Replicação Não Uniforme com Consistência Eventual

Gonçalo Cabrita e Nuno Preguiça

NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa

Resumo A replicação é uma técnica chave no desenho de sistemas distribuídos confiáveis e eficientes. À medida que a informação cresce, torna-se difícil ou até impossível guardar todos os dados em todas as réplicas. Uma solução comum para este problema passa por utilizar técnicas de replicação parcial, onde cada réplica mantém apenas uma parte dos dados. Por consequência, cada réplica consegue apenas responder a um subconjunto das *queries* suportadas. Neste artigo, introduzimos o conceito de replicação não uniforme, onde cada réplica guarda apenas parte dos dados mas consegue responder a todas as *queries* suportadas. Aplicamos este conceito à consistência eventual e aos tipos de dados livres de conflitos. A nossa avaliação mostra que a replicação não uniforme é mais eficiente que a replicação tradicional, conseguindo reduzir o espaço de armazenamento utilizado, a quantidade de dados transmitidos, e aumentando a escalabilidade do sistema.

Palavras-chave: Replicação não uniforme; Replicação parcial; Tipos de dados replicados; Consistência eventual

1 Introdução

Um grande número de aplicações corre sobre infraestruturas de nuvem compostas por múltiplos centros de dados distribuídos geograficamente pelo mundo. Estas aplicações utilizam bases de dados replicadas para obterem latências baixas, alta disponibilidade e tolerância a falhas [6, 7, 9, 11]. Algumas bases de dados fornecem consistência forte [7, 16], dando a ilusão de que apenas uma única réplica existe. Estes sistemas requerem coordenação entre as réplicas para a execução de operações, o que tem impacto na latência e disponibilidade. Outras bases de dados [9, 11] optam por fornecer alta disponibilidade e latência baixa, permitindo a execução local de operações num único centro de dados. Desta forma, conseguem responder aos clientes sem terem de coordenar as operações entre as réplicas, levando a uma latência baixa. Esta aproximação permite a ocorrência de modificações concorrentes, tendo as aplicações de lidar com um modelo de consistência fraca em que é necessário reconciliar réplicas divergentes.

Recentemente, os tipos de dados livres de conflitos [20] (CRDTs) têm sido adotados em várias bases de dados chave-valor distribuídas [5, 21] como um mecanismo de reconciliação seguro. Os CRDTs permitem simplificar a adoção

de modelos de consistência fraca, removendo a necessidade de lidar diretamente com a resolução de conflitos.

Com o aumento da informação mantida nas bases de dados, é impossível ou indesejável manter todos os dados em todas as réplicas. Para além de aplicar técnicas de *sharding* entre múltiplas máquinas em cada centro de dados, é útil manter apenas parte dos dados em cada centro de dados. Em sistemas que adotam um modelo de replicação parcial [8, 19], como cada réplica mantém apenas parte dos dados, só é possível processar um subconjunto de *queries*.

Neste artigo exploramos um modelo de replicação parcial alternativo, o modelo de replicação não uniforme, onde cada réplica mantém apenas parte dos dados mas consegue processar todas as *queries* localmente. A ideia principal é que para alguns objetos replicados, não são necessários todos os dados para responder a uma *query*. Por exemplo, um objeto que mantém o conjunto dos K maiores elementos apenas precisa de manter os K maiores elementos em cada réplica. No entanto, os elementos menores podem ser necessários se uma operação de remoção estiver disponível. Aplicámos este modelo de replicação não uniforme aos CRDTs, formalizando-o para um modelo onde as réplicas sincronizam propagando operações. Adicionalmente, apresentamos um tipo de dados que utiliza este modelo. As nossas avaliações mostram que o modelo de replicação não uniforme leva a uma redução do espaço de armazenamento utilizado e da quantidade de dados transmitidos, e a um aumento da escalabilidade do sistema.

O resto deste artigo está organizado da seguinte forma. A secção 2 discute o trabalho relacionado. A secção 3 descreve o modelo de replicação não uniforme. A secção 4 aplica o modelo a um sistema de consistência eventual. A secção 5 introduz um tipo de dados que segue o modelo. A secção 6 apresenta uma avaliação da solução proposta.

2 Trabalho relacionado

Vários protocolos de replicação foram propostos na última década [2, 10, 14–16, 18, 22]. Em protocolos de replicação total, cada réplica replica a totalidade dos dados. Estes protocolos têm o benefício de que todas as réplicas conseguem responder a todas as *queries* mas requerem que todas as réplicas (ou um quórum) processem todas as operações de escrita. Adicionalmente, cada réplica tem de armazenar todos os dados, o que pode não ser possível ou práticos. A replicação parcial [3, 8, 19] lida com as limitações da replicação total mantendo em cada réplica apenas parte dos dados. Esta aproximação aumenta a escalabilidade do sistema, no entanto como cada réplica mantém apenas parte dos dados só consegue processar um subconjunto de *queries*.

CRDTs [20] são tipos de dados que podem ser replicados, garantindo que todas as réplicas convergem para o mesmo valor após todas as modificações terem sido propagadas para todas as réplicas. A propagação das modificações pode ser feita através do envio das operações executadas (CRDTs baseados em operações) ou através do envio do estado da réplica (CRDTs baseados em estado).

Os CRDTs baseados em deltas [1] são uma evolução dos CRDTs baseados em estado, nos quais durante a sincronização se envia apenas a diferença entre o estado antigo e o mais recente, reduzindo o custo de disseminação das modificações. Os CRDTs computacionais [17] são um extensão dos CRDTs baseados em estado em que o estado de um objeto é o resultado de uma computação sobre as operações executadas – e.g. a média, os K maiores elementos inseridos. Tal como no modelo que propomos neste artigo, as réplicas destes CRDTs não precisam de ter estados equivalentes. O nosso trabalho evolui as ideias inicialmente propostas nos CRDTs computacionais em diversos aspetos, incluindo a definição de um modelo de replicação não uniforme e a sua aplicação a um modelo de consistência eventual onde as réplicas sincronizam por troca de operações.

3 Replicação não uniforme

Consideramos um sistema distribuído assíncrono composto por n nós. Sem perda de generalidade, assumimos que o sistema replica um único objeto. O objeto suporta uma interface composta por um conjunto de operações de leitura, $\mathcal{Q}$, e um conjunto de operações de escrita, $\mathcal{U}$. O estado que resulta da execução de uma operação o num estado s representa-se por $s \bullet o$. Para uma operação de leitura, $q \in \mathcal{Q}$, $s \bullet q = s$. O resultado da aplicação de uma operação $o \in \mathcal{Q} \cup \mathcal{U}$ num estado $s \in \mathcal{S}$ representa-se por $o(s)$.

Definimos o estado do sistema replicado como um tuplo $(s_1, s_2, \ldots, s_n)$, onde s_i refere-se ao estado da réplica i. O estado das réplicas é sincronizado por um protocolo de replicação que envia mensagens entre os nós do sistema e atualiza o estado das réplicas. Não consideramos nenhum protocolo específico, visto que o nosso modelo pode ser aplicado a diferentes protocolos.

Dizemos que o sistema encontra-se num estado quiescente se, para um dado conjunto de operações, o protocolo de replicação tiver propagado todas as mensagens necessárias para sincronizar todas as réplicas, i.e., mensagens posteriormente enviadas pelo protocolo de replicação não modificarão o estado das réplicas. De uma forma geral, os protocolos de replicação tentam garantir uma propriedade de convergência, onde o estado de quaisquer duas réplicas é equivalente num estado quiescente.

Definição 1 (Estado equivalente). *Dois estados s_i e s_j, são equivalentes, $s_i \equiv s_j$, sse o resultado da execução de uma operação $o_n \in \mathcal{Q} \cup \mathcal{U}$ após a execução de uma qualquer sequência de operações $o_1, \ldots, o_{n-1}$ com $o_1, \ldots, o_n \in \mathcal{Q} \cup \mathcal{U}$ em ambos os estados é igual, i.e., $o_n(s_i \bullet o_1 \bullet \ldots \bullet o_{n-1}) = o_n(s_j \bullet o_1 \bullet \ldots \bullet o_{n-1})$*

Esta propriedade é garantida pela maioria dos protocolos de replicação, independentemente de fornecerem um modelo de consistência forte ou fraca [13, 14, 22]. Esta propriedade não requer que o estado interno das réplicas seja igual, mas apenas que as réplicas retornem os mesmos resultados para a execução de uma qualquer sequência de operações. Neste artigo, propomos relaxar esta propriedade requerendo apenas que a execução de operações de leitura retorne o mesmo valor. Denominamos esta propriedade de *observavelmente equivalente* e definimo-la formalmente da seguinte forma.

Definição 2 (Estados observavelmente equivalentes). *Dois estados, s_i e s_j, são observavelmente equivalentes, $s_i \overset{o}{\equiv} s_j$, sse o resultado da execução de uma operação $o \in \mathcal{Q}$ em ambos os estados for igual, i.e., $o(s_i) = o(s_j)$.*

De seguida definimos um sistema não uniforme como um sistema que garante apenas que as réplicas convergem para um estado observável equivalente.

Definição 3 (Sistema replicado não uniforme). *Dizemos que um sistema replicado é não uniforme se o protocolo de replicação garante que num estado quiescente, o estado de quaisquer duas réplicas é observavelmente equivalente.*

4 Consistência eventual não uniforme

4.1 Modelo do sistema

Consideramos um sistema distribuído assíncrono composto por n nós, onde os nós podem exibir falhas *fail-stop* mas não falhas bizantinas. Assumimos que é utilizada uma ligação fiável e que está disponível um sistema de comunicação com uma primitiva de comunicação, $mcast(m)$, que pode ser usada por um processo para enviar uma mensagem para todos os outros processos do sistema.

Um objeto é definido como um tuplo $(\mathcal{S}, s^0, \mathcal{Q}, \mathcal{U}_p, \mathcal{U}_e)$, onde $\mathcal{S}$ é o conjunto de estados validos do objeto, $s^0 \in \mathcal{S}$ é o estado inicial do objeto, $\mathcal{Q}$ é o conjunto de operações de leitura, $\mathcal{U}_p$ é o conjunto de operações *prepare-update* e $\mathcal{U}_e$ é o conjunto de operações *effect-update*. Uma operação de leitura executa apenas na réplica onde a operação é invocada, a origem, e não tem efeitos colaterais, i.e., não altera o estado do objeto. Quando uma aplicação pretende atualizar o estado de um objeto, deve invocar uma operação *prepare-update*, $u_p \in \mathcal{U}_p$. Uma operação u_p executa apenas na origem, não tem efeitos colaterais e gera uma operação *effect-update*, $u_e \in \mathcal{U}_e$. Na origem, u_e executa imediatamente após u_p.

Como apenas as operações *effect-update* podem alterar o estado do objeto, para raciocinar sobre a evolução das réplicas restringimos a análise a estas operações. A execução de uma operação u_p gera uma instância de uma operação *effect-update*. Por simplicidade, chamamos a instâncias de operações simplesmente operações. Sendo O_i o conjunto de operações geradas no nó i, o conjunto de operações geradas numa execução, é $O = O_1 \cup \ldots \cup O_n$.

4.2 Consistência eventual não uniforme

Para uma qualquer execução, sendo O o conjunto de operações da execução, um sistema replicado fornece *consistência eventual* sse num estado quiescente: (i) todas as réplicas executaram todas as operações de O; e (ii) os estados de qualquer par de réplicas são equivalentes.

Uma condição suficiente para garantir a primeira propriedade é propagar todas as operações geradas utilizando difusão fiável e executar qualquer operação recebida. Uma condição suficiente para garantir a segunda propriedade passa

por apenas permitir operações comutativas. Assim, se todas as operações comutarem, a execução de uma qualquer serialização de O no estado inicial do objeto resulta num estado equivalente. A partir de agora, assumimos que todas as operações comutam. Como todas as serializações de O são equivalentes, denotamos a execução de uma serialização de O num estado s como $s \bullet O$.

Para uma dada execução, sendo O o conjunto de operações da execução, um sistema replicado fornece *consistência eventual não uniforme* sse num estado quiescente: (i) o estado de qualquer réplica é observavelmente equivalente ao estado obtido pela execução de uma serialização de O; e (ii) os estados de qualquer par de réplicas são observavelmente equivalentes. Para um dado conjunto de operações numa execução O, dizemos que $O_{core} \subseteq O$ é o conjunto de operações essenciais de O sse $s^0 \bullet O \overset{\circ}{\equiv} s^0 \bullet O_{core}$. Definimos o conjunto de operações irrelevantes para o estado final das réplicas da seguinte forma: $O_{masked} \subseteq O$ é o conjunto de operações irrelevantes de O sse $s^0 \bullet O \overset{\circ}{\equiv} s^0 \bullet (O \setminus O_{masked})$.

Teorema 1 (Condições suficientes para NuEC). *Um sistema replicado fornece consistência eventual não uniforme (NuEC) se, para um dado conjunto de operações O, as seguintes condições forem verdadeiras: (i) cada réplica executa um conjunto de operações essenciais de O; e (ii) todas as operações comutam.*

Demonstração. Tendo a definição de operações essenciais de O, e sabendo que todas as operações comutam, podemos dizer que se qualquer réplica executa um conjunto de operações essenciais, então o estado final de todas as réplicas é igual e equivalente ao estado obtido pela execução de uma qualquer serialização de O.

4.3 Protocolo para consistência eventual não uniforme

Agora utilizamos as condições anteriormente definidas para criar um protocolo de replicação que tenta minimizar as operações propagadas para outras réplicas. A ideia principal é evitar propagar operações que façam parte do conjunto de operações irrelevantes. O desafio é fazê-lo utilizando apenas informação local, que inclui apenas um subconjunto das operações executadas.

O algoritmo 1 apresenta o pseudocódigo de um algoritmo utilizado para fornecer *consistência eventual não uniforme*. O algoritmo não garante a durabilidade das operações irrelevantes, sendo esta questão discutida na Secção 4.4.

Para além do estado do objeto, o algoritmo mantém também três conjuntos de operações: (i) $log_{coreLocal}$, o conjunto de operações essenciais geradas localmente na réplica que ainda não foram propagadas para outras réplicas; (ii) log_{local}, o conjunto de operações geradas localmente na réplica que ainda não foram propagadas para outras réplicas; e (iii) log_{recv}, o conjunto de operações que foram propagadas para todas as réplicas, incluindo as operações geradas localmente.

Quando uma operação é gerada, a função *execOp* é invocada. Esta função adiciona a nova operação ao conjunto de operações locais e atualiza o estado do objeto local. Se a nova operação tem um impacto no estado observável do objeto, então esta é também adicionada ao conjunto de operações locais essenciais.

A função *sync* é utilizada para propagar operações locais para réplicas remotas. A função começa por calcular que operações novas devem ser propagadas,

Algoritmo 1 Algoritmo de replicação para consistência eventual não uniforme

```
1: S : state: initial s^0                                        ▷ Estado do objeto
2: log_recv : set of operations: initial {}
3: log_local : set of operations: initial {}                     ▷ Operações locais não propagadas
4: log_coreLocal : set of operations: initial {}                 ▷ Operações essenciais locais não propagadas
5:
6: execOp(op): void                                              ▷ Nova operação gerada localmente
7:   if hasImpact(op, S) then
8:     log_coreLocal = log_coreLocal ∪ {op}
9:   log_local = log_local ∪ {op}
10:  S = S • op
11:
12: opsToPropagate(): set of operations                          ▷ Operações a serem propagadas
13:   ops = maskedForever(log_local ∪ log_coreLocal, S, log_recv)
14:   log_local = log_local \ ops
15:   log_coreLocal = log_coreLocal \ ops
16:   opsImpact = log_coreLocal ∪ hasObservableImpact(log_local, S, log_recv)
17:   opsPotImpact = mayHaveObservableImpact(log_local, S, log_recv)
18:   return opsImpact ∪ opsPotImpact
19:
20: sync(): void                                                 ▷ Propaga operações locais para réplicas remotas
21:   ops = opsToPropagate()
22:   mcast(ops)
23:   log_coreLocal = {}
24:   log_local = log_local \ ops
25:   log_recv = log_recv ∪ ops
26:
27: on receive(ops): void                                        ▷ Processa operações remotas
28:   S = S • ops
29:   log_recv = log_recv ∪ ops
```

realiza a difusão dessas operações, e finalmente atualiza o conjunto local de operações. Quando uma réplica recebe um conjunto de operações (linha 27), esta atualiza o estado local e o conjunto de operações propagadas para todas as réplicas.

A função *opsToPropagate* aborda o desafio de decidir que operações necessitam de ser propagadas para outras réplicas. Para este fim, dividimos as operações em quatro grupos: (i) o conjunto de operações que serão sempre irrelevantes; (ii) o conjunto de operações essenciais (*opsImpact*); (iii) o conjunto de operações que podem ter impacto no estado observável dependendo das operações de outras réplicas que não foram propagadas (*opsPotImpact*); e (iv) as restantes operações que podem ter impacto no estado observável do objeto dependendo das operações que poderão ser executadas e que sejam propagadas para todas as réplicas.

Para demonstrar que o algoritmo pode ser usado para fornecer consistência eventual não uniforme, temos de provar a seguinte propriedade.

Teorema 2. *O algoritmo 1 garante que num estado quiescente todas as réplicas receberam todas as operações essenciais.*

Demonstração. Para demonstrar esta propriedade, temos de provar que não existe nenhuma operação não propagada que seja necessária para um conjunto de operações essenciais. As operações do primeiro grupo são identificadas como sendo irrelevantes para sempre independentemente das operações que executem no futuro. Assim, qualquer conjunto de operações essenciais não necessitará de incluir estas operações. O quarto grupo inclui operações que não influenciam o estado observável considerando todas as operações executadas – se pudessem ter

impacto, estariam no terceiro grupo. Sendo assim, esta operações não precisam de estar em nenhum conjunto essencial. Todas as outras operações são propagadas para todas as réplicas. Assim, num estado quiescente, cada réplica recebeu todas as operações que podem ter impacto no estado observável.

4.4 Tolerância a falhas

A replicação não uniforme tem como objetivos reduzir os custo de disseminação e o tamanho das réplicas, evitando propagar operações que não influenciem o estado observável do objeto. Esta aproximação levanta questões sobre a durabilidade das operações que não são imediatamente propagadas para todas as réplicas (por serem temporariamente irrelevantes).

Uma solução possível passa simplesmente por propagar todas as operações para pelo menos $f+1$ réplicas, tolerando assim até f falhas. Deste modo, temos a garantia que uma operação não se perde mesmo tendo f falhas. No entanto, seria necessário adaptar o algoritmo proposto para que no caso em que a réplica recebesse uma operação para propósitos de durabilidade, esta teria de propagar a operação para outras réplicas se a réplica de origem falhasse.

5 CRDTs não uniformes

Nesta secção, mostramos como aplicar o conceito de replicação não uniforme para criar CRDTs baseados em operações [20]. O tipo de dados apresentado é inspirado nos CRDTs computacionais [17], que também permitem que réplicas divirjam num estado quiescente.

5.1 Top-K com remoções

Nesta secção apresentamos um CRDT não uniforme que representa um *top-K*. O tipo de dados permite o acesso aos K maiores elementos adicionados e pode ser utilizado para manter uma tabela de classificação de um jogo online. O algoritmo proposto pode ser adaptado para definir um CRDT que filtre os elementos utilizando uma qualquer função determinística, substituindo a função *topK* utilizada por outro filtro.

A semântica das operações definidas no CRDT é a seguinte: a operação *add(id,val)* adiciona um novo par ao objeto. A operação *rmv(id)* remove todos os pares associados ao identificador que foram adicionados por uma operação que ocorreu antes da remoção. Esta semântica leva a uma política *add-wins*, onde uma remoção não tem impacto em operações de adição concorrentes. A operação *get()* devolve os K maiores pares (de acordo com a ordem total definida para os pares de elementos, usada pela função *topK*).

O algoritmo 2 apresenta um tipo de dados que implementa esta semântica. A operação *prepare-update add* gera um *effect-update* que inclui um parâmetro adicional contendo uma estampilha temporal $\langle replicaid, val\rangle$, onde val é um inteiro gerado de forma monotonicamente crescente. A operação *prepare-update rmv*

Algoritmo 2 Top-K com remoções

```
1: elems : set of ⟨id, score, ts⟩ : initial {}
2: removes : map id ↦ vectorClock: initial []
3: vc : vectorClock: initial []
4:
5: GET() : set
6:   return {⟨id, score⟩ : ⟨id, score, ts⟩ ∈ topK(elems)}
7:
8: prepare ADD(id, score)
9:   generate add(id, score, ⟨getReplicaId(), ++ vc[getReplicaId()]⟩)
10:
11: effect ADD(id, score, ts)
12:   if removes[id][ts.siteId] < ts.val then
13:     elems = elems ∪ {⟨id, score, ts⟩}
14:     vc[ts.siteId] = max(vc[ts.siteId], ts.val)
15:
16: prepare RMV(id)
17:   generate rmv(id,vc)
18:
19: effect RMV(id, vc_rmv)
20:   removes[id] = pointwiseMax(removes[id], vc_rmv)
21:   elems = elems \ {⟨id_0, score, ts⟩ ∈ elem : id = id_0 ∧ ts.val < vc_rmv[ts.siteId]}
22:
23: HASIMPACT(op, S): boolean
24:   R = S • op
25:   return topK(S) ≠ topK(R)
26:
27: MASKEDFOREVER(log_local, S, log_recv): set of operations
28:   adds = {add(id_1, score_1, ts_1) ∈ log_local :
29:     (∃add(id_2, score_2, ts_2) ∈ log_local : id_1 = id_2 ∧ score_1 < score_2 ∧ ts_1.val < ts_2.val)∨
30:       (∃rmv(id_3, vc_rmv) ∈ (log_recv ∪ log_local) : id_1 = id_3 ∧ ts_1.val < vc_rmv[ts_1.siteId]}
31:   rmvs = {rmv(id_1, vc_1) ∈ log_local :
32:     ∃rmv(id_2, vc_2) ∈ (log_local ∪ log_recv) : id_1 = id_2 ∧ vc_1 < vc_2}
33:   return adds ∪ rmvs
34:
35: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
36:   return {}                                  ▷ Este caso não afeta este tipo de dados
37:
38: HASOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
39:   adds = {add(id_1, score_1, ts_1) ∈ log_local : ⟨id_1, score_1, ts_1⟩ ∈ topK(S.elems)}
40:   rmvs = {rmv(id_1, vc_1) ∈ log_local :
41:     ∃⟨id_2, score_2, ts_2⟩ ∈ topK(S.elems) : id_1 = id_2 ∧ ts_2.val < vc_1[ts_2.siteId]}
42:   return adds ∪ rmvs
```

gera um *effect-update* que inclui um parâmetro adicional contendo um relógio vetorial que sumariza as operações de adição que ocorreram antes da operação de remoção. O objeto mantém um relógio vetorial que é atualizado sempre que uma nova operação de adição é gerada ou executada localmente. Adicionalmente, este relógio vetorial deve ser atualizado sempre que uma réplica recebe uma mensagem de uma réplica remota (para sumarizar também as operações de adição conhecidas pela origem que não foram propagadas para esta réplica).

Para além deste relógio vetorial, vc, cada réplica do objeto mantém: (i) um conjunto, $elems$, contendo os elementos adicionados pelas operações de adição conhecidas localmente (e que ainda não foram removidos); e (ii) um mapa, $removes$, que para cada elemento id tem um relógio vetorial que sumariza as operações de adição que aconteceram antes de todas as operação de remoção de id (por simplicidade, assumimos que uma chave que não faça parte do mapa tem associado um relógio vetorial contendo apenas zeros para cada réplica).

A execução de uma operação de adição consiste em adicionar o elemento ao conjunto, $elems$, se a operação não aconteceu antes de uma remoção an-

teriormente recebida sobre o mesmo elemento – isto pode acontecer visto que as operações podem não ser propagadas por ordem causal [12]. A execução de uma operação de remoção consiste em atualizar o mapa de remoções, *removes*, e apagar do conjunto, *elems*, a informação relativa às adições do elemento que ocorreram antes da remoção. Para averiguar se uma adição ocorreu antes de uma remoção, verificamos se a estampilha temporal associada à adição está refletida no relógio vetorial da remoção (linhas 12 e 21). Deste modo garantimos a semântica do CRDT, assumindo que as funções utilizadas pelo protocolo são corretas.

Agora analisamos o código dessas funções. A função hasImpact verifica se o estado observável é alterado após a execução da nova operação.

A função maskedForever calcula: as adições locais que se tornam irrelevantes tendo em conta outras operações de adição (com o mesmo elemento, mas com um valor mais baixo) e remoção (com o mesmo elemento que ocorreram antes da adição); as remoções que se tornaram irrelevantes tendo em conta outras remoções (com o mesmo elemento, mas com um relógio vetorial menor).

A função mayHaveObservableImpact devolve um conjunto vazio, visto que para ter impacto em qualquer estado observável uma operação teria também de ter impacto no estado local observável por si só.

A função hasObservableImpact calcula: as adições locais que não foram propagadas para outras réplicas e que fazem parte do top-K na réplica local; as remoções locais que removeram um elemento no top-K.

6 Avaliação

Nesta secção comparamos o Top-K proposto neste artigo (NuCRDT) com uma solução que usa replicação total, mantendo o conjunto de elementos em todas as réplicas. Para tal, usamos um CRDT *OR-Set*, que modela um conjunto com suporte para adições e remoções, com uma politica *add-wins* à semelhança do Top-K. A nossa avaliação foi efetuada usando a base de dados chave-valor AntidoteDB [21] e pretende avaliar se o modelo não uniforme permite: (i) reduzir o tamanho das réplicas; (ii) reduzir a quantidade de dados transmitidos; e (iii) aumentar a escalabilidade do sistema.

As experiências foram executadas na plataforma AWS EC2, utilizando instâncias de máquinas do tipo *m3.xlarge* para todos os nós. Todas as experiências usaram 5 nós para o AntidoteDB, cada um a executar nas seguintes regiões: *eu-west*, *eu-central*, *us-east*, *us-west*, e *ap-northeast*. Nas execuções, o tipo de dados foi configurado da seguinte forma: K é 100, os identificadores dos jogadores são selecionados a partir de uma distribuição uniforme tendo um domínio de 10.000, e os valores das pontuações foram selecionados entre 1 e 250.000 usando uma distribuição uniforme. O sistema foi configurado para suportar entre zero a duas falhas, com a réplica de origem a propagar as operações irrelevantes para pelo menos f réplicas.

Como esperamos que a remoção seja uma operação pouco utilizada (usada apenas quando um jogador é removido do jogo), a carga de trabalho é dividida da seguinte forma: 95% para operações de adição e 5% para operações de remoção.

6.1 Tamanho das réplicas e quantidade de dados transmitidos

Para medir o tamanho das réplicas e a quantidade de dados transmitidos entre réplicas utilizamos um único cliente (a correr numa instância EC2 em separado) para executar operações nos 5 nós de AntidoteDB diferentes.

Nesta avaliação o cliente executa uma sequência de operações gerada aleatoriamente sobre os dois tipos de dados a serem comparados. Os valores são obtidos a cada 5.000 operações, obtendo o tamanho total das mensagens enviadas entre cada centro de dados e o tamanho médio do objeto em cada réplica. Os resultados representam o resultado médio de três execuções independentes. A figura 1 mostra os resultados obtidos.

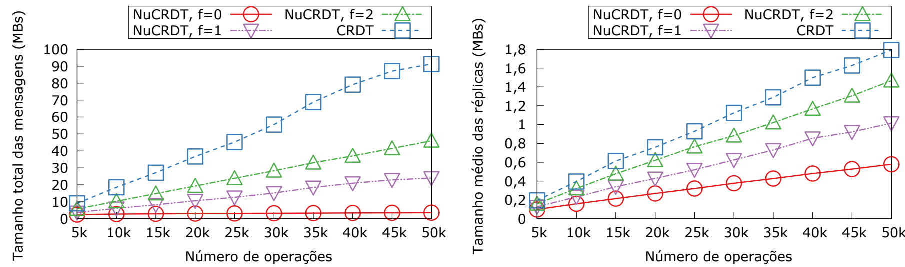

Figura 1. Tamanho total de mensagens e tamanho médio das réplicas do Top-K

O nosso tipo de dados consegue reduzir significativamente o tamanho total das mensagens enviadas entre réplicas (até 96% para $f = 0$) e o tamanho médio das réplicas (até 67% para $f = 0$) quando comparado com o CRDT que usa replicação total. Isto verifica-se principalmente por o CRDT ter de propagar todas operações para todas as réplicas enquanto o NuCRDT propaga apenas as operações essenciais para todas as réplicas e as restantes para um subconjunto de réplicas por questões de durabilidade.

6.2 Escalabilidade

Para medir a escalabilidade dos tipos de dados utilizámos cinco instâncias EC2 extra, cada uma a correr a ferramenta de avaliação Basho Bench [4]. Cada nó Basho Bench executa um número variável de clientes que contactam o nó AntidoteDB a executar na mesma região. Cada execução dura 3 minutos e os resultados apresentados são as médias de três execuções independentes.

A figura 2 apresenta os resultados obtidos, que mostram que ambos os tipos de dados se comportam de forma semelhante com taxas de transferência reduzidas. No entanto, quando se aumenta o número de cliente, a solução com o NuCRDT apresenta melhor escalabilidade, com a solução baseada em CRDTs a ter uma queda drástica da taxa de transferência e um aumento da latência a partir das 4.000 operações por segundo.

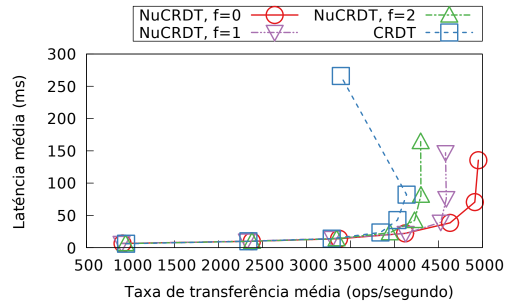

Figura 2. Escalabilidade do Top-K

7 Conclusões

Neste artigo apresentámos o modelo de replicação não uniforme, um modelo alternativo de replicação que combina as vantagens da replicação total, permitindo a qualquer réplica responder a todas as operações de leitura, e da replicação parcial, permitindo que cada réplica mantenha apenas parte dos dados.

Mostrámos como aplicar este modelo à consistência eventual, e propusemos um protocolo de sincronização por troca de operações genérico que fornece replicação não uniforme. Apresentámos também um tipo de dados, o top-K, que utiliza este modelo de replicação não uniforme. As nossas avaliações mostram que a aplicação deste novo modelo de replicação ajuda a reduzir o espaço de armazenamento utilizado, a quantidade de dados transmitidos, e a aumentar a escalabilidade do sistema.

Agradecimentos

Este trabalho foi parcialmente suportado pelo projeto europeu LightKone (grant agreement 732505) e pelo projeto FCT/MCT NOVA-LINCS Ref. UID/CEC/04516/2013.

Referências

1. Almeida, P.S., Shoker, A., Baquero, C.: Efficient state-based crdts by delta-mutation. In: Proc. 3rd International Conference on Networked Systems, NETYS 2015 (2015)
2. Almeida, S., Leitão, J.a., Rodrigues, L.: Chainreaction: A causal+ consistent datastore based on chain replication. In: Proc. 8th ACM European Conference on Computer Systems. EuroSys '13 (2013)
3. Alonso, G.: Partial database replication and group communication primitives. In: Proc. European Research Seminar on Advances in Distributed Systems (1997)
4. Basho Technologies, Inc.: Basho Bench, `https://github.com/basho/basho_bench`
5. Basho Technologies, Inc.: Riak KV, `http://docs.basho.com/riak/kv`
6. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. 1(2) (Aug 2008)

7. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: Proc. 10th USENIX Conference on Operating Systems Design and Implementation. OSDI'12 (2012)
8. Crain, T., Shapiro, M.: Designing a causally consistent protocol for geo-distributed partial replication. In: Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data. PaPoC '15 (2015)
9. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proc. 21st ACM SIGOPS Symposium on Operating Systems Principles. SOSP '07 (2007)
10. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proc. 6th Annual ACM Symposium on Principles of Distributed Computing. PODC '87 (1987)
11. Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2) (Apr 2010)
12. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7) (Jul 1978)
13. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2) (May 1998)
14. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In: Proc. 23rd ACM Symposium on Operating Systems Principles. SOSP '11 (2011)
15. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: Proc. 10th USENIX Conference on Networked Systems Design and Implementation. nsdi'13 (2013)
16. Mahmoud, H., Nawab, F., Pucher, A., Agrawal, D., El Abbadi, A.: Low-latency multi-datacenter databases using replicated commit. Proc. VLDB Endow. 6(9) (Jul 2013)
17. Navalho, D., Duarte, S., Preguiça, N.: A study of crdts that do computations. In: Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data. PaPoC '15 (2015)
18. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1) (Mar 2005)
19. Schiper, N., Sutra, P., Pedone, F.: P-store: Genuine partial replication in wide area networks. In: Proc. 29th IEEE Symposium on Reliable Distributed Systems. SRDS '10 (2010)
20. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Proc. 13th International Conference on Stabilization, Safety, and Security of Distributed Systems. SSS'11 (2011)
21. SyncFree: AntidoteDB, `http://antidote-db.com`
22. Vogels, W.: Eventually consistent. Commun. ACM 52(1) (Jan 2009)