Exclusão Mútua Tolerante a Faltas Bizantinas na Cloud

Ricardo Mendes, Tiago Oliveira, Alysson Bessani

LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
{rmendes,toliveira}@lasige.di.fc.ul.pt,anbessani@fc.ul.pt

Resumo Recentemente, uma série de sistemas têm sido propostos para o armazenamento e partilha de dados em múltiplos serviços de cloud. Estes trabalhos usualmente requerem o uso de algoritmos de exclusão mútua para permitir aos clientes reservar recursos por determinados períodos de tempo com vista a evitar o uso concorrente desses recursos. Neste artigo apresentamos um novo protocolo de exclusão mútua que tira partido de vários objetos de trinco implementados em diferentes serviços da cloud para garantir o correcto funcionamento do protocolo mesmo na presença de provedores de cloud faltosos. Este novo protocolo é *centrado em dados*, não necessitando portanto de nenhum servidor específico a correr nas clouds para além dos serviços propriamente ditos. Outra contribuição deste trabalho é a implementação e descrição de vários objetos de trinco que usam diferentes serviços de cloud no mercado. Apresentamos ainda um estudo dos custos inerentes ao uso tanto do novo protocolo de exclusão mútua como das referidas implementações de objetos, assim como uma avaliação de desempenho dos mesmos em diferentes contextos de utilização.

Palavras-chave: exclusão mútua, coordenação, computação em clouds, tolerância a faltas bizantinas.

1 Introdução

Nos últimos anos têm surgido um conjunto de sistemas que propõem o uso de múltiplos serviços de armazenamento na cloud (e.g., Amazon S3, Microsoft Azure Storage, Google Storage, Rackspace Files) para armazenar e partilhar dados [9,11,12,16]. Nestes sistemas, os dados a serem armazenados na cloud são replicados por múltiplos serviços - chamados *cloud-of-clouds* - de tal forma que os dados possam sobreviver a falhas nos serviços de clouds. A maioria desses sistemas consideram um modelo de dados em que cada item de dados (ou ficheiro) pode ser escrito por apenas um utilizador de cada vez, i.e., não suportam escritas concorrentes num mesmo item de dados.

A fim de garantir que dois escritores nunca tentam escrever num mesmo item de dados, esses algoritmos assumem a existência de um algoritmo de exclusão mútua. Alguns sistemas como o RACS [9] e o SCFS [12] usam serviços de coordenação (e.g., Zookeeper [19]) para concretizar essa primitiva. Esta primitiva permite a partilha controlada de ficheiros entre os clientes destes sistemas.

Outros sistemas, como é o caso do DepSky [11], incluem algoritmos de exclusão mútua desenhados especificamente para um ambiente de multiplas clouds. A grande vantagem destes algoritmos em relação ao uso de sistemas de coordenação reside no fato de eles não requererem a execução de servidores, sendo executados directamente sobre as clouds de armazenamento. Por outro lado, um dos seus problemas fundamentais prende-se com o seu baixo desempenho: para obter acesso à zona crítica, são necessários três acessos à cloud, mesmo em execuções sem acessos concorrentes.

Neste artigo apresentamos um novo algoritmo centrado em dados para a cloud-of-clouds. Diferentemente do algoritmo proposto no DepSky, o nosso algoritmo considera o uso de diferentes tipos de serviços de cloud (e.g., Amazon DynamoDB, Azure Queue, Google Datastore), ao invés de apenas serviços de armazenamento de objetos (e.g., Amazon S3). Além disso, o algoritmo é construído de forma modular: inicialmente construímos algoritmos de exclusão mútua base em cima de cada serviço de cloud que usamos, e depois combinamos estes algoritmos numa versão composta que tolera faltas bizantinas em até um terço dos serviços empregados. Desta forma, o algoritmo proposto apresenta um desempenho muito melhor do que o usado no DepSky.

Outra vantagem importante do nosso algoritmo é o fato de ter poucos requisitos de sincronia. A maioria dos algoritmos de exclusão mútua e de gestão de trincos na presença de processos faltosos requer hipóteses temporais para garantir a *safety* do sistema. O nosso algoritmo requer tais hipóteses apenas para garantir *liveness*, tornando-o muito mais apropriado para ser usado na internet.

Em sumário, este artigo apresenta as seguintes contribuições:

1. um conjunto de algoritmos de exclusão mútua eficientes construídos sobre diferentes serviços de cloud;
2. um algoritmo de exclusão mutua tolerante a faltas bizantinas que faz uso de serviços disponíveis na internet;
3. as provas de correção de todos os algoritmos mencionados anteriormente e uma avaliação de custo e desempenho dos mesmos tendo em conta vários serviços de cloud reais.

Os algoritmos apresentados neste artigo estão a ser usados no sistema de ficheiros distribuído CHARON, usado para integrar biobancos e bioinformáticos no contexto do projeto BiobankCloud [13].

2 Trincos na *Cloud-of-Clouds*

Locks são contratos usados para controlar acessos concorrentes a recursos (e.g. um ficheiro), prevenindo assim conflitos de versões. *Leases* [17], tal como os *locks*, são também contratos usados para coordenar o acesso a recursos partilhados, mas com uma noção de tempo de validade que permite que um recurso reservado por um cliente volte a ficar disponível em caso de falha deste após esse período de validade expirar. Doravante neste artigo, adoptamos esta última definição para o termo português *trinco*.

Sistemas como o RACS [9] e o SCFS [12] coordenam acessos concorrentes a recursos recorrendo a algoritmos de exclusão mútua feitos em cima de serviços de coordenação [19] específicos instalados em VMs nas clouds. Por sua vez, os algoritmos de exclusão mútua centrados em dados e tolerantes a faltas existentes (e.g., [11]) são desenhados para funcionar em cima de serviços de armazenamento de objetos. O problema destas soluções são, no primeiro caso os custos de manter as VMs, e em ambos a sua falta de modularidade. Neste artigo nós propomos um algoritmo centrado em dados e tolerante a faltas bizantinas mas modular pois permite tirar partido de qualquer serviço fornecido pelas clouds (não só o de armazenamento). Na prática, utilizamos um protocolo que utiliza $3f + 1$ objetos que permitem a gestão de trincos (aos quais chamamos *objetos de trinco base*) para implementar um objecto (*objecto de trinco composto*) que tolera f faltas por parte dos serviços de cloud utilizados pelos objetos base. Esta capacidade é importante pois, desta forma, é possível usar serviços de cloud com melhor desempenho ou com suporte a funções com maior poder de sincronização [18] para melhorar o desempenho do algoritmo.

O algoritmo de trinco do DepSky tem ainda a limitação de necessitar de sincronização dos relógios entre os clientes. A nossa solução não tem esta limitação pois, ao invés de utilizar valores dos relógios dos clientes utiliza os das clouds (§ 4).

2.1 Modelo e Garantias do Sistema

O nosso modelo de sistema é equivalente a outros modelos usados nos algoritmos tradicionais centrados em dados e tolerantes a faltas bizantinas (e.g., [8,11]). Um número ilimitado de clientes podem aceder a um conjunto de objetos base (e.g., serviços de cloud) que compreendem o serviço de exclusão mútua. Todos esses clientes, e no máximo f objetos base, podem sofrer faltas bizantinas. Cada um dos objetos base fornece mecanismos de controlo de acesso, de forma a ser garantido que somente clientes autorizados podem invocar operações sobre eles [21]. Uma vez que a noção de trinco implica garantias de tempo, é assumido um limite máximo de tempo para a transmissão de mensagens entre os clientes e os objetos base. Contudo, *esta assumpção é só necessária para a propriedade de liveness.*

O nosso algoritmo garante exclusão mútua (sempre segura) pois garante que, em cada momento, está no máximo um processo a aceder ao recurso partilhado e que esse processo só vai aceder ao recurso numa quantidade de tempo limitada. Cada recurso fornece três operações de trinco diferentes: `obter(T)`, `renovar(T)` e `libertar()`. As operações `obter(T)` e `renovar(T)` têm a função de adquirir e renovar o trinco por T segundos, respectivamente. Por sua vez, a operação `libertar()` serve para terminar o trinco. Estas operações satisfazem as seguintes propriedades:

- *Exclusão Mútua (safety):* Nunca existem dois clientes correctos com um trinco válido em simultâneo.
- *Liberdade de Obstrução (liveness):* Um cliente correcto que tente adquirir o trinco sem concorrência terá sucesso.
- *Limitação Temporal (liveness):* Um cliente correcto que obtém o trinco vai possui-lo por um máximo de T unidades de tempo, excepto se este for renovado.

Estas propriedades não impedem um cliente bizantino de adquirir o trinco e renová-lo indefinidamente, nem impedem o acesso direto ao recurso sem um trinco válido. No entanto, isto é aceitável pois um cliente malicioso pode sempre danificar todos os recursos (e.g., ficheiros) aos quais tenha acesso. Do mesmo modo, se um cliente falha enquanto possui o trinco, este estará disponível novamente passados, no máximo, T unidades de tempo (propriedade de *limitação temporal*).

O nosso algoritmo não satisfaz nenhuma propriedade mais forte que as descritas acima pois isso implicaria um número superior de acessos às clouds, resultando numa diminuição do desempenho do algoritmo. Esta escolha justifica-se com o facto de a concorrência esperada entre clientes, no domínio dos sistemas de partilha de dados em cloud reais, ser reduzida.

3 Protocolo de Trinco Composto Tolerante a faltas Bizantinas

A operação *obter()* do objecto de trinco composto é apresentada no Algoritmo 1. Com vista a obter um trinco composto, um cliente chama simultaneamente a operação *obter()* em todos os $3f + 1$ objetos de trinco base (linhas 5–6) e espera, ou por $2f + 1$ repostas bem sucedidas, ou por $f + 1$ respostas de operações que falharam (linha 7). Note que só no primeiro caso o cliente obtém o trinco. Caso contrário o trinco está indisponível ou sob concorrência e, desta forma, o cliente necessita de libertar todos os trincos que potencialmente tenha obtido – tanto os advidos de operações bem sucedidas como os das operações para as quais ainda não foi obtida uma resposta (linhas 11–12). Neste caso, o cliente espera um pouco, repetindo o processo descrito após algum tempo (linha 13). Este algoritmo é repetido até ser obtido sucesso na aquisição do trinco composto ou até um temporizador expirar.

ALGORITMO 1: Trinco composto pelo cliente c.

```
1  function obter(time) begin
2      res ← false;
3      repeat
4          L[0 .. n − 1] ⟵ ⊥;
5          parallel for 0 ≤ i ≤ n − 1 do
6              L[i] ⟵ trincoBase_i.obter(time);
7          wait until i : (|{L[i] = true}| > 2f) ∨ (|{L[i] = false}| > f);
8          if |{i : L[i] = true}| > 2f then
9              res ← true;
10         else
11             for i : (L[i] = ⊥) ∨ (L[i] = true) do
12                 trincoBase_i.libertar();
13             dormir por algum tempo;
14     until res ≠ false;
15     return res;
```

Para libertar o trinco composto basta libertar $n-f$ trincos base. Na renovação de um trinco o processo é similar ao da operação de *obter()* mas é necessário um

acesso adicional às clouds para remover a informação relacionada com o trinco a ser renovado.

3.1 Prova de correção do Protocolo de Trinco Composto

Nesta secção apresentamos a prova de correção do protocolo de trinco composto tolerante a faltas bizantinas. Com vista a provar a exclusão mútua no acesso a recursos é necessário definir com precisão o que significa para um processo deter um trinco.

Definição 1 *Um cliente correcto c detém um trinco no instante t por $T' > 0$ unidades de tempo se obtiver T' como resposta da execução da operação obter(T) num quorum de objetos de trinco base.*

Dada esta definição, vamos proceder à prova das propriedades do Algoritmo 1. Na prova destas propriedades usamos a função C() que mapeia o valor do relógio local de um processo para o valor de um relógio de tempo real (global). Note-se que os processos não têm acesso a esta função, sendo que esta é apenas um instrumento teórico para podermos definir a correção do protocolo.

Teorema 1 (Exclusão Mútua). *Nunca existem dois clientes correctos com um trinco válido em simultâneo.*

Prova. Assuma que isto é falso: existe um instante de tempo real t no qual dois clientes c_1, c_2 detêm o trinco. Vamos provar que esta assumpção leva a uma contradição. Seja t_1 (resp. t_2) o instante do relógio local no qual a operação *obter(T)* retorna o valor T' ao cliente c_1 (resp. c_2), i.e., o momento no qual o cliente obtém o trinco. Seja também t_{start1} (resp. t_{start2}) o instante do relógio local no qual a operação *obter(T)* é chamada por c_1 (resp. c_2). Dado isto, o momento no qual c_1 assume que o trinco expirou é $v_1 = t_{start1} + T'$ (resp. c_2 and $v_2 = t_{start2} + T'$). Com estas definições, a existência de t requer $\mathrm{C}(t_2) \leq \mathrm{C}(v_1)$ ou $\mathrm{C}(t_1) \leq \mathrm{C}(v_2)$.

Tendo em conta que a invocação de *obter(T)* precede o seu retorno, e que este precede a expiração do trinco em todos os objetos de trinco base, nós temos:

$$\mathrm{C}(t_{start1}) < \mathrm{C}(t_1) < \mathrm{C}(v_1) \tag{1}$$

$$\mathrm{C}(t_{start2}) < \mathrm{C}(t_2) < \mathrm{C}(v_2) \tag{2}$$

Assuma o caso esquerdo apresentado na figura 1. Neste caso, visto cada objecto de trinco base garantir a propriedade de *exclusão mútua*, o cliente c_2 só estará apto a obter o trinco quando o trinco detido por c_1 tiver expirado em pelo menos $n - f$ objetos base, i.e, $\mathrm{C}(t_1) + T' < \mathrm{C}(t_2)$. Dada esta condição juntamente com as equações 1 e 2 nós temos:

$$\begin{aligned} \mathrm{C}(t_1) + T' < \mathrm{C}(t_2) \implies \mathrm{C}(t_{start1}) + T' < \mathrm{C}(t_2) \implies \\ (\mathrm{C}(t_{start1} + T') < \mathrm{C}(t_2) \implies \mathrm{C}(v_1) < \mathrm{C}(t_2)) \end{aligned} \tag{3}$$

A equação 3 contradiz o caso da esquerda da figura 1 para a existência de t. A mesma abordagem tem de ser usada, assumindo que c_2 obtém o trinco antes de c_1, para provar que a condição associada a esta situação é também impossível.

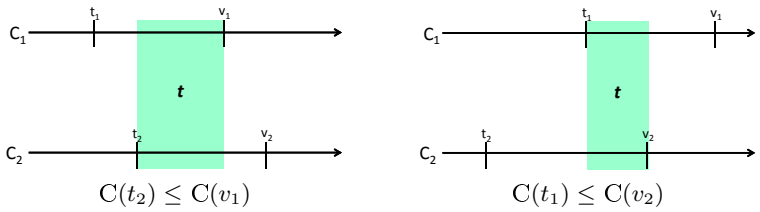

Fig. 1. Ilustração da prova de exclusão mútua.

Teorema 2 (Liberdade de Obstrução). *Um cliente correcto que tente adquirir o trinco sem concorrência terá sucesso.*

Prova. O cliente c começa por tentar obter $n - f$ respostas de sucesso (trincos) dos objetos base (linhas 5 e 6). Ele irá obter o trinco em todos os objetos base correctos pois, (1) nenhum outro cliente detém o trinco, (2) nenhum outro cliente está a tentar obter obter o trinco concorrentemente, e (3) cada objecto de trinco base tem (por definição) de obrigatoriamente garantir a propriedade de "*liberdade de obstrução*". O cliente c irá então obter o trinco do object composto após obter $n - f$ respostas bem sucedidas dos objetos base.

Teorema 3 (*Limitação Temporal*). *Um cliente correcto que obtém o trinco vai possui-lo por um máximo de T unidades de tempo, excepto se este for renovado.*

Prova. Vamos provar este teorema por contradição. Assumamos então que o a definição do mesmo é falsa, i.e., existe um caso no qual um cliente c detém o trinco por $T' > T$ unidades de tempo. Sejam t_1 e t_{start} os instante do relógio local no qual a operação *obter(T)* retorna o valor T' a c_1, ou seja, o momento no qual o cliente obtém o trinco e o momento em que esta operação é chamada pelo mesmo cliente, respectivamente.

Com isto, c_1 detém o trinco por $T' > T$ unidades de tempo e, dado que a invocação da operação *obter(T)* precede o seu seu retorno ($t_{start} < t_1$), temos:

$$T' = T - (t_1 - t_{start}) > T \implies -(t_1 - t_{start}) > 0 \implies \\ (t_1 - t_{start} < 0 \implies t_1 < t_{start}) \tag{4}$$

O resultado da equação 4 contradiz a definição causal de t_{start} e de t_1.

4 Implementação de Trincos Base

A maioria dos serviços fornecidos pelas *clouds* públicas, desde o armazenamento de objetos até às bases de dados atómicas, fornecem mecanismos que permitem a criação de objetos de trinco base. No entanto, todas as implementações de objetos base têm de ser condizentes com algumas especificações de forma a cooperarem entre si como peças integrantes de um trinco composto.

Primeiramente, a operação `obter()` requer a criação bem sucedida de diferentes tipos de entradas de trinco no serviço de cloud. Em segundo lugar, os clientes são responsáveis por colectar as suas entradas de trinco que estejam expiradas

ou inválidas com vista a poupar recursos. Na maioria das implementações, esta limpeza requer pelo menos uma operação extra à cloud nas operações de *obter()* e de *renovar()*. Terceiro, as entradas de trinco têm de ser assinadas antes de serem enviadas para a cloud para garantir que os provedores de cloud não conseguem criar ou corromper trincos. Em quarto lugar, as implementações dos objetos trinco base usam os mecanismos de controlo de acesso dos serviços de cloud para garantir que apenas clientes autorizados podem aceder ao trinco. Desta forma, clientes maliciosos só conseguem prejudicar clientes correctos se estes inadvertidamente lhes tiverem dado acesso ao recurso. Quinto, os clientes não usam o seu relógio local para marcar as entradas de trinco (para marcar o início do seu período de validade). Ao invés disso, eles contam com o serviço de cloud para marcar as entradas de trinco ou usam os valores do relógio da cloud retornados de cada operação efectuada aos serviços. Estas marcas são também usadas para verificar a validade do trinco (ao invés do relógio local).

Nesta secção vamos descrever quatro implementações de objetos de trinco base desenvolvidos em cima de diferentes serviços oferecidos por alguns dos mais populares provedores de cloud. Embora estas implementações utilizem diferentes serviços na cloud, todas elas foram desenhadas seguindo um conjunto similar de técnicas que nos permitem lidar com comportamentos maliciosos e a duração dos trincos.

4.1 Armazenamento de objetos

Os serviços de armazenamento de objetos mantêm objetos de dados de tamanho variável em contentores que são acessíveis através de uma interface de armazenamento de pares chave-valor hierárquica. O algoritmo 2 apresenta a nossa implementação de um objecto de trinco base para este serviço de cloud.

Este algoritmo funciona em três passos. Um cliente começa por listar todos os objetos no contentor (linha 3). Se nenhuma entrada de trinco válida for encontrada ou se for encontrada uma entrada que pertença ao próprio cliente (um trinco que necessita ser renovado), significa que o trinco está livre e o cliente pode tentar obtê-lo (linhas 7-10 e 13), caso contrário o trinco não pode ser obtido e a função retorna *false* (linha 12). Para obter o trinco o cliente chama o procedimento *performLease*() no qual, primeiramente, ele insere uma nova entrada assinada no contentor (linha 16), e de seguida lista os objetos novamente para verificar se nenhuma outra entrada de trinco foi inserida concorrentemente por outro cliente (linha 17). Se nesta segunda listagem for observada outra entrada de trinco válida, o cliente remove a sua entrada de trinco e retorna *false* (linhas 18-20). Caso contrário, e antes de retornar o sucesso da operação, é necessário perceber se esta corresponde à renovação de um trinco. Nesse caso, a entrada de trinco mais antiga é removida (linhas 21 e 22).

Para libertar um trinco, o cliente tem apenas de remover a sua entrada de trinco do serviço de armazenamento.

Este algoritmo funciona em serviços como o Amazon S3 [2], Google Storage [4], Azure Blob Storage [14] e o Rackspace Files [7] visto todos eles garantirem consistência forte no que diz respeito à criação de objetos e fornecerem o valor do relógio do respectivo provedor de cloud em todas as respostas às operações em si efectuadas.

ALGORITMO 2: Trinco no serviço de armazenamento pelo cliente c.

```
1  function servArmazenamentoObter(time)
2  begin
3      L ⟵ cloud.list();
4      cTime ⟵ cloud.getTime();
5      lease_id ←"lease-"+c+"-"+time;
6      index ⟵ 0;
7      foreach lease-c'-T'_i ∈ L do
8          if lastTimeModified(lease-c'-T'_i) + T' > cTime ∧ verify(lease-c'-T'_i, K_{u_c'})) then
9              if c' = c then
                   // é uma operação de renovação
10                 return performLease(lease_id,lease-c'-T'_i);
11             else
12                 return false;
13     return performLease(lease_id, ⊥);
14 procedure performLease(lease_id, oldLease)
15 begin
16     cloud.write(lease_id, sign(lease_id, K_rc));
17     L ⟵ cloud.list();
18     if ∃e ∈ L : e ≡lease-c'-T' : c' ≠ c ∧ lastTimeModified(e)+T'> cTime ∧ verify(e, K_{u_c'}))
       then
19         cloud.delete(lease_id));
20         return false;
       // verificar se é uma renovação
21     if oldLease ≠⊥ ∧oldLease ≠ lease_id then
22         cloud.delete(oldLease);
23     return true;
```

Prova informal. Para provar que este algoritmo cumpre a propriedade de exclusão mútua é necessário perceber que, para que haja dois clientes com um trinco válido em simultâneo, ambos teriam que ver a sua entrada de trinco como sendo a única válida na segunda listagem (linha 17) num determinado período. Claramente isto é impossível pois, visto que estes escrevem a sua entrada de trinco antes da segunda listagem, ambos verem apenas a sua entrada de trinco significaria que as suas visões do serviço eram inconsistentes, o que é impossível dado que estes serviços são fortemente consistentes na criação das entradas de trinco. Na mesma linha, provamos que a propriedade de liberdade de obstrução é garantida pois, se um cliente estiver a executar o algoritmo sem concorrência, claramente a sua entrada de trinco vai ser a única válida devolvida na segunda listagem. O algoritmo garante também a propriedade de limitação temporal pois a validade das entradas de trinco é testada sempre que o algoritmo é executado e por todos os clientes (linha 8), comparando o momento de criação do trinco (*lastTimeModified*()) com o valor actual do relógio da cloud. Caso a entrada de trinco já tenha passado o seu período de validade, significa que o cliente já não detém o trinco.

4.2 Filas Aumentadas

Serviços como a Windows Azure Queue [6] e a Rackspace Queue [5] têm operações de *enqueue* (inserir), *dequeue* (obter e remover) e *list* (listar) fortemente consistentes, fornecendo desta forma uma abstracção de memória partilhada universal capaz de resolver problemas de sincronização [18]. Uma característica importante

fornecida por estes serviços de fila é o facto de ambos permitirem ao cliente especificar a duração máxima na qual os dados (no nosso caso as entradas de trinco) estão válidos no serviço. Após esse período de tempo, a entrada desaparece da fila. O algoritmo 3 apresenta a nossa implementação de objetos de trinco base utilizando estes serviços.

No algoritmo, o cliente começa por listar os objetos presentes na fila para verificar se existem entradas de trinco de outros clientes (linha 4). Se no resultado da listagem não existir nenhuma entrada de trinco de um outro cliente, significa que ele pode tentar obter o trinco (através da chamada do procedimento *performLease*() (linhas 5 e 6), caso contrário o trinco não pode ser obtido e é retornado *false* (linha 7). Neste ponto o cliente insere uma nova entrada de trinco assinada e lista novamente todas as entradas da fila para verificar se a sua entrada é a entrada válida com menor índice (cabeça da fila), o que significa que o cliente detém o trinco (linhas 12-14). Neste momento, o cliente remove todas as entradas da fila com a excepção da sua (linhas 14-16), tanto para remover as suas entradas de trinco desactualizadas (na operação *renovar()*), como para garantir que nenhum outro cliente obtém o trinco inadvertidamente quando a sua entrada expirar e desaparecer da fila. Para libertar um trinco neste objecto base, o cliente apenas necessita remover a sua entrada de trinco da fila.

Prova informal. O algoritmo cumpre a propriedade de exclusão mútua pois, para que hajam dois clientes com o trinco em simultâneo, teriam que existir dois trincos válidos com o menor índice na fila na segunda listagem (linha 11), o que é claramente impossível pois, independentemente da ordem pela qual os pedidos de inserção de entradas de trinco sejam executados pelo serviço, apenas um terá o menor índice. Garante também a propriedade de liberdade de obstrução pois, se um cliente estiver a executar o algoritmo sem concorrência, após a inserção

ALGORITMO 3: Trinco em filas aumentadas pelo cliente c.

```
1  function filasObter(T)
2  begin
3      lease_id ←— "lease-"+c+nonce+"-"+sign(lease_id, K_{r_c});
4      L ←— queue.list();
5      if ∄e ∈ L : e ≡lease-c'-nonce'-s ∧ c' ≠ c ∧ verify(s, K_{u_{c'}}) then
6          return performLease(L, lease_id, T);
7      return false;
8  procedure performLease(L, lease_id, T)
9  begin
10     queue.add(lease_id, T);
11     L_2 ←— queue.list();
12     for 0 ≤ i ≤ size(L_2) do
13         if L_2[i] ≡lease-c'-nonce'-s ∧ verify(s, K_{u_{c'}}) then
14             if L_2[i] = lease_id then
15                 queue.delete(L);
16                 return true;
17             else if c' ≠ c then
18                 queue.delete(lease_id);
19                 return false;
20     return false;
```

da entrada de trinco, esta será a única válida logo a que tem menor índice. A propriedade de limitação temporal é também garantida pois os serviços de fila garantem que a entrada de trinco só fica na fila durante o período pré-definido (que define a validade do mesmo).

4.3 Bases de dados NoSQL

O Amazon DynamoDB [1] é um serviço fortemente consistente que armazena dados como pares que contêm uma chave única associada com um conjunto de valores, e que fornece uma operação de actualização que permite a implementação de objetos base eficientes. No algoritmo 4 é apresentada a implementação de um objecto de trinco base que tira partido deste serviço.

Neste algoritmo, o cliente verifica se já existe alguma entrada de trinco válida na base de dados. Em caso afirmativo, e se esta entrada pertencer a outro cliente a operação retorna *false* (linhas 3–7). Caso contrário, o cliente escreve a nova entrada de trinco assinada usando a operação de actualização condicional (linha 12). Esta operação garante que a entrada é actualizada apenas no caso de nenhum outro cliente ter adicionado uma entrada de trinco no entretanto, retornando no resultado se a actualização teve ou não sucesso.

Para libertar o trinco, basta remover a entrada de trinco da base de dados.

Prova informal. Para um cliente obter um trinco tem de obter sucesso da operação *testAndSetItem*() (linha 12). Este algoritmo garante exclusão mútua pois, visto esta operação ser atómica, se ambos executarem esta operação concorrentemente, independentemente do valor do parâmetro *res*, apenas um irá ter sucesso. Da mesma forma, este garante também liberdade de obstrução pois, visto não haver concorrência, o cliente irá ter $res = \perp$ e a operação *testAndSetItem*() irá ser bem sucedida. Garante ainda a propriedade de limitação temporal pois a validade dos trincos é testada a cada execução do algoritmo por todos os clientes (linha 6), comparando o momento de criação do trinco com o valor actual (usando o relógio da cloud).

ALGORITMO 4: Trinco em bases de dados NoSQL pelo cliente c.

```
1  function bdNoSlqObter(time)
2  begin
3      res ⟵ db.query("key", EQ,"lease");
4      cloudTime ⟵ db.getTime();
5      if res ≠⊥ ∧res.cId ≠ c then
6          if res.expirationTime < cloudTime ∧ verify(res, K_{u_res.cId}) then
7              return false;
8      lease.key ⟵"lease";
9      lease.expirationTime ⟵ cloudTime + time;
10     lease.cId ⟵ c;
11     lease.sign ⟵ sign(lease, K_{r_c});
       // apenas é bem sucedida se o valor no serviço for igual ao valor dado (i.e. res)
12     succeed ⟵ db.testAndSetItem(res, lease);
13     return succeed;
```

4.4 Bases de dados transaccionais

Este tipo de serviços armazena os dados em tabelas e suportam transacções ACID. Um exemplo é o Google Datastore [3], que é uma base de dados fornecida como serviço de cloud. Contrariamente ao Amazon DynamoDB, este não suporta a função de *test-and-set*, no entanto suporta transacções atómicas que permitem, da mesma forma, a implementação eficiente de objetos de trinco base. O algoritmo 5 mostra essa implementação.

A operação de *obter()* é executada numa transacção. O cliente começa por iniciar a transacção e efectuar uma pesquisa na base de dados por uma entrada de trinco (linhas 3 e 4). Se existir uma entrada válida que pertença a outro cliente, a transacção é abortada e a operação retorna *false* (linhas 8-10). Caso contrário, o cliente está apto a tentar obter/renovar o trinco através da chamada do procedimento *performLease*(). Neste ponto, consoante a operação seja uma renovação ou uma obtenção de um trinco, o cliente chama as operações *update* ou *insert* do serviço de cloud, respectivamente, para actualizar/inserir uma entrada de trinco assinada (linhas 18–21). Depois disto, é feito o *commit* da transacção, dependendo o sucesso na obtenção do trinco do sucesso desta operação (linha 22).

A libertação do trinco, à imagem de outros algoritmos apresentados, é feita pela remoção da entrada de trinco da base de dados.

Prova informal. Este algoritmo garante exclusão mútua pois, visto a transacção ser iniciada antes da execução da operação de pesquisa (linha 3), e o *commit* da transacção ser executado após a inserção/actualização da entrada de trinco, estas operações são feitas de forma atómica. Isto garante que, entre a operação de pesquisa e o *commit*, não existe concorrência de operações entre clientes. Assim,

ALGORITMO 5: Trinco em bases de dados transaccionais pelo cliente c.

```
1  function bdTransObter(time)
2  begin
3      trans ⟵ ds.beginTransaction();
4      res ⟵ ds.lookUp("lease", trans);
5      cloudTime ⟵ ds.getTime();
6      if res =⊥ then
7          return performLease((cloudTime + time), false, trans);
8      if res.cId ≠ c ∧ res.expirationTime < cloudTime ∧ verify(res.sign, K_{u_{res.cId}}) then
9          ds.abort(trans);
10         return false;
11     return performLease((cloudTime + time), true, trans);
12 procedure performLease(expTime, isRenew, trans)
13 begin
14     lease.key ⟵"lease";
15     lease.cId ⟵ c;
16     lease.expirationTime ⟵ expTime;
17     lease.sign ⟵ sign(lease, K_{r_c});
18     if !isRenew then
           // apenas é bem sucedida se não existir uma chave igual no serviço
19         ds.insert(lease, trans);
20     else
21         ds.update(lease, trans);
22     return ds.commit(trans);
```

se dois clientes tentarem obter o trinco concorrentemente apenas um deles terá oportunidade de inserir uma entrada no serviço e assim obter o trinco, pois o outro irá obter uma entrada de trinco válida na operação de pesquisa. A liberdade de obstrução é garantida pois, se um cliente estiver a executar o algoritmo sem concorrência, não existirá nenhuma entrada válida na operação de pesquisa logo este irá inserir uma nova, e obter o trinco. O algoritmo garante a limitação temporal pois, tal como em algoritmos anteriores, a validade das entradas de trinco é testada a cada execução do algoritmo (linha 8) utilizando o relógio da cloud.

4.5 Comparação de Objetos de Trinco Base

A tabela 1 resume as propriedades dos diferentes objetos de trinco base implementados em cima de diferentes serviços de cloud.

Esta tabela mostra que a maioria dos objetos base necessitam de três acessos às clouds para implementa a operação *obter()*. A liberdade de obstrução é a propriedade de progresso suportada por todos os algoritmos, no entanto, todos os algoritmos que não são baseados em serviços de armazenamento de objetos satisfazem também a propriedade de *liberdade de impasse* [10]. A propriedade de *liberdade de impasse* garante que se dois ou mais clientes tentarem obter o trinco de forma concorrente, um deles é bem sucedido. Actualmente,o nosso protocolo de trinco composto satisfaz apenas a propriedade de liberdade de obstrução, mesmo que os n objetos de trinco base garantam a propriedade de *liberdade de impasse*.

Há uma diferença de custos significativa em executar os algoritmos de trinco base. No entanto, o nosso trinco composto vai ser significativamente mais barato que correr um serviço de trincos tolerante a faltas em múltiplas VMs nas clouds, especialmente se considerarmos que os trincos são apenas necessários para coordenar escritas e são mantidos por algum tempo. Mais especificamente, cada obtenção de trinco custará cerca de μ\$40 enquanto que ter VMs em quatro provedores diferentes puderá custar \$1200 por mês, mais o custo com o tráfego de dados (que é significante pois as réplicas do serviço de trincos precisam de se sincronizar em cada operação de obtenção de trinco) [12] e o custo com manutenção.

Serviço	Acessos	Custos (μ\$)	Progresso
Amazon S3	3	15	Liberdade Obst.
Google Storage	3	30	Liberdade Obst.
Azure Blob Storage	3	0.108	Liberdade Obst.
RackSpace Files	3	8.640	Liberdade Obst.
Azure Queue	3	0.15	Liberdade Impasse
RackSpace Queue	3	30.144	Liberdade Impasse
Amazon DynamoDB	2	subscrição	Liberdade Impasse
Google Datastore	4	1.2	Liberdade Impasse

Tabela 1. Objetos de trinco base criados em cima de serviços de cloud. A tabela mostra o número de acessos necessários para obter o trinco na ausência de concorrência; os custos monetários (em micro-dólares) dessa operação; a propriedade de progresso satisfeita por cada objecto base, sejam *liberdade de obstrução* ou liberdade de impasse (que é mais forte). *Este serviço é pago popr subscrição mensal.

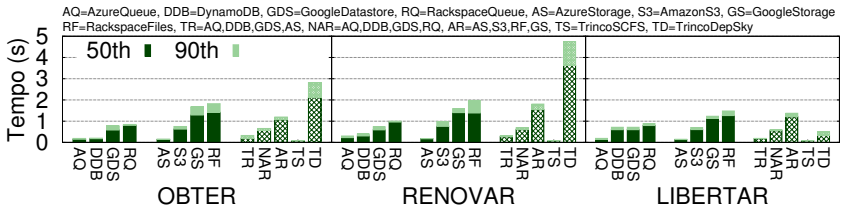

Fig. 2. Latência das operações de obtenção, renovação e libertação do trinco (sem concorrência) para todos os objetos base, para várias configurações do trinco composto, e para dois dois algoritmos de outros trabalhos, nomeadamente o do DepSky e do SCFS.

5 Avaliação

Nesta secção avaliamos o trinco composto e todas as implementações de de objetos base com e sem concorrência. Comparamos ainda dos algoritmos que apresentamos com os algoritmos de exclusão mútua do DepSky e do SCFS. O DepSky foi configurado para utilizar os serviços: Amazon S3 (US), Google Storage(US), Azure Storage (UK) e Rackspace Files (UK). Também o serviço de coordenação do SCFS foi instalado em VMs *standard* no Amazon EC2 (EU), no Microsoft Azure (EU), na Rackspace Servers (EU) e na ElasticHosts (EU).

5.1 Execuções sem concorrência

A Figura 2 apresenta a latência para obtenção, renovação e libertação do trinco na ausência de concorrência para várias configurações do algoritmo de trinco composto, para os objetos de trinco base e para o algoritmo de exclusão mútua do DepSky [11] (TD) e do SCFS [12] (TS). Note-se que no que diz respeito às configurações compostas, na composição definida por TR são aglomerados os 4 objetos de trinco base mais rápidos, mesmo que estejam no mesmo provedor de cloud. A composição NAR reúne os objetos base de serviços que não são de armazenamento, ao passo que AR mostra resultados para algoritmo composto usando só objetos base que usam serviços de armazenamento.

Os objetos de trinco base baseados nos serviços que não são armazenamento necessitam de um tempo entre 200 ms a 1.8 s para obter o trinco. Por sua vez, os objetos de trinco base referentes a serviços de armazenamento obtêm o trinco num tempo compreendido entre 200 ms e 2.8 s. No geral, as latências apresentadas por serviços de armazenamento são piores que as dos restantes serviços. Nós pensamos que isto acontece porque os serviços de armazenamento em cloud são orientados á taxa de transferência, sendo menos eficazes quando lidam com objetos pequenos (entradas de trinco têm aproximadamente 60 bytes).

Os resultados para a obtenção do trinco nas configurações de trinco composto reflectem o desempenho dos seus objetos de trinco base. Mais especificamente, o protocolo de trinco composto espera por um quorum de $2f + 1 = 3$ confirmações dos seus objetos base, o que significa que a latência do objecto composto é similar à latência do terceiro objecto base mais rápido. Um exemplo disto é a configuração NAR cuja latência é similar ao GDS, que é pior que o DDB e AQ, mas melhor

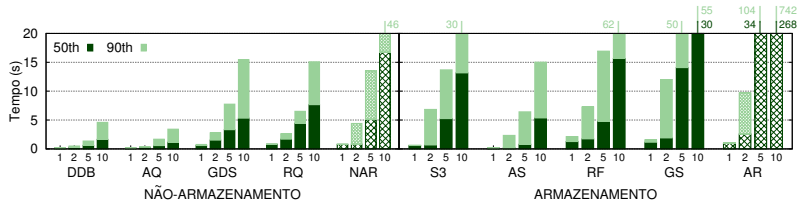

Fig. 3. Latência da obtenção do trinco na presença de concorrência de até 10 clientes.

que o RQ. Para o trinco composto usando só serviços de armazenamento em cloud (AR), observámos uma latência 100% pior que no trinco composto NAR. A configuração TR usa os objetos base mais rápidos, apresentando um latência 100% inferior quando comparada com a NAR. Contudo, esta configuração tem uma limitação importante que se prende o facto de usar dois objetos base na mesma cloud (AQ e AS), o que conduz a uma menor diversidade. Pode verificar-se ainda que o algoritmo do DepSky (TD) tem resultados de aproximadamente dobro da latência da configuração AR (com o o seu comportamento é comparável), e o quadruplo da NAR. Isto acontece porque o algoritmo do DepSky acede ás clouds por fases, enquanto que os restantes (AR, NAR e TR) executam os algoritmos de trinco em paralelo. Por fim, verificámos que o algoritmo de exclusão mútua do SCFS (TS) é o que apresenta melhores resultados. Contudo, se o nosso protocolo oferece uma muito melhor relação custo/desempenho se tivermos em conta todos os custos associados à manutenção dos servidores do SCFS.

Como também se pode ver na figura, o padrão observado na obtenção do trinco verifica-se também nas restantes operações. A única excepção aparece no comportamento do TD quando comparado com as configurações NAR e AR. Isto justifica-se com o facto de todos os objetos base, além de libertarem o trinco, efectuarem também a remoção de entradas de trinco inválidas.

5.2 Execuções na presença de concorrência

Um aspecto importante a ter em conta nos algoritmos de exclusão mútua é a sua capacidade da solução escalar quanto ao número de clientes a tentar obter o trinco concorrentemente. Para avaliar a nossa solução neste sentido, realizámos um teste em que vários clientes (1, 2, 5 e 10) tentam obter o trinco (e em caso de sucesso, o libertam no instante seguinte) e medimos tempo necessário para cada obtenção bem sucedida. No caso dos algoritmos que garantem apenas *liberdade de obstrução* usámos um tempo de *backoff* aleatório de entre 0 e 1 segundos.

A Figura 3 mostra os resultados obtidos os objetos base e para várias configurações de objetos compostos. Novamente, os serviços que não são de armazenamento (à esquerda) fornecem melhores resultados do que os serviços de armazenamento em cloud. Isto acontece porque os serviços do primeiro grupo satisfazem a propriedade *deadlock-freedom*, garantindo que se um conjunto de processos tentar obter um trinco concorrentemente, algum deles irá ter sucesso. Esta propriedade torna-os muito melhores para lidar com concorrência quando comparados com os restantes (ver Tabela 1). O objecto composto (NAR e AR) também garante

apenas *liberdade de obstrução*, tendo assim um aumento linear da latência na obtenção de trincos quando na presença de concorrência. Embora este algoritmo seja rápido com 1 ou 2 clientes, é visivelmente lento com 5 e 10 clientes.

6 Trabalhos Relacionados

Um dos pontos a ter em conta no desenho de serviços de armazenamento é forma como pretende coordenar acessos concorrentes aos seus recursos. Alguns dos trabalhos que propõem este tipo de algoritmos para implementar sistemas seguros são [8,11,15,20]. Em alto nível, o algoritmo de exclusão mútua apresentado neste artigo é muito similar ao apresentado em [20], que por sua vez se assemelha a algoritmos clássicos (desde os anos 80) de exclusão mútua baseada em quoruns. Contudo, este algoritmo requer servidores que executam código para resolver parte do algoritmo [20]. Em contraste, o nosso é construído em cima de algoritmos desenhados especificamente para os serviços de cloud disponíveis nos dias de hoje.

Como dito anteriormente, outro trabalho que motivou o desenvolvimento do nosso algoritmo foi [8]. Este é um protocolo bizantino de consenso baseado em discos partilhados não confiáveis (tais como as clouds de armazenamento). Uma das suas características principais é o facto de poder ser usado para implementar exclusão mútua satisfazendo a propriedade *liberdade de impasse* (que é mais forte que a *liberdade de obstrução*). No entanto, este necessita de um número elevado de acessos às clouds: pelo menos 5 acessos por ronda (podem haver várias rondas se existir concorrência). Por outro lado, o nosso protocolo composto requer apenas entre 2 a 4 acessos por objecto base para obter o trinco (ver tabela 1).

Segundo sabemos, existem apenas dois algoritmos de trinco centrados em dados tolerantes a faltas [11,15], sendo que ambos são baseados no uso de clouds de armazenamento ou de discos. O algoritmo de trinco apresentado por [15] tem duas diferenças importantes quando comparado com o algoritmo de trinco composto apresentado neste artigo. Em primeiro lugar, não garante um trinco sempre seguro no sentido em que o algoritmo admite a possibilidade de existência de mais do que processo com trinco válido. Isto acontece porque a única garantia fornecida por [15] é que, algures no tempo, apenas um processo irá ter o trinco. Este tipo de garantia é adequada para implementar eleição de líder, no entanto é inadequado para garantir exclusão mútua. Em segundo, ele apenas tolera falhas por paragem (não falhas bizantinas). Por outro lado, o algoritmo de exclusão mútua do DepSky [11] tolera faltas bizantinas, sendo o mais similar ao nosso do que diz respeito às garantias fornecidas. Todavia, o nosso algoritmo tem um desempenho quatro vezes superior e não necessita que os clientes tenham os relógios sincronizados.

7 Conclusões

Existem vários serviços que necessitam coordenar o acesso concorrente de clientes a recursos partilhados (e.g.,ficheiros). No entanto as soluções existentes são ou ineficientes ou têm elevados custos associados. Este problema levou-nos a desenhar um novo protocolo de exclusão mútua tolerante a faltas bizantinas que não necessita de nenhum servidor específico a correr nas clouds. A avaliação mostra que

este protocolo tem um desempenho, por um lado muito superior ao dos protocolos existentes com comportamento semelhante, por outro comparável a soluções que mantém servidores dedicados a correr nas clouds.

Agradecimentos. Queremos agradecer ao Vinícius Cogo pela sua ajuda em vários aspectos do artigo. Este trabalho foi suportado pela Comissão Europeia através dos projectos BiobankCloud (FP7/ICT-317871) e SuperCloud (H2020/ICT-643964), e pela Fundação para a Ciência e a Tecnologia (FCT) através de seu programa multianual (LaSIGE).

Referências

1. Amazon DynamoDB. `http://aws.amazon.com/dynamodb/`.
2. Amazon S3. `http://aws.amazon.com/s3/`.
3. Google Cloud Datastore. `https://cloud.google.com/datastore/`.
4. Google storage. `https://developers.google.com/storage/`.
5. Message queuing service with simple API – Rackspace cloud queues. `http://www.rackspace.com/cloud/queues/`.
6. Microsoft Azure Queue. `http://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-how-to-use-queues/`.
7. Rackspace cloud files. `http://www.rackspace.co.uk/cloud/files`.
8. I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.
9. H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *Procs. of the ACM SoCC*, 2010.
10. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, 2nd edition, 2004.
11. A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DepSky: Dependable and secure storage in cloud-of-clouds. *ACM Transactions on Storage*, 9(4), 2013.
12. A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: a shared cloud-backed file system. In *Procs. of the USENIX ATC*, 2014.
13. A. Bessani et al. The overbank cloud architecture, protocols and middleware, Nov. 2014. Deliverable D4.2 of BiobankCloud project.
14. B. Calder et al. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Procs. of the ACM SOSP*, 2011.
15. G. Chockler and D. Malkhi. Light-weight leases for storage-centric coordination. *International Journal of Parallel Programming*, 34(2), Apr. 2006.
16. J. Y. Chung, C. Joe-Wong, S. Ha, J. W.-K. Hong, and M. Chiang. Cyrus: Towards client-defined cloud storage. In *Procs. of the ACM EuroSys*, 2015.
17. C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the ACM SOSP*, 1989.
18. M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.
19. P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale services. In *Procs. of the USENIX ATC*, 2010.
20. D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Procs. of the IEEE SRDS*, 1998.
21. T. Oliveira, R. Mendes, and A. Bessani. Sharing files using cloud storage services. In *DIHC, co-located with Euro-Par*, 2014.