SwiftCloud: replicação sem coordenação

Valter Balegas and Nuno Preguiça*

CITI - Dep. Informática, FCT,
Universidade Nova de Lisboa, Portugal
v.sousa@campus.fct.unl.pt nuno.preguica@di.fct.unl.pt

Resumo Os sistemas de cloud computing são usados para suportar a execução de serviços numa escala global, muitas vezes recorrendo a repositórios chave-valor para armazenar os dados. Estes sistemas recorrem a mecanismos de replicação otimista para fornecer uma elevada disponibilidade e uma reduzida latência para os clientes. Para lidar com os potenciais conflitos, utilizam soluções que levam a atualização perdidas ou delegam nas aplicações o tratamento dos conflitos. Recentemente, foi proposta uma solução que permtie unificar automaticamente as atualizações efetuadas concorrentemente recorrendo a uma novo conceito: os tipos de dados replicados livres de conflitos, CRDTs. Neste artigo apresenta-se o SwiftCloud, um repositório de dados chave-CRDT, que estende um normal repositório chave-valor armazenando objetos que são CRDTs. Este sistema permite que as atualizações efetuadas concorrentemente sejam unificadas automaticamente. Adicionalmente, o sistema fornece um mecanismo de transações com o modelo de isolamento *mergeable snapshot isolation*, no qual uma transação acede a um snapshot consistente da base de dados. Ao contrário dos modelos de isolamento normais, as transações nunca falham, sendo as escritas unificadas recorrendo às propriedades dos CRDTs. Os resultados de avaliação mostram que o SwiftCloud impõe uma penalização de desempenho marginal quando comparado com um sistema de chave-valor normal, sem o risco de existirem escritas perdidas.

Keywords: Cloud computing, replicação, CRDT, transações, reconciliação automática

1 Introdução

Os serviços disponíveis na Internet, como por exemplo os sistemas de redes sociais, os provedores de conteúdos multimédia e os serviços de comércio eletrónico, tiveram um grande impacto na vida do seus utilizadores. Este tipo de serviços possui requisitos de funcionamento importantes, entre os quais a necessidade de fornecer uma elevada disponibilidade e um reduzida latência no acesso aos dados. Por exemplo, estudos recentes mostram que existe uma correlação entre o número de utilizadores de um serviço e o seu tempo de resposta [13].

* This work was partially supported by project PTDC/EIA-EIA/108963/2008 and a Google Research Award.

A replicação de dados é uma técnica muito usada [17,18] para satisfazer estes requisitos, permitindo manter cópias dos dados próximo dos clientes de forma a obter um reduzido tempo de resposta e aumentar a tolerância a falhas e a disponibilidade dos serviços. Infelizmente, para satisfazer estes requisitos num sistema distribuído sujeito a partições é necessário relaxar a consistência dos dados [6]. Assim, muitos sistemas tendem em relaxar a consistência para aumentar a sua disponibilidade [4,8], permitindo que o estado de diferentes réplicas divirja. Quando tal acontece, estes sistemas tendem a perder atualizações ou a exigir às aplicações que unifiquem as alterações concorrentes.

Recentemente, foi proposta uma solução baseada nas propriedades de comutatividade de operações que permite unificar automaticamente atualizações concorrentes: os tipos de dados replicados livres de conflitos (CRDTs) [11,15,14]. Os CRDTs garantem que o estados de todas as réplicas dum objeto convergem, independentemente da ordem pela qual as atualizações são integradas. Esta solução foi recentemente identificada [2] com uma potencial aproximação para contornar a impossibilidade expressa no teorema CAP, e é promissora no contexto dos sistemas de cloud computing devido a permitir que as réplicas sejam atualizadas sem coordenação.

Os repositórios chave-valor tornaram-se muito populares nos últimos anos, em especial nas infra-estruturas de cloud computing (e.g. [4,8,3]). Estes sistemas têm um modelo de dados muito simples, no qual cada objeto é indexado por uma chave. Este modelo de dados simples permite a escalabilidade do sistema através do particionamento e replicação dos objetos em múltiplas máquinas.

Neste artigo apresenta-se o SwiftCloud, um repositório de dados chave-CRDT, que estende um normal repositório chave-valor armazenando objetos que são CRDTs. Este sistema permite que as atualizações efetuadas concorrentemente sejam unificadas automaticamente recorrendo às regras definidas nos CRDTs.

O SwiftCloud inclui um mecanismo de transações com a semântica *mergeable snapshot isolation*. Este mecanismo permite que uma aplicação execute operações num conjunto de CRDTs, lendo um snapshot coerente da base de dados e fazendo com que as modificações sejam aplicadas de forma atómica. As transações nunca abortam devido a modificações concorrentes, sendo aplicadas neste caso as regras de unificação de alterações concorrentes definidas nos CRDTs.

O SwiftCloud foi implementado como um sistema de middlewarecom com base no sistema chave-valor Riak [8]. Os resultados da avaliação efetuada mostram que o desempenho do sistema é comparável aos da base de dados Riak numa aplicação realista. Adicionalmente, mostram que a replicação geográfica permite melhorar o tempo de acesso ao sistema face a uma solução baseada em apenas um centro de dados.

As contribuições deste trabalho são: *(i)* a proposta de CRDTs multi-versão e sua implementação (secção 2); *(ii)* o desenho e implementação do primeiro repositório chave-CRDT (secção 3); *(iii)* um sistema transacional implementado sobre um repositório chave-CRDT (secção 4). Este artigo apresenta ainda a

avaliação do sistema na secção 5, a discussão do trabalho relacionado na secção 6, e a conclusão na secção 7.

2 CRDTs

Nesta secção introduz-se de forma informal os CRDT e a solução proposta com suporte para versões.

2.1 Consistência futura forte e CRDTs

Modelo do sistema: Considera-se que um objeto é replicado num conjunto fixo de processos $\{p_0, p_1, ..., p_n\}$, que podem falhar e recuperar. Neste modelo excluem-se as falhas bizantinas. Os objetos são modificados pela execução de operações. As operações podem ser executadas em qualquer réplica. As réplicas sincronizam periodicamente par-a-par, trocando o estado entre si.

Consistência futura forte: O modelo de consistência futura garante que todas as operações de atualização dum objeto são entregues e executadas nas réplicas desse objeto. Em caso de atualizações concorrentes, o sistema deve incluir um mecanismo de unificação das atualizações concorrentes. Em geral, este mecanismo pode exigir a coordenação das réplicas e levar a que atualizações efetuadas numa réplica sejam descartadas ou desfeitas.

O modelo de consistência futura forte (SEC) [15] estende o modelo de consistência futura garantindo a consistência das réplicas sem que estas tenham que descartar ou desfazer o efeito de algumas operações. Para que as réplicas forneçam essas propriedades, as operações definidas devem assegurar um comportamento determinístico independentemente do contexto.

CRDT: Um CRDT é um tipo de dados que respeita o modelo SEC. Para tal, um CRDT define, além das operações de modificação do estado, um operação de *unificar* que efetua a unificação de duas versões dum CRDT. Quando se dá uma sincronização, as réplica executam localmente a operação de *merge* que produz um novo estado que inclui todas as operações executadas em cada réplica.

Shapiro et. al. [15] demostraram as condições suficientes para que um tipo de dados seja um CRDT. Brevemente, num CRDT os possíveis estados devem formar um semi-reticulado, sendo que a operação de *merge* devolve o supremo de dois estados.

Exemplo (OR-Set): Na figura 1 apresenta-se um exemplo dum CRDT conjunto, o OR-Set sem tombstones [14]). Neste CRDT, a operação de *adicionar* gera um novo identificador único, associado ao elemento adicionado. A operação de *remover* remove o valor e o identificador. A operação de *merge* garante um comportamento determinístico em que em caso de operações concorrentes sobre o mesmo valor, a adição ganha. O estado inclui para cada elemento adicionado o identificador único gerado no momento da sua criação e um vector versão que

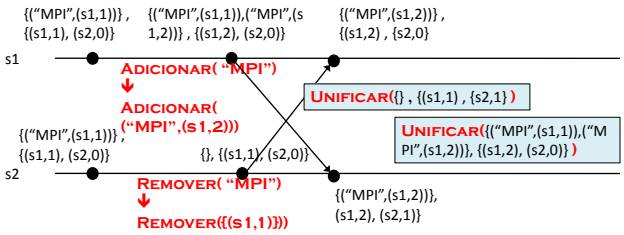


Figura 1: Exemplo de processamento de operações com o CRDT OR-Set.

mantém um sumário dos identificadores únicos observados (representado como um conjunto de pares *(servidor, contador)*).

A figura apresenta o exemplo duma execução concorrente de um *adicionar* e *remover* sobre o mesmo valor. Na operação de *merge*, ao considerar um elemento que se encontra numa réplica e não se encontra na outra, usam-se os vectores versão para decidir se o elemento deve ser mantido no resultado final - se o identificador já tiver sido observado na réplica em que o elemento não existe é porque o elemento foi removido nessa réplica, não devendo pertencer ao estado final. No exemplo da figura, o facto da réplica $s2$ possuir um vector versão que já inclui o identificador $(s1, 1)$, permite decidir que foi executada uma operação de remoção sobre esse identificador.

2.2 CRDTs Multi-Versão

O objetivo dos CRDTs multi-versão consiste em permitir aceder e alterar os valores que pertencem a um CRDT num determinado instante. Um CRDT multi-versão pode ser implementado mantendo as versões necessárias dum CRDT, como num sistema de bases de dados multi-versão.

Uma aproximação para implementar um CRDT multi-versão baseia-se na utilização dos identificadores únicos gerados nas operações (para os CRDTs que os usem). A ideia é que considerando os identificadores únicos gerados a partir da mesma fonte, é possível identificar quais as operações que foram executadas até um certo momento. Um snapshot pode ser identificado por um vector versão que indique, para cada fonte de identificadores únicos, quais os identificadores que devem ser considerados. Assim, é possível implementar um CRDT multi-versão estendendo a versão base par registar no estado as operações de remoção e definir as operações que permitem aceder ao estado desejado.

A figura 1 apresenta a especificação do OR-Set CRDT multi-versão. Nesta versão, cada elemento possui um identificador de inserção e remoção. Como podem existir remoções concorrentes, pode existir mais do que um tuplo para um dado elemento inserido. Um elemento pertence ao conjunto num dado snapshot se o identificador de inserção estiver refletido no vetor versão do snapshot e não

existir nenhum tuplo em que o vetor de remoção também esteja refletido no vetor versão.

Algoritmo 1 Especificação do CRDT OR-set multi-versão

```
1:  payload set E                          -- E: elements; element (e, insertID, removeID)
2:  initial ∅
3:  update add (element e, VersionVector VV)
4:      let id = unique()
5:      E := E ∪ (e, id, nil)
6:  update remove (element e, VersionVector VV)
7:      let rem = unique()
8:      let E' = {(e, id, nil) ∈ E | id ∈ VV}
9:      let E'' = {(e, id, rem) | (e, id, nil) ∈ E'}
10:     E = E \ E' ∪ E''
11: query contains (element e, VersionVector VV) : boolean b
12:     let b = (∀(e, id, r) ∈ E, id ∈ VV ∧ (r ∉ VV ∨ r = nil))
13: query elements () : set S
14:     let S = {e | ∀(e, id, r) ∈ E, id ∈ VV ∧ (r ∉ VV ∨ r = nil)}
15: merge (A, B): Z
16:     Z.E := A.E ∪ B.E
```

3 Arquitetura

O SwiftCloud é um repositório chave-CRDT implementado sobre um repositório chave-valor Riak [8]. A ideia consiste em enriquecer o modelo de dados deste sistema com CRDTs para que se possa usufruir de convergência automática de conflitos e do suporte para transações.

Os repositórios chave-valor permitem o acesso a dados através de uma chave. Os dados associados a cada chave não possuem qualquer organização, permitindo guardar qualquer tipo de dados binário. O sistema distribui os dados por um conjunto de nós que replicam parcialmente o conteúdo da base de dados. Os nós da rede podem estar distribuídos geograficamente ou todos instalados no mesmo centro de dados. Adicionalmente, os dados guardados nos nós dum centro de dados podem ser completamente replicados nos nós de outros centro de dados.

Ao replicarem-se os dados em diferentes localizações é possível que diferentes réplicas dos mesmos dados sejam modificadas concorrentemente. Para lidar com os conflitos, o Riak possui duas técnicas diferentes: guardar apenas o estado mais recente, ou manter os estados de todas as atualizações concorrentes.

O SwiftCloud é implementado como uma camada sobre o cliente de acesso à base de dados Riak, como se representa na figura 2 (os elementos relativos ao mecanismo de transações serão apresentados na secção seguinte). O sistema obriga a que todos os valores que são armazenados na base de dados sejam

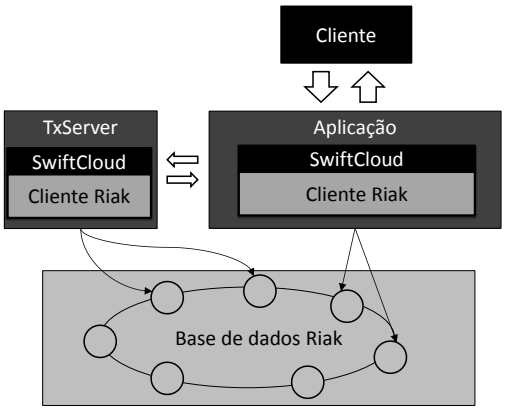


Figura 2: Arquitetura do SwiftCloud.

CRDTs. Para cada CRDT, além do seu valor, o sistema armazena um conjunto de meta-dados (e.g., classe do CRDT, vector-versão) utilizado internamente. O sistema Riak é configurado para manter todas as atualizações concorrentes efetuadas.

Quando uma aplicação executa uma operação de *put*, o CRDT começa por ser serializado para um vector de bytes, sendo posteriormente executada a operação de *put* correspondente no sistema Riak. Adicionalmente, o sistema atualiza os meta-dados de forma apropriada.

Quando uma aplicação executa uma operação de *get*, o SwiftCloud lê os valores armazenados no sistema Riak e de-serializa os CRDTs usando a meta-informação sobre o tipo do objeto. Caso exista mais do que uma versão do CRDT, o SwiftCloud efetua a unificação das versões usando a função de *merge* definida na interface do CRDT.

Esta aproximação permite oferecer a reconciliação automática das operações concorrentes que sejam efetuada. O protótipo do sistema executa sobre a versão Riak Enterprise, que inclui um mecanismo de sincronização inter-cluster, podendo ser usado para suportar replicação geográfica de forma eficiente. O mecanismo de serialização serializa os CRDTs para JSON, o que permitiria a partilha de CRDTs entre aplicações escritas em diferentes linguagens - a versão atual do protótipo apenas utiliza a linguagem Java.

4 Transações no SwiftCloud

Modelo: O sistema implementa a semântica que denominámos de *mergeable snapshot isolation*, por se assemelhar ao nível de isolamento *snapshot isolation* das bases de dados relacionais. Assim, uma transação acede a um snapshot da base de dados que reflete a execução dum conjunto de transações. As modificações duma transação tornam-se visíveis de forma atómica quando a transação

é confirmada. Em caso de existirem modificações concorrentes sobre os mesmos objetos, estas modificações são unificadas utilizando as regras definidas nos CRDTs.

Blocos base da solução: Como se explicou anteriormente, os CRDTs multi-versão permitem aceder a qualquer estado do objeto no passado identificado por um vetor versão. No SwiftCloud, as transações são identificadas através dum identificador único, podendo-se usar um vetor versão para identificar todas as transações que foram executadas no sistema até um dado momento. Assim, este vetor pode ser utilizado para aceder a um CRDT multi-versão e assim obter o estado desse CRDT nesse momento.

Para se guardar o estado da base de dados e gerar identificadores de novas transações, utiliza-se um servidor de transações (TxServer). O TxServer mantém o vetor versão da base de dados do sistema. As transações são identificadas por um identificador único composto pelo identificador do servidor e um contador. No sistema podem existir múltiplos TxServers para aumentar a tolerância a falhas, a disponibilidade e diminuir a latência do sistema. Os TxServers sincronizam periodicamente para trocar os identificadores de novas transações.

Uma transação inclui geralmente várias operações. Como as operações de um CRDT necessitam de gerar identificadores únicos, estes identificadores únicos são compostos pelo identificador da transação e um contador extra que distingue as operações de uma transação.

Protocolo: As mensagens que são trocadas para executar uma transação são mostradas na figura 3. Para se iniciar uma nova transação, o cliente envia uma mensagem de *begin* ao TxServer. Este devolve a versão da base de dados e o identificador da nova transação. No exemplo da figura, a versão da base de dados inclui duas transações, T1 e T2, e o identificador da transação iniciada pelo cliente é T3. O cliente utiliza a versão da base de dados para aceder a um estado consistente dos CRDTs armazenados no Riak e efectua operações de atualização localmente. No momento de tornar a transação persistente, o cliente envia um *commit request* ao TxServer. A mensagem enviada contém a lista de chaves modificadas localmente e o identificador da transação que lhes corresponde. O TxServer armazena as chaves dos objetos que foram modificados e concede permissão ao cliente para guardar as chaves modificadas no Riak. De seguida, o cliente escreve as novas versões dos CRDTs na base de dados e notifica o TxServer quando termina, o qual atualiza o vetor representado a versão da base de dados[1].

Se o TxServer falhar antes do cliente enviar a mensagem de *commit request*, a transação é abortada sem ter que aplicar nenhuma operação de compensação. Se o TxServer detetar que o cliente falhou, antes de receber o *commit request*,

[1] Como podem ser iniciadas múltiplas transações no TxServer que terminem por ordem diferente da do início, é necessário usar vetores versão com exceções [9] para representar o estado da base de dados - e.g. estes vetores permitem indicar que as transações T1 e T3 já terminaram, mas que a transação T2 ainda não terminou.

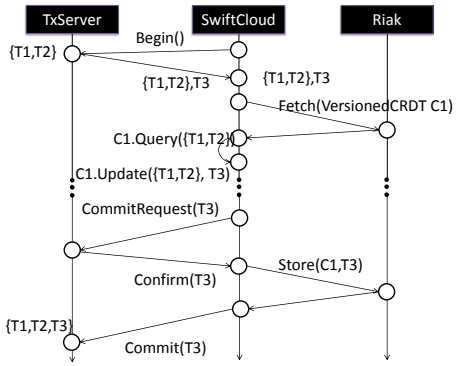


Figura 3: Protocolo de execução de uma transação no SwiftCloud.

considera que a transação terminou e não responde a qualquer mensagem posterior referente a essa transação. Quando as falhas ocorrem após a confirmação da mensagem de *commit request*, se o cliente falha, o TxServer tem que restaurar o estado dos objetos antes da transação ter início usando a lista de chaves modificadas pela transação enviadas no *commit request*, removendo qualquer referência à transação que falhou dos CRDTs. Se o TxServer falha e o cliente não consegue confirmar que a transação terminou, então este deve restaurar as modificações efetuadas por si. Se ambos falharem, as operações nunca serão visíveis porque o identificador da transação nunca será incluído na versão da base de dados.

Exemplo de código: Para utilizar o SwiftCloud, as aplicações devem delimitar as transações. Os objetos acedidos devem ser todos CRDTs e isso requer a alteração das implementações dos tipos de dados utilizados nos objetos. Para além disso, aplicações que já utilizem repositórios chave-valor apenas precisem pequenas alterações de interface para utilizar o nosso sistema - o código da listagem 2, mostra o exemplo de uma transação que adiciona um item a um carrinho de compras na aplicação TPC-W usada na avaliação do sistema.

Algoritmo 2 Exemplo de uma transação com o SwiftCloud.

```
function addToCart(cartId, itemId, quantity)
    TransactionHandler handler = SwiftClient.begin();
    SwiftObject scObjWrapper = handler.fetch("shoppingCartTable", cartId);
    ShoppingCart sc = scObjWrapper.getObject();
    sc.addItem(itemId, quantity, handler);
    handler.commit();
```

5 Avaliação

Para testar o sistema baseamo-nos numa implementação do TPC-W [7] e implementámos uma versão que utiliza o Riak e duas versões com o SwiftCloud, uma que usa transações e outra que não usa. A versão sem transações permite-nos avaliar o peso introduzido pelos CRDTs face à nossa base de comparação, a implementação para Riak. A versão transacional permite avaliar o custo de fornecer transações no sistema.

A implementação do benchmark foi adaptada para utilizar CRDTs com a versão SwiftCloud. Os CRDT utilizados foram o LWW-Register [14] para representar as entidades que são apenas de leitura e o OR-Sets para representar o carrinho de compras e outras entidades que requerem a gestão de um conjunto de dados. Nos resultados apresentados neste artigo apenas utilizaremos a configuração *ordering* do benchmark para analisar a performance. Nesta configuração, a percentagem de operações de escrita corresponde a 50%. Os resultados com as outras configurações apresenta um comportamento semelhante.

Configurações do sistema: Foram utilizadas duas configuração do sistema, instaladas na plataforma EC2 da Amazon. A primeira pretende medir o desempenho base do sistema (1-Cluster). A base de dados Riak é composta por dois nós, ambos instalados num centro de dados na Europa. Os clientes executam em outras máquinas, todas instaladas no mesmo centro de dados. A segunda configuração pretende avaliar os ganhos obtidos por replicar os dados em diferentes localizações geográficas (2-Cluster). Nesta configuração, temos nós de base de dados em dois centros de dados, na Europa e na costa oeste dos Estados Unidos. Cada base de dados replicada é composta por dois nós e replica completamente o estado da base de dados, sincronizando periodicamente utilizando o mecanismo de sincronização do Riak Enterprise estendido com a sincronização dos TxServers. Os clientes são igualmente distribuídos pelos centros de dados e acedem à réplica mais perto de si. Quando se usa o sistema Riak, só existe uma base de dados, pois o sistema não permite escritas em centros de dados diferentes sem a perda de informação ou a necessidade da aplicação lidar com os conflitos. As máquinas utilizadas para a base de dados são as correspondentes ao perfil *EC2.Medium* e as máquinas dos clientes correspondem ao *EC2.Small*. Todas as máquinas correm a mesma distribuição modificada de Linux 64bit e a versão do Riak utilizada é a Enterprise 1.0. A latência entre duas máquinas no mesmo data-center é aproximadamente 0.5 ms e entre os centros de dados da Europa e Estados Unidos é aproximadamente 100 ms.

1-Cluster: A figura 4 mostra o débito do sistema para as três diferentes implementações. No gráfico observa-se que o sistema escala linearmente até alcançar o limite de desempenho da base de dados. A implementação sem transações fornece um desempenho muito similar ao Riak. A versão transacional apresenta uma quebra reduzida na performance, resultante da comunicação com o TxServer e da gestão dos CRDTs multi-versão.

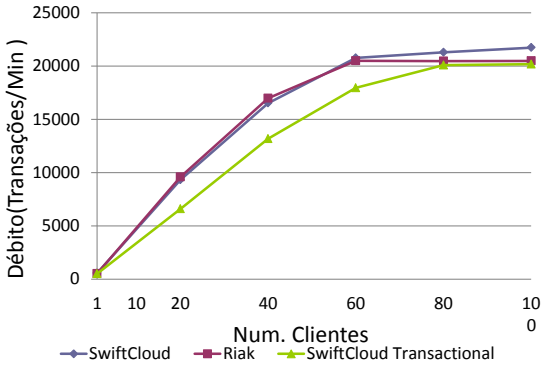
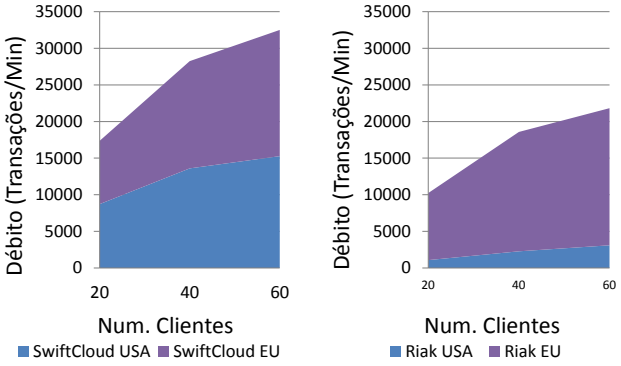


Figura 4: Débito do benchmark TPC-W para diferentes implementações.



(a) SwiftCloud (b) Riak

Figura 5: Débito do sistema TPC-W com clientes em diferentes data-centers.

2-Cluster: Os resultados, quando existem clientes a aceder ao sistema a partir de localizações distintas, mostram que o sistema pode sofrer benefícios consideráveis por ter os dados replicados num centro de dados local. A figura 5 compara o débito as implementações Riak e SwiftCloud. Observa-se que a versão SwiftCloud tem um débito quase duas vezes superior ao sistema Riak devido à existência de um centro de dados perto dos clientes que executam nos Estados Unidos. No sistema Riak com apenas um centro de dados, a latência dos clientes presentes no centro de dados remoto é também muito superior.

6 Trabalho relacionado

A replicação é um tema que tem sido estudado em detalhe no contexto dos sistemas de gestão de dados. Nos ambientes de cloud computing têm sido usadas, em geral, técnicas de replicação otimista [12]. Estas técnicas permitem

uma elevada disponibilidade, mas necessitam de lidar com as modificações concorrentes. Os CRDT [14] são um das técnicas propostas para lidar com este problema, baseada na utilização das propriedades de comutatividade das operações definidas num objeto. Os primeiros CRDTs mantinham uma sequência de caracteres e eram usados no contexto da edição cooperativa de texto [11,19], sendo propostos como uma alternativa às técnicas de transformação de operações [17,5]. Mais recentemente foram propostas outros tipos de dados, incluindo conjuntos, mapas, registos, contadores e grafos[14,15,1]. Este artigo apresenta uma nova variante de CRDTs: o CRDT multi-versões.

Um elevado número de sistemas usou técnicas de replicação para melhor a disponbilidade e escalabilidade. O Ganymed [10] é um middleware que faz a replicação de uma base de dados e que utiliza um servidor para processar todas as escritas e vários servidores para processar as leituras. Esta solução aumenta a disponibilidade para leitura do sistema, mas no entanto a performance do sistema continua a ser limitada pela quantidade de escritas no sistema, pois são todas processadas pela mesma máquina.

O PNUTS! [3] estende a aproximação anterior para um sistema de cloud computing. Neste sistema, cada elemento de dados tem um primário que se localiza no centro de dados mais próximo do utilizador que o usa mais frequentemente - as escritas são efetuadas inicialmente no primário e propagadas para as restantes réplicas. Esta aproximação permite uma baixa latência para as leituras, mas em caso de escritas pode levar a uma elevada latência. Adicionalmente, não inclui suporte para transações.

O sistema Dynamo [4] (e derivados como o Cassandra e o Riak) permitem uma elevada disponibilidade através da replicação e particionamento dos dados. No entanto, na presença de escritas concorrentes levam a atualizações perdidas ou exigem que as aplicações lidem com os conflitos.

O Walter [16] é um sistema que providencia geo-replicação e suporte transacional. Neste sistema, as transações requerem a coordenação das réplicas caso as transações envolvam objetos que não são sejam conjuntos. O SwiftCloud permite efetuar transações com baixa latência que envolvam outros tipos de dados.

7 Conclusão

A resolução automática de conflitos é uma técnica desejável nos sistemas de cloud computing pois permite simplificar o desenvolvimento de aplicações.As transações também simplificam o desenvolvimento de aplicações, porque evitam que estas sejam expostas a estados intermédios do sistema.

Neste artigo apresentámos um repositório *chave-valor* que tira partido das propriedades dos CRDTs para fornecer resolução automática de conflitos e suporte para transações num modelo de replicação consistência futura. O sistema foi desenvolvido como um middleware que não requer a alteração da infraestrutura para funcionar, o que facilita a sua integração. Os resultados mostram um baixo custo comparativamente com o Riak e grandes benefícios quando utilizado para replicar a base de dados em localizações próximas dos clientes.

Referências

1. Khaled Aslan, Pascal Molli, Hala Skaf-Molli, and Stéphane Weiss. C-Set : a Commutative Replicated Data Type for Semantic Stores. In *RED: Fourth International Workshop on REsource Discovery*, Heraklion, Greece, May 2011.
2. Eric A. Brewer. Pushing the cap: Strategies for consistency and availability. *IEEE Computer*, 45(2):23–29, 2012.
3. Brian Cooper and et. al. PNUTS: Yahoo!'s hosted data serving platform. In *Proc. 34th VLDB*, 2008.
4. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proc. SOSP '07*, 2007.
5. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, June 1989.
6. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, page 51.
7. P. Gomes, J. Pereira, and R. Oliveira. An object mapping for the cassandra distributed database. 2011.
8. Rusty Klophaus. Riak Core. In *ACM SIGPLAN Commercial Users of Functional Programming on - CUFP '10*, New York, New York, USA.
9. Dahlia Malkhi and Doug Terry. Concise version vectors in WinFS. *Distributed Computing*, 20(3):209–219, September 2007.
10. Christian Plattner and Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proc. Middleware'04*, 2004.
11. Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A Commutative Replicated Data Type for Cooperative Editing. In *Proc. of ICDCS '09*, June 2009.
12. Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), 2005.
13. E. Schurman and J. Brutlag. Performance related changes and their user impact. Presented at Velocity Web Performance and Operations Conference, 2009.
14. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche 7506, INRIA, Rocquencourt, France, January 2011.
15. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proc. SSS'11*, 2011.
16. Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proc. SOSP'11*, 2011.
17. Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, March 1998.
18. D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP '95*, 1995.
19. Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Proc. of ICDCS '09*, 2009.