ChainReaction: uma Variante de Replicação em Cadeia com Coerência Causal+*

Sérgio Almeida, João Leitão, Luís Rodrigues
sergiogarrau@gsd.inesc-id.pt, jleitao@gsd.inesc-id.pt, ler@ist.utl.pt

INESC-ID, Instituto Superior Técnico, Techincal University of Lisbon

Resumo O equilíbrio entre garantias de coerência, disponibilidade e desempenho, em particular nos sistemas que suportam Geo-replicação, é um dos principais desafios no desenho de sistemas de armazenamento de dados para suporte a aplicações na nuvem. Por este motivo, várias combinações de diferentes modelos de coerência e técnicas de replicação têm sido propostas nos últimos anos. Este artigo propõe um novo sistema de armazenamento de dados, chamado ChainReaction, que oferece coerência "causal+"com elevado desempenho, tolerância a faltas, e capacidade de escala. Baseando-se numa especialização da técnica de replicação em cadeia (chain-replication), o ChainReaction evita os pontos de estrangulamento associados à coerência atómica, e oferece um desempenho competitivo com os sistemas que apenas oferecem coerência eventual. Para além disso, o ChainReaction é apresentado como uma base para uma futura extensão para o cenário de Geo-replicação.

1 Introdução

As aplicações construídas de acordo com o paradigma de Computação em Nuvem têm vindo a crescer tanto em importância como em número. Isto leva a que o problema de se manter o equilíbrio entre coerência, disponibilidade e desempenho seja ainda mais grave neste cenário. Este problema é capturado no bem conhecido teorema CAP [1], que afirma que não é possível oferecer, ao mesmo tempo, coerência, disponibilidade (em caso de partições de rede) e escalabilidade. Desta forma, vários sistemas de armazenamento de dados foram propostos nos últimos anos [2,3,4,5]. Estes oferecem diferentes combinações de garantias de coerência e de protocolos de replicação. Uns optam por oferecer garantias mais fracas (coerência eventual) de forma a obter o desempenho esperado. No entanto, estas garantias mais fracas impõem um maior esforço ao programador. Por outro lado, as soluções que oferecem garantias mais fortes, como por exemplo a coerência atómica, facilitam a vida do programador mas sofrem de problemas de escalabilidade.

* Este trabalho foi parcialmente apoiado pela Fundação para a Ciência e Tecnologia por via do financiamento multianual do INESC-ID através do programa de bolsas PIDDAC, no âmbito do projecto PEst-OE/ EEI/ LA0021/ 2011, e por via do projecto HPCI (PTDC/ EIA-EIA/ 102212/ 2008).

Este artigo visa contribuir para o avanço deste campo através da proposta de uma nova arquitectura para um sistema de armazenamento de dados, chamado ChainReaction. A nossa solução baseia-se numa especialização da técnica de replicação em cadeia (chain-replication), que oferece coerência "causal+"(recentemente formalizada em [5]) e que permite beneficiar da existência de múltiplas réplicas para distribuir a carga de leituras. Como resultado, o ChainReaction evita o baixo desempenho de soluções que oferecem coerência atómica e oferece um desempenho competitivo com os sistemas que oferecem garantias mais fracas. Para mais, o ChainReaction serve de base a um futuro sistema para Geo-replicação (vários centros de dados).

Avaliámos experimentalmente a nossa solução através da utilização do Yahoo Cloud Serving Benchmark [6] sobre um protótipo da nossa solução. Esta bancada de testes foi também aplicada ao Apache Cassandra [2] e ao FAWN-KV [3].

O resto do artigo está estruturado da seguinte forma. A Secção 2 descreve algum trabalho relacionado focando-se em algumas soluções que ofereçem armazenamento de dados para aplicações na nuvem. Na Secção 3 apresentamos a nossa solução. A Secção 4 inclui os resultados da avaliação experimental. Por fim, na Secção 5 apresentamos algumas direcções em termos de trabalho fufuro e na secção 6 apresentamos as conclusões.

2 Trabalho Relacionado

Um sistema de armazenamento de dados para aplicações baseadas no paradigma de computação em nuvem (Geo-replicado), tem de ter em conta a ocorrência de faltas, a localidade dos clientes (latência) e a ocorrência de partições de rede. Desta forma, este tipo de sistemas tem de implementar algum mecanismo de replicação que considere a existência de centros de dados distantes. De entre as técnicas de replicação mais conhecidas podemos enumerar as seguintes: replicação activa (tipicamente baseada no Paxos [7]), replicação passiva ("primary-backup"), sistemas de quóruns e replicação em cadeia (chain-replication). Não nos vamos focar na descrição de cada uma das técnicas devido à falta de espaço e ao facto de estas serem bem conhecidas hoje em dia. De seguida apresentamos uma breve descrição da técnica de replicação em cadeia, dado que esta está intimamente relacionada com a nossa solução.

A replicação em cadeia (ou Chain Replication), introduzida em [8], consiste numa abordagem do tipo "primary-backup"que permite a construção de sistemas de armazenamento de dados que oferecem coerência atómica, alto desempenho e disponibilidade. Esta técnica assume que as réplicas estão organizadas numa *cadeia*. Esta cadeia consiste num conjunto de nós em que cada nó tem um sucessor e um predecessor à excepção do primeiro (cabeça) e último (cauda). As operações de escrita são sempre processadas pela cabeça e são de seguida propagadas pela cadeia até chegarem à cauda (ponto onde se retorna ao cliente). A forma como as escritas são propagadas asseguram uma propriedade que tem o nome de *update propagation invariant*. Ao contrário das operações de escrita, as operações de leitura são sempre processadas pela cauda da cadeia. Como é

garantido que os valores contidos na cauda já passaram por toda a cadeia, as leituras são sempre coerentes. Esta técnica tem um maior custo em termos de latência do que soluções baseadas em protocolos de difusão, no entanto a sua simplicidade levou-a a ser adoptada por vários sistemas.

Posto isto, estes sistemas lidam com um dilema aquando da escolha entre coerência, disponibilidade e eficiência. Em suma, um sistema de armazenamento de dados distribuído para aplicações Geo-replicadas tem de considerar dois aspectos complementares: i) A coerência e o desempenho das operações num único centro de dados; ii) A coerência e o desempenho das operações submetidas por clientes que acedem a vários centros de dados.

Discutimos agora como é que algumas das técnicas descritas anteriormente têm sido aplicadas em diferentes sistemas de armazenamento de dados. O Apache Cassandra [2] é um sistema que oferece garantias de coerência atómica através da utilização de quóruns. Este também pode ser adaptado para um cenário com vários centros de dados. Em ambos os cenários os quóruns podem ser configurados para oferecer garantias mais fortes, em detrimento da sua escalabilidade, ou garantias mais fracas. A técnica de quóruns é também utilizada no Amazon Dynamo [4], onde a configuração do tamanho dos quóruns define as garantias oferecidas. Se se usar coerência eventual, este sistema emprega um mecanismo de resolução de conflitos baseado em relógios vectoriais. O Dynamo pode ser configurado para vários centros de dados, no entanto só pode oferecer garantias de coerência eventual. O FAWN-KV [3] e o Hyperdex [9] são exemplos de sistemas que oferecem garantias de coerência atómica através da utilização da replicação em cadeia. O Google Megastore [10] é um outro exemplo de um sistema adaptável a um cenário de Geo-replicação, oferecendo transacções com garantias de serialização.

O COPS [5] é um sistema desenhado para oferecer alta escalabilidade num cenário de Geo-replicação. Para isso, faz uso de um modelo de coerência fraco, que ao contrário da coerência eventual oferece algumas garantias ao programador. Este modelo de coerência chamado de *causal+*, garante que a ordenação das operações respeita a ordem causal [11] e que operações concorrentes são ordenadas de uma forma coerente em todos os centros de dados[1]. Para oferecer estas garantias, o COPS requer que os seus clientes mantenham um conjunto de metadados que codificam as *dependências* entre as operações. Estas dependências são incluídas nos pedidos de escrita feitos pelo cliente. As garantias e a escalabilidade do COPS são resultado do uso destas dependências, uma estratégia que também utilizamos. No entanto, ao contrário do COPS, não necessitamos que o centro de dados local ofereça garantias de coerência atómica (em [5] os autores utilizam a replicação em cadeia), pois esta abordagem tem um custo adicional nas operações realizadas num único centro de dados.

[1] De facto, este modelo de coerência já tinha sido usado antes [12,13], mas foi apenas formalizado como *causal+* em [5].

3 ChainReaction

Nesta secção descrevemos a operação do ChainReaction num único centro de dados, como base para um sistema mais complexo que suporte Geo-replicação. Começa-se por introduzir resumidamente os critérios de coerência suportados pela nossa solução (importados do COPS [5]), seguindo-se uma explicação geral da arquitectura do sistema e a subsequente explicação de cada um dos seus componentes.

3.1 Modelo de Coerência

No nosso sistema decidimos oferecer o modelo de coerência *causal+*[12,13,5]. Seleccionámos este modelo pois oferece um bom equilíbrio entre coerência e desempenho. Ao contrário da coerência atómica, este modelo permite uma quantidade razoável de paralelismo no processamento de operações concorrentes. Mesmo assim, garante que as operações concorrentes de escrita são ordenadas globalmente, evitando a existência de réplicas divergentes (um problema comum da causalidade). Por outro lado, oferece melhores garantias sobre o estado observado pelas aplicações, ao contrário da coerência eventual.

3.2 Visão Geral da Arquitectura

A nossa arquitectura é semelhante à arquitectura do FAWN-KV. Consideramos que o sistema é composto por vários *servidores de dados* e várias *proxies*. Os servidores de dados são responsáveis por processar os pedidos de escrita e de leitura para um ou mais objectos. As proxies recebem os pedidos dos utilizadores finais (por exemplo, um browser) e redireccionam os pedidos para o servidor de dados apropriado.

Os servidores de dados estão organizados numa DHT (Distributed Hash Table) que utiliza dispersão coerente (*consistent hashing*) para distribuir os objectos pelos servidores. Cada objecto é guardado em R servidores consecutivos no anel da DHT. Os servidores responsáveis por um dado objecto executam o protocolo de replicação em cadeia para manter as cópias coerentes: o nó responsável pelo objecto no anel corresponde à cabeça da cadeia e o seu $R - 1$ sucessor corresponde à cauda da cadeia. É de notar que em resultado da dispersão dos dados um servidor de dados pode servir vários objectos, podendo fazer parte de múltiplas cadeias. A função de um servidor de dados também pode ser diferente consoante o objecto: pode ser a cabeça da cadeia para um objecto, um nó intermédio para outro e a cauda para um terceiro objecto.

Assumimos ainda, que os servidores, mesmo sendo em grande número, podem ser mantidos numa "one-hop DHT" [14]. Desta forma, cada nó do sistema consegue mapear localmente as chaves dos objectos para um dado servidor, sem ter de se dirigir a um serviço de directório externo.

Considerando a arquitectura descrita acima, descrevemos agora o ciclo de vida de um pedido típico no FAWN-KV que utiliza o protocolo de replicação em cadeia original. Os pedidos enviados pelo cliente são recebidos numa das

proxies que utiliza uma função de dispersão para decidir qual o servidor que deve processar o pedido: se for uma escrita é redireccionada para a cabeça da cadeia; se for uma leitura é directamente processada pela cauda da cadeia. No caso da escrita, o pedido é processado pela cabeça e encaminhado para o próximo nó na cadeia, até chegar à cauda (onde termina a escrita). Tanto nas operações de leitura como nas de escrita, a cauda é responsável por enviar a resposta para a proxy que em seguida a envia para o cliente. O ChainReaction utiliza uma variante deste processo cuja explicação será aprofundada nas próximas secções.

3.3 Variante de Replicação em Cadeia

O protocolo de replicação em cadeia original, descrito acima, é capaz de oferecer coerência atómica. De facto, todas as operações (i.e., leituras e escritas) têm de ser serializadas e processadas por um único nó, a cauda da cadeia. A desvantagem desta abordagem é que a existência de diversas réplicas não promove a distribuição da carga de leituras entre elas.

No ChainReaction decidimos oferecer coerência *causal+* pois desta forma permitimos uma melhor utilização dos recursos existentes. Também possibilitamos a adição de mais nós que podem ser usados para distribuição da carga, com poucos efeitos negativos.

A nossa abordagem parte da seguinte observação: se um nó na cadeia, digamos x, está causalmente coerente de acordo com um conjunto de operações de um dado cliente, então todos os predecessores de x na cadeia estão também coerentes. Esta propriedade deriva da *invariante na propagação de escritas* do protocolo original. Desta forma, considere-se que um cliente observa o valor do objecto O retornado pelo nó x numa operação de leitura ou escrita, denominada *op*. As próximas operações de leitura sobre O, que dependem causalmente de *op*, têm de ser processadas por um nó entre a cabeça da cadeia e o nó x de forma a obter um estado coerente (de acordo com os critérios do *causal+*). No entanto, a partir do momento que a operação *op* fica estável (i.e., já foi processada pela cauda), as próximas leituras sobre O, que dependem causalmente de *op*, podem ser processadas por qualquer nó da cadeia.

O ChainReaction utiliza este facto para distribuir a carga das leituras pelas várias réplicas. Adicionalmente, permite ainda aumentar a cadeia (de forma a ter mais réplicas para distribuição de carga) sem aumentar a latência das operações de escrita, permitindo que estas retornem assim que forem processadas pelas primeiras k réplicas (onde k, normalmente inferior ao tamanho da cadeia, define o número de faltas toleradas pelo sistema). A propagação das operações desde o nó k até à cauda pode ser feita de forma "lazy", cujo propósito é aumentar o número de réplicas que podem processar as leituras.

Para garantir a correcção das operações de leitura, de acordo com o modelo de coerência *causal+*, sobre múltiplos objectos, os clientes têm de guardar a posição da cadeia onde foi processada a sua última leitura para cada um dos objectos. Esta informação é guardada na forma de *metadados* que são armazenados na *biblioteca cliente*, que é responsável por gerir estes metadados do lado do cliente. Adicionalmente, garantimos que os resultados das operações de escrita

apenas ficam visíveis depois das suas dependências causais terem estabilizado no centro de dados local. Nas próximas subsecções discutimos a biblioteca cliente do ChainReaction e descrevemos pormenorizadamente as operações de leitura e escrita.

3.4 Interface e Biblioteca Cliente

A interface básica oferecida pelo ChainReaction é semelhante à interface da maioria dos sistemas de armazenamento de dados existentes. As operações que fornecemos aos clientes são as seguintes:

put(key, val): uma operação de PUT permite atribuir (*i.e.*, escrever) o valor *val* num objecto identificado pela chave *key*. De acordo com a descrição da próxima secção, as operações de PUT são sempre aplicadas sobre a versão mais recente do objecto.

val ← get(key): Um GET retorna (*i.e.*, lê) o valor do objecto identificado pela chave *key*, reflectindo as operações de PUT anteriores.

Estas operações são fornecidas ao cliente pela biblioteca cliente que é responsável pela gestão dos metadados que são adicionados aos pedidos e extraídos das respostas. Quando consideramos o sistema instalado em apenas um centro de dados, os metadados são guardados numa tabela que inclui uma entrada por cada objecto acedido. Cada entrada consiste num tuplo *(key, version,* chainIndex*)*. O *chainIndex* consiste num identificador que captura a posição do nó, na cadeia, que processou e respondeu ao último pedido do cliente, para um dado objecto. Quando um cliente invoca uma leitura sobre um objecto identificado por *key*, os metadados descritos acima têm de ser incluídos. Caso não exista uma entrada para o objecto não são enviados quaisquer metadados (i.e., o cliente ainda não viu qualquer versao do objecto, logo pode ler em qualquer nó). Para mais, o ChainReaction pode ter de actualizar os metadados após executar uma operação.

3.5 Processamento de Operações Put

Nesta secção fornecemos uma descrição pormenorizada de como as operações PUT são processadas pelo ChainReaction. Quando um cliente invoca uma operação de PUT, através da biblioteca cliente, a chave *key* e o valor *val* são enviados para o centro de dados local e interceptados por uma proxy. A biblioteca no cliente também acrescenta à mensagem os metadados respeitantes à operação PUT anterior assim como os metadados respeitantes às operações GET realizadas desde o último PUT. Os metadados são mantidos apenas para os objectos cujas versões ainda não estão estáveis; as versões estáveis não impõem qualquer restrição nas operações de PUT e GET. Como nos focamos em melhorar o desempenho das leituras oferecendo garantias de coerência *causal+*, optámos por atrasar (ligeiramente) as escritas da seguinte forma: uma operação de PUT tem de garantir que todas as versões, das quais depende causalmente, estão estáveis (i.e., foram aplicadas nas caudas das respectivas cadeias) antes de ser executada. Isto garante

que nenhum cliente é capaz de ler versões de dois objectos diferentes que podem violar as dependências causais.

Assim que as versões reflectidas nas dependências estejam estáveis, a proxy utiliza a função de dispersão para enviar o pedido para a cabeça da cadeia correspondente. A cabeça processa o pedido de PUT, atribuindo uma nova versão ao objecto, e enviando o pedido para baixo na cadeia como no protocolo original. No entanto, o pedido é propagado apenas até ao nó k (chamamos a esta fase *eager phase*), momento em que uma resposta é enviada para o proxy, que por sua vez a envia para o cliente. Finalmente, a biblioteca recebe a resposta e extrai os metadados da mesma, actualizando as entradas correspondentes (substitui a versão do objecto pela nova versão, e o *chainIndex* pelo valor de k).

Em paralelo com o envio da resposta, a operação de PUT continua a ser propagada de uma forma "lazy"até chegar à cauda da cadeia. Como referido anteriormente, um servidor de dados pode pertencer a várias cadeias, sendo assim, pode ter de processar escritas para diferentes objectos. As operações que são propagadas de forma "lazy"têm uma menor prioridade que as operações na fase "eager". Embora o processamento destas operações seja igual, a prioridade atribuída garante que a latência das operações de escrita sobre um dado objecto não são afectadas negativamente pelo factor de replicação adicional de outro objecto. Quando as operações de PUT chegam à cauda, a versão é considerada como estável e uma mensagem de confirmação é enviada para cima na cadeia para informar os restantes nós. Esta mensagem inclui a chave e a versão do objecto e é propagada até à cabeça da cadeia.

3.6 Processamento de Operações Get

Nesta secção fazemos uma descrição pormenorizada de como as operações GET são processadas pelo ChainReaction. Um GET é processado de uma forma diferente de um PUT. Quando a biblioteca cliente recebe um GET, esta consulta os metadados existentes para a *key* pedida e encaminha o pedido juntamente com a versão e o *chainIndex* para a proxy. Esta última utiliza o *chainIndex* incluído nos metadados para decidir qual o nó que vai servir o GET. Se o *chainIndex* for igual a R (i.e., a versão está estável), o pedido pode ser processado por qualquer nó na cadeia (seleccionado aleatoriamente). Caso contrário a proxy escolhe um servidor de dados t com um índice entre 0 (cabeça da cadeia) e *chainIndex*. Esta estratégia permite distribuir a carga de leituras pelos vários servidores de dados disponíveis. O servidor de dados t processa o pedido e envia uma resposta para a proxy com o valor do objecto, a versão lida e os metadados relevantes. Em seguida, a proxy encaminha esta resposta para o cliente (incluindo todos os campos referidos anteriormente). A biblioteca cliente recebe a resposta e extrai os metadados actualizando as entradas relevantes. Considerando que *tindex* corresponde ao índice do nó t, que *newversion* corresponde à versão retornada pela operação e que *pversion* corresponde à versão guardada no cliente. Os metadados são actualizados da seguinte forma: i) Se *newversion* estiver estável, então o *chainIndex* é colocado a R, a versão é também actualizada caso seja maior que *pversion*; ii) Se *newversion* é igual a *pversion*, então o *chainIndex* é

colocado a $max(chainIndex, tindex)$; iii) Se *newversion* é maior do que *pversion* então o *chainIndex* é colocado a *tindex* (a versão é também actualizada para *newversion*).

3.7 Tolerância a Faltas

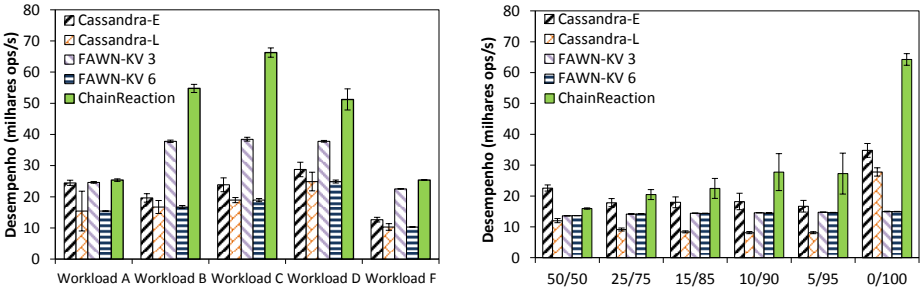
Dado que o ChainReaction é baseado no protocolo de replicação em cadeia, os mecanismos de recuperação de faltas dos nós da cabeça, cauda e do meio de uma cadeia são os mesmos do que no protocolo original. No entanto, ao contrário da replicação em cadeia, o ChainReaction consegue continuar a processar pedidos mesmo que a cauda falhe. No nosso sistema, uma cadeia com R nós consegue suportar $R - k$ falhas, dado que não se consegue processar um PUT com menos de k nós. Por outro lado, a reconfiguração da cadeia devido a uma falha de um nó pode invalidar parte dos metadados presentes no cliente, nomeadamente a semântica do *chainIndex*. Para contornar este cenário é possível usar a última versão lida presente também no cliente. Se o nó que processa o GET não tem uma versão igual ou superior à última versão observada pelo cliente, o pedido de GET é então encaminhado para os seus predecessores até que um destes tenha a versão desejada.

4 Avaliação Experimental

Nesta secção apresentamos os resultados experimentais obtidos da nossa solução. Os resultados foram obtidos a partir de um protótipo da nossa solução. Este protótipo foi implementado sobre uma versão do FAWN-KV, que optimizamos, como ponto de partida (os resultados apresentados nesta secção relativos ao FAWN-KV foram extraídos utilizando a versão optimizada deste sistema). No entanto, não implementámos qualquer mecanismo de recuperação de faltas, dado que este não é o foco central da nossa contribuição.

Extraímos resultados de desempenho em comparação com outros dois sistemas: o FAWN-KV (versão optimizada) e o Apache Cassandra. A avaliação aqui relatada consistiu em medir o desempenho do ChainReaction num único centro de dados e compará-lo com o desempenho do FAWN-KV e do Apache Cassandra 0.8.10. Todos os resultados obtidos correspondem à média de 10 execuções independentes. Os intervalos de confiança estão evidenciados em todas as figuras.

O nosso cenário experimental contém 9 servidores onde os sistemas foram instalados e 1 nó para gerar a carga nos sistemas. Cada nó tem o sistema operativo Ubuntu 10.04.3 LTS e conta com 2x4 core Intel Xeon E5506 CPUs, 16GB RAM e 1TB de disco rígido. Os nós estão interligados por uma rede Ethernet de 1 Gbit. Nos testes utilizámos 5 configurações diferentes para os sistemas, os quais se descrevem de seguida: i) Configuração do Apache Cassandra com um factor de replicação igual a 6 e que oferece coerência eventual (Cassandra-E). As operações de escrita são aplicadas em 3 nós antes de retornar, enquanto as operações de leitura podem ser feitas em apenas um nó; ii) Configuração do Cassandra também com 6 réplicas mas que oferece garantias de coerência atómicas



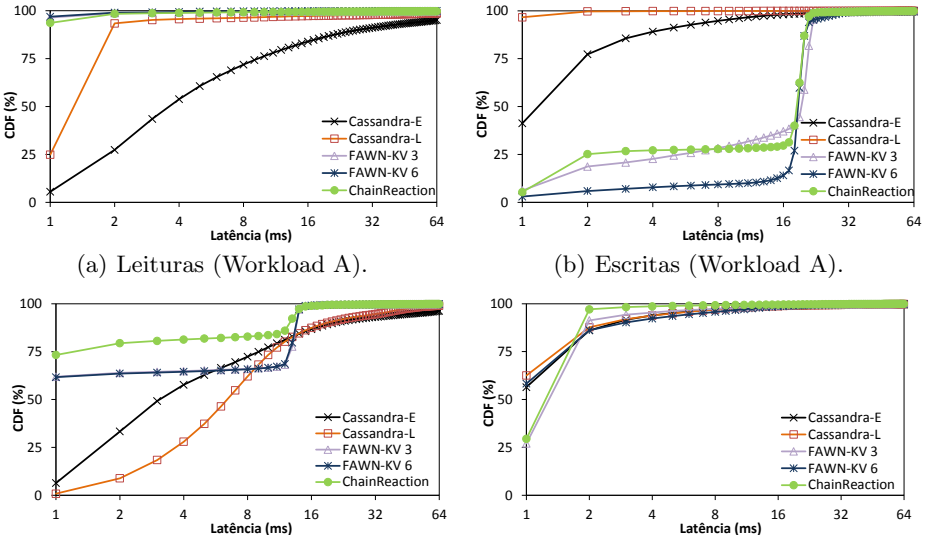
(a) Testes de Carga do YCSB. (b) Testes de Carga Personalizados.

Figura 1. Desempenho dos sistemas testados.

(Cassandra-L). As escritas e leituras são sempre aplicadas num quórum de nós antes de retornarem (4 nós); iii) Configuração do FAWN-KV com 3 réplicas que oferece coerência atómica (FAWN-KV 3), através da técnica de replicação em cadeia; iv) Configuração do FAWN-KV com 6 réplicas. Também oferece garantias de coerência atómica (FAWN-KV 6); v) Configuração do ChainReaction com $R = 6$ e $k = 3$. Oferece coerência *causal+*.

Todas estas configurações foram submetidas a testes de carga do Yahoo! Cloud Serving Benchmark (YCSB) 0.1.3 [6]. Utilizámos os testes de carga originais do YCSB (à excepção do workload E, visto o FAWN-KV não suportar operações de scan) com 1.000.000 de objectos. Nas nossas experiências os objectos tinham um tamanho de 1 KByte. Também realizámos um teste localizado através do uso de testes de carga personalizados com um único objecto, onde o rácio de escritas/leituras varia entre 50/50 e 0/100. Este teste permite verificar o comportamento da nossa solução quando apenas uma cadeia se encontra activa. Todos os testes de carga foram executados num cliente YCSB que utiliza 200 threads para simular 200 clientes, que no total submetem 2.000.000 de operações.

Os resultados de desempenho (milhares de operações por segundo) são apresentados na Figura 1. Os resultados da função de distribuição cumulativa (Cumulative Distribution Function) para a latência são apresentados nas Figuras 2 e 3. A Figura 1(a) mostra que o ChainReaction num único centro de dados é superior tanto ao FAWN-KV como ao Cassandra. Com as cargas A e F (que contêm muitas operações de escrita) o desempenho do ChainReaction aproxima-se do Cassandra-E e do FAWN-KV 3. Isto seria de esperar dado que a nossa modificação ao protocolo de replicação em cadeia não se foca na optimização das operações de escrita. De facto, para uma elevada carga de escritas seria de esperar que a nossa solução fosse inferior ao FAWN-KV 3, isto porque o ChainReaction precisa de 6 réplicas em vez de 3 (i.e., cada nó é responsável por um maior número de objectos em média). Este efeito é compensado pelos ganhos nas operações de leitura. Isto pode ser observado nos resultados para a latência nas Figuras 2(a) e 2(b). Estas figuras mostram também que o Cassandra exibe uma melhor latência nas escritas. No entanto é de notar que as operações de leitura



(a) Leituras (Workload A). (b) Escritas (Workload A).

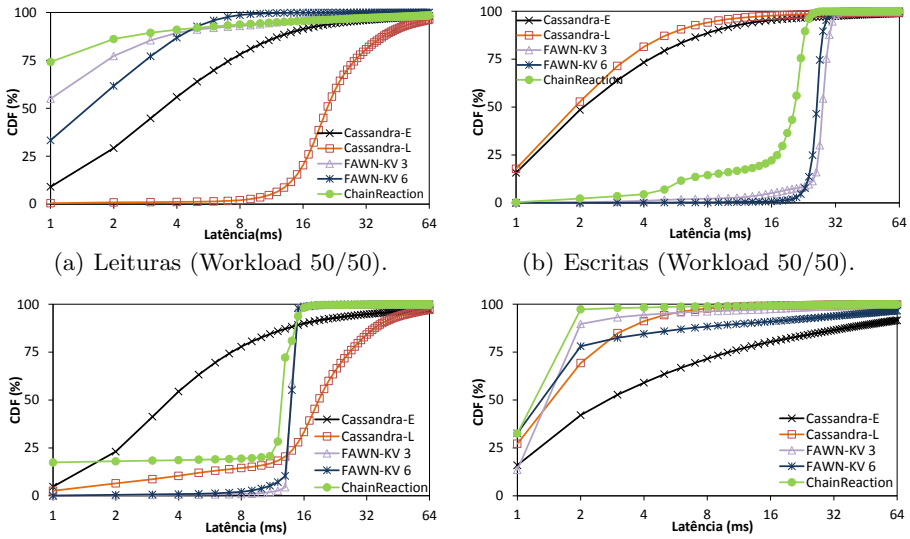
(c) Leituras (Workload B). (d) Escritas (Workload B).

Figura 2. Função de Distribuição Cumulativa (CDF) da Latência para os Testes de Carga do YCSB (é de notar que a latência se encontra numa escala logarítmica).

no Cassandra têm uma latência muito superior em relação ao ChainReaction e ao FAWN-KV.

Por outro lado, para a carga B (com grande número de leituras) seria de esperar que o ChainReaction supere as restantes soluções. Isto confirma-se com uma melhoria de 178% em relação ao Cassandra-E e uma melhoria de 45% em relação ao FAWN-KV 3. Os resultados para o workload D (Figura 1(a)) são semelhantes a estes resultados. É de notar que a latência das operações na nossa solução é muito inferior comparando com as restantes soluções (Figuras 2(c),2(d)). Adicionalmente, no workload C (só leituras) o ChainReaction exibe uma melhoria de 177% em relação ao Cassandra-E e de 72% em relação ao FAWN-KV 3.

O teste localizado, baseado nos testes de carga personalizados com um único objecto, tem como objectivo mostrar que a nossa solução faz um melhor uso do recursos existentes na cadeia para melhorar o desempenho das operações de leitura, quando comparada com as restantes soluções. Se fosse possível aumentar o desempenho de uma forma linear, a nossa solução ao operar com 6 réplicas deveria obter um desempenho 6 vezes superior ao FAWN-KV num teste de carga só com leituras e um único objecto. No entanto, na prática isto não é possível devido à existência de atrasos na rede e de atrasos de processamento nas proxies. Desta forma, a melhoria é sublinear como se pode observar na Figura 1(b) obtendo-se um desempenho 4.3 vezes superior ao FAWN-KV. Também é possível observar que a nossa solução apresenta melhores resultados que a concorrência quando o número de operações de leitura aumenta. Por fim, a Figura 3 ilustra a distribuição cumulativa da latência das operações para alguns workloads.



(a) Leituras (Workload 50/50).

(b) Escritas (Workload 50/50).

(c) Leituras (Workload 5/95).

(d) Escritas (Workload 5/95).

Figura 3. Função de Distribuição Cumulativa (CDF) da Latência para os Testes de Carga Personalizados (é de notar que a latência se encontra numa escala logarítmica).

5 Trabalho Futuro

Como foi referido anteriormente, o ChainReaction foi pensado como um bloco básico para a construção de um sistema com suporte para vários centros de dados. Futuramente, pretende-se adaptar o ChainReaction para que possa operar num cenário de Geo-replicação. Também pretendemos oferecer uma operação semelhante às GET-TRANSACTIONS fornecidas pelo COPS, que permitem a um cliente obter uma visão coerente de um conjunto de chaves. Finalmente, pretende-se avaliar experimentalmente estas extensões ao ChainReaction e comparar o seu desempenho com o FAWN-KV, Cassandra e COPS.

6 Conclusão

Este artigo introduziu o ChainReaction, um sistema de armazenamento de dados distribuído que oferece alto desempenho, escalabilidade e alta disponibilidade. Este sistema pode ser instalado num único centro de dados, sendo compatível com aplicações construídas de acordo com o paradigma de computação em nuvem. O ChainReaction oferece um melhor desempenho que a concorrência em ambos os cenários descritos anteriormente, através do uso de um novo protocolo de replicação baseado na replicação em cadeia. Isto pode ser confirmado pelos resultados obtidos através da avaliação experimental realizada. O nosso sistema oferece o modelo de coerência *causal+*, recentemente introduzido, que

oferece garantias úteis para o programador. O ChainReaction apresenta-se como um bloco de construção adequado ao desenvolvimento de uma solução com suporte para Geo-replicação.

Referências

1. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. PODC '00, New York, NY, USA, ACM (2000) 7
2. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Operating Systems Review **44** (April 2010) 35–40
3. Andersen, D.G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., Vasudevan, V.: Fawn: a fast array of wimpy nodes. Commun. ACM **54**(7) (July 2011) 101–109
4. Hastorun, D., Jampani, M., Kakulapati, G., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: In Proc. SOSP. (2007) 205–220
5. 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: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. SOSP '11, New York, NY, USA, ACM (2011) 401–416
6. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. SoCC '10, New York, NY, USA, ACM (2010) 143–154
7. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16** (May 1998) 133–169
8. van Renesse, R., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6, Berkeley, CA, USA, USENIX Association (2004) 91–104
9. Robert Escriva, B.W., Sirer, E.G.: Hyperdex: A distributed, searchable key-value store for cloud computing. Technical report, Computer Science Department, Cornell University (December 2011)
10. Baker, J., Bond, C., Corbett, J., Furman, J.J., Khorlin, A., Larson, J., Leon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR. (2011) 223–234
11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21** (July 1978) 558–565
12. Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Proceedings of the sixteenth ACM symposium on operating systems principles. SOSP '97, New York, NY, USA, ACM (1997) 288–301
13. Belarami, N., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., Zheng, J.: PRACTI replication. NSDI '06 (May 2006) 59–72
14. Lesniewski-Laas, C.: A sybil-proof one-hop dht. In: Proceedings of the 1st Workshop on Social Network Systems. SocialNets '08, New York, NY, USA, ACM (2008) 19–24