartilhamentos por meio da técnica de compartilhamento secreto (seção 1.3.1). Os dados cifrados são codificados com o mecanismo de *erasure code* (seção 1.3.3.2) para possibilitar economia no armazenamento de dados. As partes cifradas das chaves e documentos são assinadas com o algoritmo de Hess, para garantir a integridade dos dados.

Com a integração das técnicas de gerenciamento de chaves e armazenamento de dados em provedores de nuvens, o CloudSec provê os seguintes benefícios:

- Apenas o usuário possui a chave privada; as entidades distribuidoras possuem apenas componentes necessários à geração da chave privada.
- Após a retirada de um usuário do grupo, o mesmo não terá mais acesso aos documentos cifrados, nem às próximas versões do documento. Isto deve-se ao fato de o identificador do documento conter a versão do documento.
- Faltas bizantinas são toleradas, com a utilização replicada (em $3f + 1$) de provedores de dados (nuvens públicas) e provedores de chaves.

1.5. Conclusões

A computação em provedores de nuvens é uma realidade que estabeleceu-se e possui demanda crescente no espaço da Internet. A comunidade científica também avança nas pesquisas sobre esse tema, expondo trabalhos nas conferências mais relevantes da área, listadas na tabela 1.6, na percepção dos autores deste minicurso. Nota-se que os eventos que possuem tópicos como sistemas operacionais, confiabilidade e sistemas distribuídos têm tratado de artigos na área de computação em nuvens. Isto deve-se ao fato de que nos bastidores do provedor de nuvens as tecnologias relacionadas a esses tópicos são predominantes na construção da arquitetura e na operacionalização dos serviços.

Sigla	Evento	Vínculo
DSN	International Conference on Dependable Systems and Networks	IEEE
FAST	Conference on File and Storage Technologies	Usenix
NSDI	Symposium on Networked Systems Design and Implementation	Usenix
OSDI	Symposium on Operating Systems Design and Implementation	Usenix
SOSP	Symposium on Operating Systems Principles	ACM
SRDS	International Symposium on Reliable Distributed Systems	IEEE
ATC	Annual Technical Conference	Usenix

Tabela 1.6: Principais conferências com artigos sobre armazenamento em nuvens.

Um dos maiores desafios continua sendo a garantia de confidencialidade em dados e operações, em um ambiente que não pode ser considerado confiável, pois encontra-se em local não conhecido pelo cliente e sob administração de terceiros. Além disso, recentemente tem-se buscado direitos à privacidade, bem como o ”direito a ser esquecido”; empresas como a Google estão promovendo discussões[34] sobre como lidar com o armazenamento de informações na Internet de forma a permitir garantias de exclusão de dados, quando requisitadas pelo usuário.

A busca pelo equilíbrio entre o esforço para manter os dados confidenciais e a transparência na utilização do serviço remoto motiva o desenvolvimento de protocolos e arquiteturas como essas que foram explanadas neste minicurso. Como toda novidade, a computação em nuvens talvez encontre seu ponto de equilíbrio na utilização das nuvens híbridas, com parte de informações críticas, ou metadados, no controle das organizações e clientes, e volumes de dados ou cópias adicionais de segurança sob a responsabilidade dos grandes provedores de nuvens, que possuem capacidades e infraestrutura praticamente ilimitados, sendo o custo o maior fator de ponderação de sua utilização.

Referências

[1] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[2] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[34] http://www.bbc.com/news/technology-28344705

[3] Algirdas Avizienis and John PJ Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80, 1984.

[4] Rida A Bazzi and Yin Ding. Non-skipping timestamps for byzantine data storage systems. In *Distributed Computing*, pages 405–419. Springer, 2004.

[5] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *Transactions on Storage*, 9(4):12:1–12:33, November 2013.

[6] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. Scfs: a shared cloud-backed file system. In *Proc. of the USENIX ATC*, 2014.

[7] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. Depspace: a byzantine fault-tolerant coordination service. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 163–176. ACM, 2008.

[8] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *CRYPTO*, pages 213–229. Springer, 2001.

[9] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[10] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Distributed systems (2nd ed.). chapter The Primary-backup Approach, pages 199–216. ACM Press/Addison-Wesley Pub., NY, USA, 1993.

[11] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[12] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[13] George F Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed systems: concepts and design*. Pearson Education, 2012.

[14] Rick Lopes de Souza, H.V. Netto, L. C. Lung, and R. F. Custodio. Cloudsec - um middleware para compartilhamento de informacoes sigilosas em nuvens computacionais. In *XIV Simposio Brasileiro de Seguranca da Informacao e de Sistemas Computacionais - SBSEG*, Belo Horizonte, MG, 2014.

[15] Yves Deswarte, Laurent Blain, and J-C Fabre. Intrusion tolerance in distributed computing systems. In *Proc. of Symposium on Security and Privacy*, pages 110–121. IEEE, 1991.

[16] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proc. of Conference on Internet Measurement*, pages 481–494. ACM, 2012.

[17] Joni Fraga and David Powell. A fault-and intrusion-tolerant file system. In *Proc. of International Conference on Computer Security*, volume 203, pages 203–218, 1985.

[18] JL Gonzalez, Jesus Carretero Perez, Victor Sosa-Sosa, Juan F Rodriguez Cardoso, and Ricardo Marcelin-Jimenez. An approach for constructing private storage services as a unified fault-tolerant system. *Journal of Systems and Software*, 86(7):1907–1922, 2013.

[19] Glenn Greenwald and Ewen MacAskill. Nsa prism program taps in to user data of apple, google and others. *The Guardian*, 7(6):1–43, 2013.

[20] Paul T Jaeger, Jimmy Lin, Justin M Grimes, and Shannon N Simmons. Where is the cloud? geography, economics, environment, and jurisdiction in cloud computing. *First Monday*, 14(5), 2009.

[21] Hugo Krawczyk. Secret sharing made short. In *Advances in Cryptology - CRYPTO'93*, pages 136–146. Springer, 1993.

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

[23] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

[24] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[25] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM TOCS*, 29(4):12:1–12:38, 2011.

[26] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[27] Ricardo Padilha and Fernando Pedone. Belisarius: Bft storage with confidentiality. In *10th International Network Computing and Applications (NCA) Symposium on*, pages 9–16. IEEE, 2011.

[28] Ricardo Padilha and Fernando Pedone. Augustus: scalable and robust storage for cloud applications. In *Proc. of the 8th European Conference on Computer Systems*, pages 99–112. ACM, 2013.

[29] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. of the Int. Conf. on Management of Data*, SIGMOD, pages 109–116, NY, USA, 1988. ACM.

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

[31] Chhanda Ray et al. *Distributed Database Systems*. Pearson Education India, 2009.

[32] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM CSUR*, 22(4):299–319, 1990.

[33] Adi Shamir. How to share a secret. *Communic. of the ACM*, 22(11):612–613, 1979.

[34] Adi Shamir. Identity-based cryptosystems and signature schemes. In *Advances in cryptology*, pages 47–53. Springer, 1985.

[35] Andrew Tanenbaum and Maarten Van Steen. *Distributed systems*. Pearson Prentice Hall, 2007.

[36] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[37] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proc. of SOSP*, pages 292–308. ACM, 2013.

