Revista de Sistemas de Informação da FSMA
n. 15 (2015) pp. 23-29

SISTEMAS DE INFORMAÇÃO
http://www.fsma.edu.br/si/sistemas.html

TRILHA ESTUDANTIL

Implementação do Protocolo de *Lock* para o Sistema de Armazenamento DEPSKY

Poliana S. Nascimento, *Graduanda em Tecnologia em Análise e Desenvolvimento de Sistemas, IFBA*,
Allan E. S. Freitas, *Doutor em Ciência da Computação, IFBA*

***Resumo*—O armazenamento de dados em ambientes formados por diferentes nuvens computacionais (nuvem de nuvens) requer o provimento de segurança, confiabilidade e integridade. DEPSKY assegura tais características, por meio de mecanismos como criptografia, replicação de dados e codificação. Contudo, este ambiente de gerenciamento não permitia escritas simultâneas, funcionalidade desejável para o funcionamento de muitas aplicações. Diante disto, este artigo apresenta a implementação e uma análise de desempenho do algoritmo de *lock* para o sistema de armazenamento virtual DEPSKY. O mesmo também apresenta testes realizados para validar o funcionamento e desempenho do algoritmo. O protocolo proposto visa possibilitar a escrita concorrente de processos em nuvens, através de um mecanismo de trava de baixa contenção que utiliza arquivos de bloqueio para definir o escritor na unidade de dados.**

***Palavras-chave*—Computação em Nuvem, Replicação, Segurança.**

Development of the Lock Protocol for DEPSKY Storage System

***Abstract*—Data management in environments based on several clouds (cloud-of-clouds) should be dependable and secure. DEPSKY may assure that characteristics through mechanisms as cryptography and data replication, however DEPSKY does not support concurrent writing, a desirable functionality for many applications. This paper presents the development and a performance analysis of a lock algorithm for DEPSKY storage system. The paper also presents validation and performance tests of the algorithm. Such protocol allows concurrent writing, through a low contention lock mechanism that uses lock files to define who is allowed to write in a data unit.**

***Index Terms*—Cloud Computing, Replication, Security.**

I. Introdução

O uso crescente dos ambientes computacionais, integrados à Internet e com compartilhamento de informações e de recursos, torna preemente a utilização de ambientes de computação em nuvens [1]. Nestes ambientes, face ao compartilhamento de recursos, há uma crescente preocupação quanto à segurança e privacidade dos dados [2].

Diversos trabalhos da literatura abordam o problema de armazenamento em nuvens. Por exemplo, RACS (*Redundant Array of Cloud Storage*) é um *proxy* transparente de armazenamento em nuvens que distribui a carga ao longo de muitos prestadores de serviços [3]. O algoritmo proposto é tolerante a falhas e reduz o *vendor lock-in*[1], facilitando a migração entre plataformas. Contudo, não lida com problemas de segurança. Ainda, HAIL (*High-Availability and Integrity Layer*) é um sistema distribuído criptográfico que administra a integridade dos arquivos e sua disponibilidade, por meio de uma coleção de servidores ou serviços de armazenamento em nuvens [4]. Entretanto, o mesmo apresenta limitações como: a necessidade de servidores que executem código; a falta de mecanismos de proteção da confidencialidade dos dados armazenados; e o problema de trabalhar somente com dados estáticos [5].

DEPSKY é um sistema de armazenamento confiável e seguro que provê gerência de dados em computação em nuvem baseado em diferentes serviços de nuvens comerciais em um ambiente de nuvem de nuvens [6]. Desta forma, DEPSKY oferece facilidades para contornar importantes limitações de armazenamento de dados em nuvens, tais como: perda de disponibilidade, perda e alteração dos dados, perda da privacidade, e *vendor lock-in*. Em contraponto com os sistemas discorridos anteriormente o DEPSKY lida com problemas de segurança, tem mecanismos de proteção da confidencialidade dos dados armazenados e não necessita que servidores executem códigos.

Conforme o teorema CAP *(Consistency, Availability, Partition tolerance)* [7], uma base de dados distribuído só poderia atender simultaneamente a duas de três seguintes propriedades: Consistência, a garantia de que todos os dados estão atualizados em todas as réplicas; Disponibilidade (do inglês *Avaliability*), de forma que, em caso de falha em um nó, exista outro nó capaz de atender as requisições; e Tolerância de Particionamento, sendo a capacidade do sistema de funcionar mesmo quando as

Autor correspondente: Allan E. S. Freitas, *allan@ifba.edu.br*. O presente trabalho, de implementação do código proposto e sua avaliação de desempenho, foi desenvolvido pela estudante Poliana dos Santos Nascimento em estágio de pesquisa no âmbito de intercâmbio no grupo LaSIGE na Faculdade de Ciências da Universidade de Lisboa sob orientação do prof. Alysson Bessan e tutoria do doutorando Tiago Oliveira.

[1]O *vendor lock-in* está relacionado à dependência tecnológica da plataforma utilizada e torna difícil o deslocamento de clientes de um fornecedor para outro, tornando alguns provedores dominantes.

réplicas não possam se comunicar entre si.

Desta forma, DEPSKY pode prover uma forma de consistência relaxada, em face a possíveis particionamentos de rede [8], o que pode não ser problemático em cenários como os de aplicações que não necessitem o manuseio de dados sensíveis e que demandam baixo tempo de resposta, como, por exemplo, aplicações de redes sociais. O uso de plataformas adequadas de nuvens comerciais, como, por exemplo, *Amazon S3*, pode reduzir a probabilidade deste particionamento. Esta discussão não é objeto do presente artigo.

Contudo, a implementação do DEPSKY não suportava acesso a múltiplos escritores, o que é uma funcionalidade desejável para o pleno uso deste ambiente. Desta forma, este artigo apresenta o desenvolvimento de um algoritmo de *lock*, conforme proposta de [6]. Como a maioria dos sistemas de tolerância a falhas que utilizará o DepSky somente terá a existência de um nó, não se torna necessário um protocolo de concorrência de alto nível. Por isto, foi proposto um mecanismo de trava de baixa contenção utilizando arquivos de bloqueio para definir quem é o escritor na unidade de dados, permitindo concorrência.

O algoritmo desenvolvido foi avaliado por medição em um serviço de nuvem comercial, o *Amazon S3* [9]. Desta forma, foi possível testar e avaliar o desempenho do sistema diante da escrita de múltiplos clientes.

O presente artigo está estruturado da seguinte forma: a seção 2 apresenta uma breve introdução sobre o DEPSKY, discorrendo sobre os protocolos de leitura, escrita, *lock* e *unlock*, a seguir discute aspectos do desenvolvimento dos algoritmos de *lock* e *unlock*. A seção 3 expõe uma análise de desempenho e, seguida da seção 4 com as considerações finais.

II. DEPSKY e o desenvolvimento dos algoritmos de *lock* e *unlock*

O DEPSKY [6] é um sistema de armazenamento virtual que replica os dados em várias nuvens, melhorando a disponibilidade, integridade e confidencialidade das informações que são guardadas. O mesmo utiliza quóruns bizantinos e a partilha de segredos e códigos de apagamento, de modo a garantir tolerância a falhas[2] e confidencialidade. Desta forma, os usuários podem gerenciar os dados ao invocar operações nas diferentes nuvens individuais. A Figura 1 apresenta um exemplo de configuração da arquitetura do DEPSKY, no qual dois clientes mantém comunicação com quatro servidores, cada qual situado em uma nuvem distinta.

Este sistema oferece a proteção das informações mais sensíveis através de um esquema de partilha de segredos. Para isto, fornece dois protocolos que se diferem no modo como a informação é enviada para as nuvens. No protocolo ADS (*Available* DEPSKY) é enviada uma cópia da informação para todas as nuvens. No CADS (*Confidential and Available* DEPSKY), a informação é dividida em partes,

[2]DEPSKY provê tolerância a falhas bizantinas, assim, pode-se tolerar a falha de até f das n nuvens, no qual $n > 3f$.

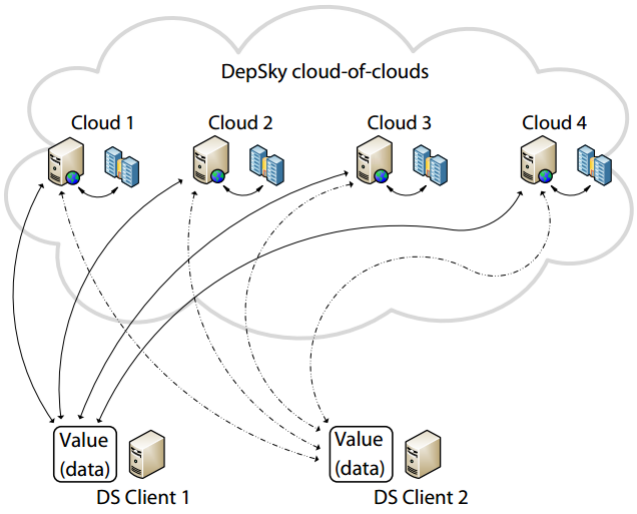

Fig. 1. Arquitetura do DEPSKY, com quatro nuvens e dois clientes [6].

conforme o número de nuvens; cada parte é enviada para uma nuvem. A informação é codificada de modo que seja necessário apenas um determinado número de partes para reconstruir a informação original, e não a totalidade das partes [6][5].

A. Protocolos de leitura e escrita

O protocolo de escrita, presente no DEPSKY, tem como ideia fundamental guardar o valor em um quórum de nuvens (unidade de armazenamento de dados), em seguida gravar os metadados correspondentes, assegurando que apenas um escritor (também conhecido como cliente, processo que deseja realizar o bloqueio da nuvem para acionar o mecanismo de escrita) seja capaz de ler os metadados para o valor armazenado. Quando um cliente escreve a primeira unidade de dados (objeto de armazenamento básico com o qual o algoritmo trabalha, contém um nome único, um número de versão, a data de verificação e os dados a serem armazenados no objeto), o mesmo entra em contato com as nuvens, para adquirir os metadados com o maior número de versão, atualizando a variável que guarda o número da versão [6].

O protocolo de leitura, presente no DEPSKY, busca os arquivos de metadados em um quórum de nuvens e escolhe o que tem o maior número de versão. Depois, o mesmo procura a versão da unidade de dados que corresponde a este número de versão e o *hash* criptográfico encontrado nos metadados. Em seguida, o protocolo busca um arquivo que contém o valor correspondente a síntese dos metadados em um quórum de nuvens. Caso essas condições sejam satisfeitas, o processo sai do *loop* e retorna o valor [6].

B. Protocolos de lock e unlock

O protocolo de *lock* desenvolvido é um mecanismo de bloqueio de baixa contenção, baseado em um arquivo que especifica qual é o escritor e por quanto tempo ele tem permissão de gravação na unidade de dados. O mesmo

pode ser ilustrado na Figura 2, que apresenta os algoritmos de *lock* e o de *unlock*. Os algoritmos funcionam da seguinte forma: quando um processo *c* quer se tornar escritor, ele lista os arquivos que estão na nuvem e busca por um arquivo chamado *lock-c'-T'*, no qual *c'* é o identificador do processo associado e *T'* é o tempo que este processo obteve a permissão de escrita (linhas 5-10). Se este arquivo for encontrado na unidade de dados, significa que tem algum processo bloqueando a mesma, então o processo *c* hiberna por um tempo aleatório (linha 21). Caso contrário, *c* pode escrever o arquivo de bloqueio, contendo uma assinatura criptográfica do nome do arquivo em todas as nuvens (linhas 11 e 12). A estratégia de retirada é necessária para assegurar que dois ou mais processos não obtenham sucesso no acesso ao mesmo tempo [6].

Por fim, o processo *c* lista novamente todos os arquivos da nuvem em busca de arquivos de bloqueio (linhas 13-17), se ele encontrar um arquivo que não seja o seu e que o tempo de escrita não expirou, *c* remove o seu arquivo de *lock* e hiberna por um dado período de tempo[3]. Caso contrário, *c* torna-se o único escritor para a unidade de dados. O desbloqueio (*unlock*) é realizado através da remoção do arquivo de *lock* pelo protocolo de *unlock* (linhas 24-27)[6].

É importante observar que o procedimento de desbloqueio não é completamente tolerante a falhas. Para liberar a escrita de um arquivo de *lock*, o mesmo deve ser excluído de todas as nuvens, mas uma nuvem pode falhar mostrando ainda o arquivo que deveria ter sido removido, no caso da função *unlock* falhar no momento da exclusão, não permitindo aquisição de bloqueio por outro escritores, entretanto, como um arquivo de bloqueio tem validade, este problema só pode afetar o sistema durante um dado período de tempo, pois o mesmo irá expirar [6].

C. Aspectos de desenvolvimento dos algoritmos

Os algoritmos de *lock* e de *unlock* foram implementados em funções correspondentes na linguagem de programação *JAVA*, assim como todo o DEPSKY, pois a mesma fornece uma variedade de bibliotecas e serviços que facilitam a leitura e escrita nas nuvens utilizadas [5] [10]. Como funções auxiliares para este algoritmo, foram desenvolvidas as funções *listQuorum*, *writeQuorum* e *deleteData*[4].

A função *listQuorum* tem por objetivo listar todos os arquivos de *lock* que estão nas nuvens. A mesma é utilizada pela função *lock* para trazer a relação de arquivos de bloqueio. A função *writeQuorum* escreve nas nuven s o arquivo de *lock*, utilizando uma assinatura criptográfica SHA-1 do nome do arquivo que contém o *timestamp* do momento em que o processo conseguiu realizar o bloqueio. A função *deleteData* é utilizada pela função *unlock* e remove o arquivo de *lock* das nuvens. Todas as mensagens são transmitidas com criptografia simétrica, utilizando o algoritmo AES.

A função *unlock* remove o arquivo de bloqueio, liberando as nuvens para que outros processos tenham o direito de escrita. Para a função *unlock* foi necessário realizar algumas modificações no algoritimo original da Figura 2, ao invés de passar somente o *lock_id*, foi necessário também passar a unidade de dados (*DataUnit*), pois a função *deleteData* exige que esse parâmetro seja passado como para realizar a exclusão.

Na função *lock* [5] é passado como parâmetro a *DataUnit* e o número de vezes que um processo quer tentar bloquear a nuvem. O mesmo verifica se algum processo está bloqueando a unidade de dados. Primeiro é invocada a função *listQuorum* para verificar se existe algum arquivo de *lock* na nuvem. Se retornar *null*, é chamada a função *writeQuorum*. Depois é chamada novamente a função *listQuorum* para verificar se existe algum arquivo de *lock* além do que o cliente escreveu. Se outro processo escreveu na nuvem, o cliente remove seu arquivo de *lock* e hiberna por um tempo aleatório, se não ele se torna o único escritor para a unidade de dados.

Se a lista não estiver vazia, é verificado se o arquivo contido na nuvem é da unidade de dados selecionada, se pertence ao processo que está tentando escrever e se a assinatura criptográfica é válida. Caso sejam satisfeitas as condições, ele bloqueia a unidade de dados e se torna o único escritor, caso contrário, ele verifica se o tempo do arquivo de bloqueio que está na nuvem expirou, se isto se confirmar, ele remove o arquivo de *lock* da nuvem e escreve o seu, se não ele hiberna por um tempo aleatório.

Algumas observações podem ser feitas acerca do protocolo e da sua implementação. O bloqueio é renovado periodicamente para garantir a existência de um único escritor a cada momento da execução. Se vários processos tentarem se tornar escritores, ao mesmo tempo, é possível que nenhum deles obtenha sucesso. Devido à estratégia de recuo utilizado, assumimos como premissa que esta condição de nenhum processo obter sucesso não seja frequente [6] – mesmo neste caso, os clientes podem resubmeter posteriormente o seu pedido de escrita.

O protocolo não garante terminação, entretanto, quando um cliente solicita o *lock*, uma variável é passada como parâmetro para informar o número de vezes que um processo tentará obter o bloqueio, isto garante que o mesmo não fique tentando incessantemente sem sucesso – evitando *starvation*. Entretanto, as propriedades de progresso não são garantidas, visto que o mesmo, por ter uma quantidade limitada de tentativas de bloqueio, pode vim a não obter sucesso na escrita. O protocolo de *lock* assume como premissa que os relógios são sincronizados periodicamente.

A Figura 3 ilustra o funcionamento do protocolo de *lock*, para o caso do cliente conseguir na primeira tentativa ser o escritor da unidade de dados. Primeiro, o mesmo solicita

[3]Este período é definido de forma aleatória em tempo de execução, conforme um limite temporal pré-definido na aplicação.

[4]Os códigos das funções *listQuorum*, *writeQuorum* e *deleteData* estão disponíveis no seguinte endereço eletrônico: https://code.google.com/p/depsky/source/browse/trunk/DepSky/src/depskys/core/LocalDepSkySClient.java

[5]Os códigos das funções *lock* e *unlock* estão disponíveis no seguinte endereço eletrônico: https://code.google.com/p/depsky/source/browse/trunk/DepSky/src/depskys/core/LocalDepSkySClient.java

ALGORITMO 1: DEPSKY bloqueio da unidade de dados pelo escritor c

```
1  function DepSkyLock(du)
2  begin
3      lock_id ← ⊥
4      repeat
           // lista os arquivos de lock de todas as nuvens para ver se a unidade de dados está bloqueada
5          L[0 .. n − 1] ← ⊥
6          parallel for 0 ≤ i ≤ n − 1 do
7              L[i] ← cloud_i.list(du)
8          wait until (|{i : L[i] ≠ ⊥}| > n − f)
9          for 0 ≤ i ≤ n − 1 do cloud_i.cancel_pending()
10         if ∄i : ∃lock-c'-T' ∈ L[i] : c' ≠ c ∧ valid(L,lock-c'-T',du) ∧ (T' + Δ > local_clock) then
               // cria o arquivo de lock para a unidade de dados e escreve o mesmo na nuvem
11             lock_id ← "lock-"+c+"-"+(local_clock + LEASE_TIME)
12             writeQuorum(du, lock_id, sign(lock_id, K_r^du))
               // lista os arquivos de lock novamente para para verificar condição de disputa
13             L[0 .. n − 1] ← ⊥
14             parallel for 0 ≤ i ≤ n − 1 do
15                 L[i] ← cloud_i.list(du)
16             wait until (|{i : L[i] ≠ ⊥}| > n − f)
17             parallel for 0 ≤ i ≤ n − 1 do cloud_i.cancel_pending()
18             if ∃i : ∃lock-c'-T' ∈ L[i] : c' ≠ c ∧ valid(L,lock-c'-T',du) ∧ (T' + Δ >
19                 DepSkyUnlock(lock_id)
20                 lock_id ← ⊥
21         if lock_id = ⊥ then dorme por algum tempo
22     until lock_id ≠ ⊥
23     return lock_id

24 procedure DepSkyUnlock(lock_id)
25 begin
26     parallel for 0 ≤ i < n − 1 do
27         cloud_i.delete(du,lock_id)

28 predicate valid(L,lock-c'-T',du) ≡ (|{i : lock-c'-T' ∈ L[i]}| > f ∨ verify(lock-c
```

Fig. 2. Algoritmos de *lock* e de *unlock*[6].

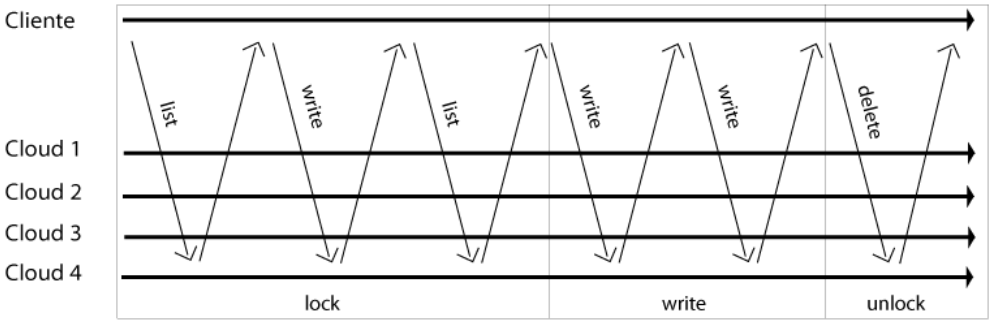

Fig. 3. Processo de escrita utilizando o protocolo de *lock*.

as nuvens que retornem uma lista dos arquivos de *lock*. Depois, escreve o arquivo de *lock* e solicita novamente a lista de arquivos de bloqueio para verificar se outro cliente tentou escrever na nuvem. Após este procedimento, ele utiliza o protocolo de *write* e finaliza, chamando a função *deletedata* para desbloquear a nuvem.

III. Análise de desempenho

Esta seção trata dos aspectos da análise de desempenho.

A. Ambiente de experimentação

O cliente foi executado em uma computador *desktop* com o sistema operacional *GNU/Linux Ubuntu 12.0*, processador *Intel Core i5*, 4 GB de memória. Para armazenamento dos dados, foram criados 4 *Buckets* no *Amazon S3*[6]

[6]O *Amazon S3 – Simple Storage Service* – é uma nuvem do tipo *PaaS*, que provê um serviço de armazenamento de dados na forma de unidades denominadas *Buckets*, por meio de uma interface simples de serviços da web que pode ser usada para armazenar e recuperar qualquer quantidade de dados, a qualquer momento e de qualquer lugar na web. [9]

[9]. Cada um destes *Buckets* foram criados em diferentes localizações: São Paulo, *Ireland*, *Stansted* e *EUA West*.

A localização distinta dos *Buckets* assegura independência de falhas. A quantidade de ao menos 4 *Buckets* é a necessária para prover um quórum tolerante de até 1 *Bucket* com falha bizantina (i.e. $N > 3F$).

Para a realização dos testes foram coletados dados referentes ao tempo das operações de *lock*, *write*, *unlock* e o tempo total. Para a medição, foi considerado que cada cliente, depois de conseguir escrever o arquivo de *lock* na nuvem, invoca a operação de *write*, seguida da operação de *unlock*.

B. Critérios analisados

- *Tempo de escrita na nuvem*: Foi analisado quanto tempo um processo leva para realizar as operações de *lock*, *write* e *unlock*. Foi observado o tempo gasto confrontado com o aumento do número de processos que concorrem para realizar as operações de escrita na nuvem. Primeiro, o cliente aciona o protocolo de *lock*, caso a nuvem esteja bloqueada, este hiberna, caso contrário, solicita o bloqueio da mesma, e, logo em seguida, executa a operação de escrita, seguida da operação de *unlock*, para desbloquear a nuvem e liberar a mesma para que outros clientes possam acessá-la.
- *Falhas de processos*: Nesta análise, um processo falha ao tentar realizar o bloqueio da nuvem e não aciona o mecanismo de desbloqueio, impedindo que outros processos tentem executar o protocolo *lock*. Foram realizados também experimentos com o objetivo de verificar se o processo pode identificar quando um arquivo de bloqueio pertence ou não ao mesmo.
- *Falha na nuvem por invasão/corrupção dos dados*: Nesta análise, a assinatura criptográfica de mais de f nuvens foram modificadas de forma indevida, no qual f é o número máximo de nuvens falhas de acordo com o quórum utilizado. Se o processo identificar o problema, deve retirar o arquivo de bloqueio para liberar a nuvem.

C. Resultados e discussão

1) Tempo de escrita na nuvem: Nesta análise, foi utilizado como fator de medição o número de processos (1, 2, 3 e 4). Foram definidos como parâmetros: o número de tentativas que um cliente tentou acionar o mecanismo de bloqueio (20 tentativas), este número foi estipulado *ad-hoc* de modo a minimizar a possibilidade dos clientes finalizarem sua execução sem obter sucesso; e o tempo de hibernação de um processo ao aguardar sua vez, este determinado por um valor fixo ($1000ms$) acrescido de um valor aleatório.

Os gráficos de distribuição cumulativa das operações de *lock*, *unlock* e tempo total são apresentados nas Figuras 4, 5 e 6, respectivamente, expondo a frequência (eixo vertical) acumulada da ocorrência de uma dada operação com a latência (eixo horizontal) observada em segundos. Os gráficos permitiram um comparativo da latência de acordo com o aumento da quantidade de clientes. O valor de frequência igual a 1 indica que todas as operações ocorrem em até a latência indicada. Cada linha representa o variação da latência de acordo com a quantidade de clientes.

Foi possível observar que quando o número de clientes aumenta, o tempo total requerido para a conclusão da operação cresce implicando em uma perda de desempenho, já que quanto mais clientes tentam acessar a nuvem, maior é o tempo para cada processo finalizar a escrita.

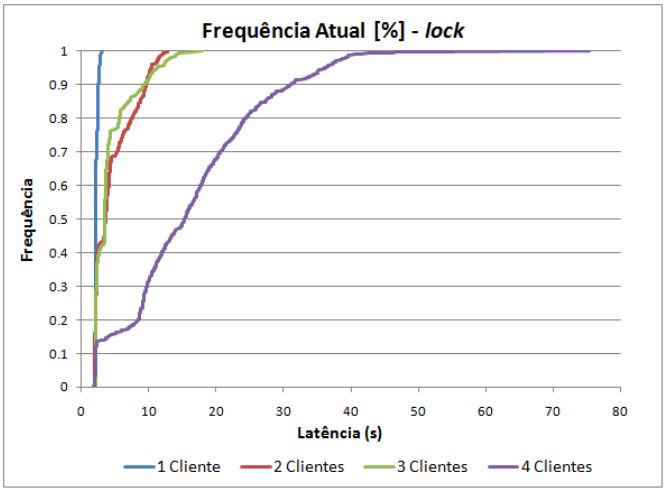

Fig. 4. Gráfico de distribuição cumulativa com um, dois, três e quatro clientes - operação *lock*.

A Figura 4 indica que a perda de desempenho na operação de *lock* decorre do aumento do número de clientes que tentavam bloquear a nuvem. Observou-se que a latência aumentou com a quantidade de processos concorrendo para escrita na unidade de dados. Foi analisado que no decorrer do aumento da quantidade de clientes, até mesmo as operações que não estavam concorrendo para escrever na nuvem, como a *write* e *unlock*, se tornaram mais ociosas, já que mais de um processo tentava ter acesso a nuvem, ocasionando perda de eficiência, o que pode ser ilustrado na Figura 5 nos gráficos de distribuição cumulativa para a operação de *unlock*.

Ao analisar o gráfico de distribuição cumulativa do *unlock*, apresentado na Figura 5, foi possível perceber que o tempo da operação de *unlock* variou, em uma margem tolerável, crescendo gradativamente com o aumento do número de clientes. O desempenho da função de *lock* caiu mais rapidamente que a de *unlock*, devido a concorrência para realizar o bloqueio da nuvem por parte dos clientes, o que não ocorreu no protocolo *unlock*, visto que o cliente já se tornou o escritor.

A Figura 6 apresenta o gráfico de frequência para o tempo total com as três operações (*lock, write e unlock*). Conforme discorrido anteriormente, é perceptível o aumento da latência no decorrer do aumento da quantidade de clientes. Mas é importante relatar que, com a utilização do protocolo de *lock* o tempo para realizar a escrita na

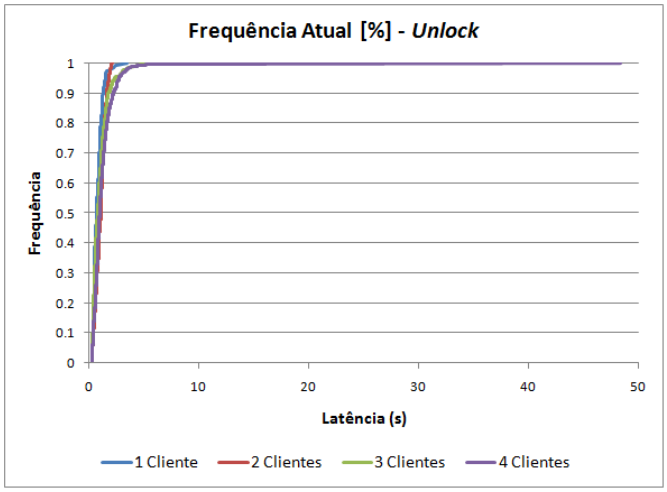

Fig. 5. Gráfico de Distribuição Cumulativa com um, dois, três e quatro clientes - Operação *unlock*.

unidade de dados se torna maior, já que foram utilizadas três operações para a concretização do processo, no qual os escritores entram em condição de corrida, implicando no ganho de funcionalidades para o sistema, mas em contraponto teve-se uma perda no requisito de eficiência do sistema.

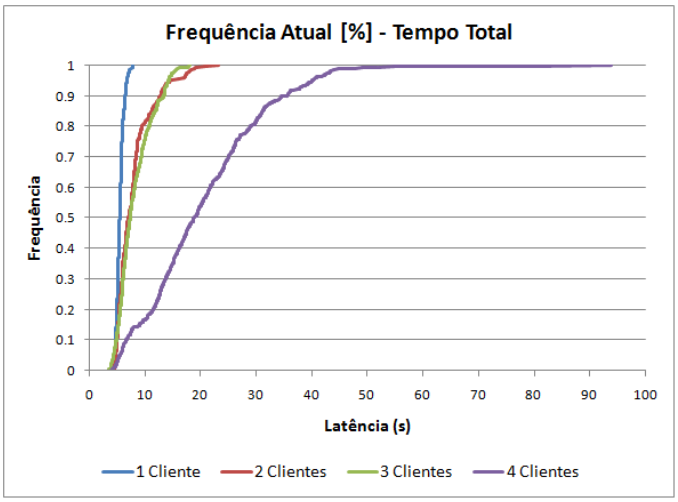

Fig. 6. Gráfico de Distribuição Cumulativa com um, dois, três e quatro clientes - Tempo total das operações *lock*, *write* e *unlock*.

A Figura 6, por apresentar a frequência do tempo total de todas as análises em um gráfico, permite fazer uma melhor comparação dos resultados obtidos do experimento com um, dois, três e quatro clientes.

A perda do desempenho, ocasionado pelo aumento do número de processos concorrendo para escrita, pode ser simplificada na Figura 7, que apresenta uma comparação da latência no decorrer do aumento de escritores para as operações de *lock e unlock* e para o tempo total do processo. Os gráficos também expõem o desvio padrão. Por fim, foi observado, que o aumento do tempo total foi ocasionado principalmente pela concorrência para os processos escreverem o arquivo de *lock* na unidade de dados.

2) Falha do processo: Nesta análise, foi analisado como o protocolo de *lock* se comportou diante da falha de um processo durante a escrita, impossibilitando que o mesmo acione o mecanismo *unlock*. Foram testados dois processos em execução. Um deles falhou, quando realizava a escrita na nuvem, enquanto o outro tentava bloquear a nuvem, mas não conseguia, pois existia um arquivo de *lock* na nuvem. Diante disto, foi possível analisar que decorrente do princípio do protocolo *lock*, no qual um processo somente pode bloquear uma nuvem por um dado período de tempo[7], mesmo que o cliente não realize o desbloqueio, quando seu tempo expirar outro processo poderá remover o seu arquivo de *lock*, realizando então o desbloqueio da nuvem.

3) Falha na nuvem por invasão/corrupção dos dados: Nesta análise, simulou-se um comportamento arbitrário em que duas das quatro nuvens executaram de forma maliciosa, sendo verificado pelo processo por meio da assinatura criptográfica informada pelas nuvens. Deve-se notar que nesta execução, o quórum bizantino não foi atendido, uma vez que apenas uma das nuvens pode falhar para atender aos requisitos. Como somente duas nuvens responderam o arquivo corretamente, não se satisfez o quórum de ao menos *2f+1* nuvens, desta forma o cliente excluiu o arquivo de *lock* contido nas nuvens, conforme esperado.

IV. Conclusão

Este artigo apresenta o desenvolvimento e uma análise de desempenho de um algoritmo de bloqueio (*lock*) para o sistema de armazenamento DEPSKY. O protocolo de *lock* possibilita a escrita de mais de um cliente na unidade de dados, através de um mecanismo de bloqueio de baixa contenção, utilizando um arquivo que contém as informações de quem é o escritor da unidade de dados.

A partir da análise dos resultados obtidos, foi possível observar que o algoritmo desenvolvido satisfaz as condições para realizar o bloqueio das nuvens, impedindo que mais de um cliente escreva ao mesmo tempo na unidade de dados. Ainda, foi possível verificar como o desempenho degrada em face à escrita concorrente: quanto maior o número de clientes que tentam acessar as nuvens para escrita, maior a sobrecarga dos serviços e maior a latência.

Referências

[1] Q. Zhang, L. Cheng e R. Boutaba, "Cloud computing: state-of-the-art and research challenges", *Journal of internet services and applications*, vol. 1, nº 1, pp. 7–18, 2010.

[2] E. Hanna, N. Mohamed e J. Al-Jaroodi, "The cloud: requirements for a better service", *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 787–792, 2012.

[7]Este período é definido de forma aleatória em tempo de execução, conforme um limite temporal pré-definido na aplicação.

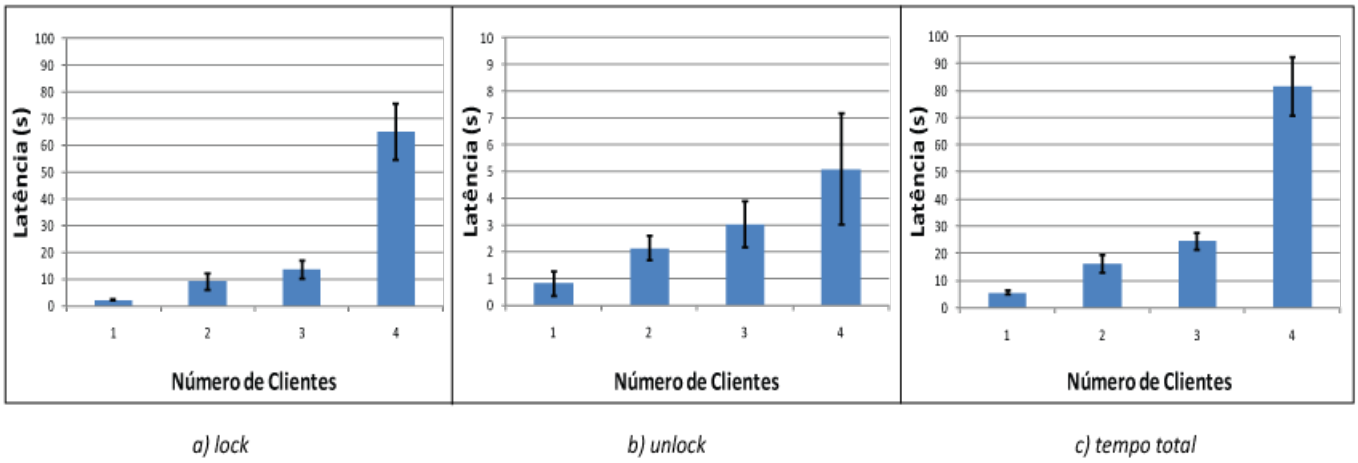

Fig. 7. Gráfico de Desvio Padrão, comparação com um, dois, três e quatro clientes - Operações lock, unlock e tempo total.

[3] H. Abu-Libdeh, L. Princehouse e H. Weatherspoon, "Racs: a case for cloud storage diversity", em *Proceedings of the 1st ACM symposium on Cloud computing*, ACM, 2010, pp. 229–240.

[4] K. D. Bowers, A. Juels e A. Oprea, "Hail: a high-availability and integrity layer for cloud storage", em *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, 2009, pp. 187–198.

[5] B. M. M. R. Quaresma, "Depsky: sistema de armazenamento em clouds tolerante a intrusões", diss. de mestrado, UNIVERSIDADE DE LISBOA. Faculdade de Ciências. Departamento de Informática, 2010.

[6] A. Bessani, M. Correia, B. Quaresma, F. André e P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds", *ACM Transactions on Storage (TOS)*, vol. 9, nº 4, p. 12, 2013.

[7] L. Frank, R. U. Pedersen, C. H. Frank e N. J. Larsson, "The cap theorem versus databases with relaxed acid properties", em *Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication*, ACM, 2014, p. 78.

[8] M. Correia, "Clouds-of-clouds for dependability and security: geo-replication meets the cloud", em *Euro-Par 2013: Parallel Processing Workshops*, Springer, 2014, pp. 95–104.

[9] *Amazon web services. available at https://console.aws.amazon.com/console/home.*

[10] *Depsky: a cloud-of-clouds storage middleware. available at https://code.google.com/p/depsky/.*