Concretização Eficiente de Coerência Causal Transaccional na Nuvem

Taras Lykhenko e Luís Rodrigues

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
{taras.lykhenko,ler}@tecnico.ulisboa.pt

Resumo Os sistemas distribuídos de armazenamento chave-valor, que oferecem modelos de coerência fraca, emergiram como uma estratégia para aumentar o desempenho e a capacidade de escala dos sistemas que operam na nuvem. Contudo, a coerência fraca dificulta o desenvolvimento de aplicações, existindo enorme interesse em oferecer outros modelos de coerência que sejam úteis para os programadores sem comprometerem a capacidade de escala. O modelo de Coerência Causal Transacional (CCT) é particularmente relevante neste contexto. Neste artigo apresentamos o FastCCS, um novo algoritmo para suportar CCT em menos rondas de comunicação que os trabalhos anteriores. Resultados experimentais mostram que o FastCCS pode suportar um débito 30% superior ao oferecido pelos sistemas anteriores.

1 Introdução

Neste artigo consideramos um sistema em que as aplicações estão estruturadas em sequências de operações de leitura e escrita, que designamos por transações, as quais acedem a dados mantidos num sistema de armazenamento chave-valor. A execução concorrente destas transações, sem mecanismos adequados de controlo de concorrência, pode gerar resultados diferentes dos desejados, devido ao encadeamento de múltiplas operações de transações distintas. Modelos de coerência forte, como a serializabilidade, evitam este problema ao assegurar que o resultado da execução concorrente das transações é equivalente a uma execução em série dessas transações. Infelizmente, os mecanismos que permitem assegurar a serializabilidade ou são bloqueantes, ou levam as transações a abortar e reexecutar, limitando severamente o desempenho dos sistemas de armazenamento. É importante sublinhar que estas desvantagens são difíceis de evitar, mesmo que a fracção de transações que alteram os dados seja reduzida, como acontece tipicamente nas aplicações que se executam na nuvem [6]. Assim, as estratégias para obter bom desempenho passam geralmente por recorrer a modelos de coerência mais fracos e a técnicas que distinguem as transações de escritas das de leitura de forma a optimizar a execução destas últimas.

Devido aos problemas de desempenho inerentes aos sistemas transaccionais clássicos, os primeiros sistemas de armazenamento chave-valor para suportar a execução de aplicações na nuvem deram prioridade ao desempenho e capacidade de escala, suportando apenas modelos de coerência bastante fracos, como

por exemplo a coerência eventual. No entanto, a experiência mostrou que estes modelos dificultam bastante o desenvolvimento das aplicações [2], pelo que existe enorme interesse em encontrar novos modelos de coerência, e técnicas para suportar estes modelos, que sejam úteis para os programadores sem comprometerem a capacidade de escala. O modelo de *Coerência Causal Transacional* (CCT) é particularmente relevante neste contexto. Este modelo de coerência estende o modelo de *Coerência Causal*, originalmente definido para operações isoladas, permitindo que aplicações leiam múltiplos objectos de um corte causal e que executem escritas atómicas de múltiplos objectos. A relevância deste modelo advém do facto de ter sido demonstrado [4] que a coerência causal é o modelo mais forte de coerência que pode ser suportado sem comprometer a disponibilidade do sistema na presença de falhas ou de partições na rede.

É importante sublinhar que, se num sistema centralizado os efeitos da execução concorrente já tornam difícil oferecer garantias de coerência, a distribuição amplifica ainda mais esta complexidade. Em particular, se chaves diferentes são armazenadas em nós distintos, o risco de um cliente ler versões incoerentes é maior, pois é na prática impossível assegurar que os múltiplos efeitos de uma única transação fiquem visíveis no mesmo instante em todos os servidores. Por outro lado, o recurso à distribuição, e à oportunidade de particionar os dados e de armazenar diferentes partições em diferentes servidores é crucial para assegurar o desempenho e a capacidade de escala dos sistemas de armazenamento para a nuvem, uma vez que permite que diferentes pedidos sejam processados em paralelo por diferentes servidores.

Os sistemas que oferecem CCT recorrem a algoritmos não bloqueantes, que usam informação de controlo (metadados), que é escrita, lida e armazenada conjuntamente com os dados, para verificarem se a transação leu de um corte causal. Caso a transação não tenha feito um conjunto de leituras mutuamente coerente, a transação pode ser obrigada a ler novas versões dos dados (e isto pode ocorrer mais do que uma vez). O tamanho dos metadados mantidos pelo algoritmo tem um impacto significativo no desempenho do sistema [14]. Por um lado, quanto maior for o volume dos metadados, menos eficiente é o sistema, uma vez que estes podem consumir uma fracção não desprezável de recursos do sistema. Por outro lado, uma maior quantidade de metadados permite uma maior precisão na identificação do corte causal, e pode evitar rondas de leitura redundantes. Muitos sistemas optam por reduzir o tamanho dos metadados, criando o que designamos por *falsas dependências*, isto é, os metadados indicam que duas operações podem estar causalmente relacionadas, quando de facto não estão.

Neste artigo apresentamos o *Fast Causal Consistent Snapshot* (FastCCS), um novo algoritmo para suportar CCT. O FastCCS explora uma nova combinação entre o grau de concorrência que o sistema oferece, o tamanho dos metadados, e o número de passos de comunicação necessários para executar uma transação. Mais concretamente, enquanto os algoritmos anteriores para suportar CCT requerem que o sistema de armazenamento seja linearizável (o que limita o paralelismo do mesmo) ou, em alternativa, a execução de 3 rondas de comunicação para executar operações, o FastCCS oferece CCT em sistemas de armazenamento

particionados usando apenas 2 rondas de comunicação no pior caso. Para além disto, quando exposto a perfis de carga dominados por transações de leitura, o FastCCS executa a maioria das transações em apenas 1 ronda. Isto é conseguido recorrendo a metadados cujo tamanho é linear com o número de partições no sistema, mais precisamente, recorrendo a relógios vectoriais que possuem uma entrada para cada partição do sistema de armazenamento chave-valor.

Recorremos a simulações para comparar experimentalmente o desempenho do FastCCS com o desempenho de outros sistemas propostos na literatura. Os resultados mostram que o FastCCS pode suportar um débito 30% superior ao oferecido pelos sistemas anteriores.

2 Modelo do Sistema e Definições

Consideramos um sistema de armazenamento chave-valor em que os dados estão distribuídos por N partições. Assumimos também que cada partição tem um comportamento linearizável, mesmo que seja suportada por mais que um servidor (técnicas de replicação como a *replicação-em-cadeia* [15] permitem assegurar esta propriedade). Cada chave é deterministicamente atribuída a uma partição por uma função de dispersão. Denomina-se por p_x a partição que mantém a chave x. Este artigo foca no algoritmo que permite executar uma transação num único centro de dados. O algoritmo é compatível com extensões que permitem replicar o resultado da transação noutros centros de dados, mas essas técnicas são ortogonais à nossa contribuição e são aqui omitidas.

Assumimos que o armazenamento mantém múltiplas versões de cada chave, isto é, uma escrita numa dada chave cria uma nova versão mas mantém a versão anterior disponível. Isto permite que as transações possam ler de um corte coerente estável, podendo ignorar transações em curso cujos resultados não estejam ainda todos visíveis. Assume-se também a existência de um sistema de reciclagem de versões antigas, que deixaram de ser visíveis para os clientes.

Os clientes executam transações de leituras e escritas dos valores de uma ou mais chaves através de um *proxy* localizado dentro do centro de dados (por simplificação, referimo-nos ao *proxy* como cliente no resto do artigo). Uma transação que faça apenas leituras designa-se por *transação de leitura* e uma transação que escreva em pelo menos uma chave designa-se uma *transação de escrita*. Assumimos também que o cliente tem a capacidade de acumular todos os valores escritos na sua memória local e de os aplicar, em conjunto, quando a transação termina. O processo de confirmação de uma transação recorre a um algoritmo distribuído que envolve as partições modificadas.

Finalmente, o sistema mantém metadados que permitem verificar se as operações de leitura feitas pelos clientes retornam versões coerentes. Quando as chaves são escritas, são geradas novas versões às quais se associam metadados que capturam o corte causal em que a transação se executou. Quando uma transação lê uma chave obtém também os metadados associados. Estes metadados são usados para verificar se as versões lidas são mutuamente coerentes. Nos casos em que as versões lidas são incoerentes, o cliente contacta de novo (algumas) partições

para obter versões que lhe permitam obter um corte causal. O funcionamento pormenorizado dos algoritmos será descrito posteriormente.

2.1 Coerência Causal

Para duas operações a e b, diz-se que b causalmente depende de a, denotado por $a \rightsquigarrow b$, se e só se uma das três condições for satisfeita: Se a e b são duas operações executadas por um único fio de execução, então $a \rightsquigarrow b$ se a operação a é executada antes da operação b. Se a é uma operação de escrita e b é uma operação de leitura, e a operação b observa o efeito de a, então $a \rightsquigarrow b$. Para as operações a, b e c, se $a \rightsquigarrow b$ e $b \rightsquigarrow c$, então $a \rightsquigarrow c$.

Sejam $e_a(k_a)$ e $e_b(k_b)$ duas operações de escrita numa mesma chave ou em duas chaves distintas, k_a e k_b. Sejam $l_a(k_a)$ e $l_b(k_b)$ duas operações de leitura, feitas por um mesmo cliente, em que l_a é executado antes de l_b e em que l_a retorna o valor escrito por e_a e l_b retorna o valor escrito por e_b. Um sistema de armazenamento diz-se causalmente coerente se no caso em que $e_a \rightsquigarrow e_b$ não existe nenhuma escrita e'_b, na chave k_b, tal que $e_b \rightsquigarrow e'_b \rightsquigarrow e_a$.

2.2 Coerência Causal Transaccional

A coerência causal é definida para operações individuais, sem ter em conta o modo como estas estão relacionadas. Isto permite sequências de operações de leitura e escrita cujos resultados podem não ser os desejados pelos programadores. Considere-se por exemplo, duas transações de escrita $T^1 = \{e^1_a(k_a), e^1_b(k_b)\}$ e $T^2 = \{e^2_a(k_a), e^2_b(k_b)\}$ em que $T^1 \rightsquigarrow T^2$. Considere-se também duas transações de leitura $T^3 = \{l^3_b(k_b), l^3_a(k_a)\}$ e $T^4 = \{l^4_a(k_a), l^4_b(k_b)\}$.

A coerência causal garante que caso a transação T^3 leia o valor de k_b escrito por T^2 então, quando posteriormente ler o valor de k_a, lê também o valor escrito por T^2 (e não o valor anteriormente escrito por T^1). Esta garantia resulta do facto de $e^1_a(k_a) \rightsquigarrow e^1_b(k_b) \rightsquigarrow e^2_a(k_a) \rightsquigarrow e^2_b(k_b)$ ser independente do modo como estas operações se agrupam em transações.

No entanto, a coerência causal não impede que a transação T^3 leia o valor escrito por T^1 em k_b e posteriormente o valor escrito por T^2 em k_a. Também não impede que a transação T^4 leia o valor de k_a escrito por T^1 e posteriormente leia o valor de k_b escrito por T^2. Nenhuma destas sequências viola a coerência causal, embora possa induzir, em certos casos, comportamentos inesperados.

Antes de continuar, damos dois exemplos concretos que ilustram problemas que podem resultar das sequências acima referidas:

Exemplo 1: Considere-se uma aplicação do tipo rede social, onde as relações de amizade são simétricas e se pretende assegurar a seguinte invariante: se o utilizador u_1 aparece na lista de amigos de u_2, então u_2 deve também aparecer na lista de amigos de u_1. Considere-se que k_i guarda a lista de amigos do utilizador u_i e que a transação T^1 estabelece uma relação de amizade entre a e b, e a transação T^2 elimina esta relação. Neste caso, quer T^3 quer T^4 iriam ler um estado que violaria a invariante.

Tabela 1. Sistemas que oferecem garantias de coerência causal. R representa o número de rondas de leitura e V o número de rondas que retornam valores. NB e WTX representam, respetivamente, leituras não bloqueantes e transações de escrita. N representa o número de partições, M o numero de centro de dados e ts o valor de relógio físico.

Sistema	R	V	NB	WTX	Metadados	Estratégia
ChainReaction [3]	≥ 1	≥ 1	✗	✗	O(M)	Seriação
Orbe [7]	2	1	✗	✗	O(N x M)	Estabilização
GentleRain [8]	2	1	✗	✗	1 ts	Estabilização
COPS [10]	≤ 2	≤ 2	✓	✗	O(|deps|)	Verificação explícita
COPS-SNOW [12]	1	1	✓	✗	O(|deps|)	Verificação explícita
Cure [2]	2	1	✗	✓	O(M)	Estabilização
Eiger [11]	≤ 3	≤ 2	✓	✓	O(|deps|)	Verificação explícita
Wren [14]	2	1	✓	✓	2 ts	Estabilização
FastCCS	≤ 2	≤ 2	✓	✓	O(N)	Estabilização

Exemplo 2: Considere-se agora que a aplicação gere uma pasta partilhada, em que k_a regista quem tem acesso à pasta e k_b regista o conteúdo da pasta. Considere que a transação T_2 altera a lista de acesso à pasta de forma a excluir um determinado utilizador e posteriormente coloca na pasta um documento ao qual esse utilizador já não deve ter acesso. A sequência atrás descrita para a transação T^4 permitiria que a aplicação lesse a lista de controlo de acesso antiga mas o novo conteúdo da pasta, não conseguindo aplicar a restrição de acesso pretendida pelo utilizador.

A *Coerência Causal Transacional* (CCT) evita as anomalias acima descritas ao assegurar que todos os efeitos de uma transação de escrita são visíveis por outras transações ou nenhum é. Convém referir que existem vários sistemas que estendem a coerência causal com suporte para transações apenas de leitura [10,7,3,8]. Esses sistemas evitam a anomalia ilustrada por T^4 mas não a anomalia ilustrada por T^3. Desta forma, a CCT é mais forte que a coerência causal com suporte para transações de leitura que por sua vez é mais forte que a coerência causal sem suporte para transações. O leitor interessado pode encontrar uma comparação hierárquica dos vários modelos de coerência que têm sido propostos na literatura em [2] mas, sublinhamos que a CCT continua a ser mais fraca que o isolamento instantâneo (*snapshot isolation*) que por sua vez é mais fraco que a serializabilidade. De facto, estes dois últimos modelos de coerência obrigam a ordenar as escritas de forma total, o que não acontece com o CCT.

3 Trabalho Relacionado

A concretização de diferentes formas de transações em sistemas com baixa coerência tem sido muito estudada nos últimos anos, sendo possível encontrar diferentes abordagens ao problema na literatura. Em comum, todas estas soluções procuram oferecer baixa latência e evitam reduzir o débito do sistema. Na Tabela 1 apresentamos uma comparação entre os sistemas relacionados relevantes no contexto deste trabalho. Na última linha da mesma tabela apresentamos também as propriedades da nossa solução, que iremos descrever na Secção 4.

Podemos classificar as estratégias em três grandes categorias, conforme a técnica que usam para assegurar que a transação se executa num corte coerente, nomeadamente: corte coerente por estabilização, corte coerente por seriação e corte coerente por verificação explícita de dependências. De seguida, discutimos sumariamente cada uma destas técnicas.

A estratégia em que o corte coerente é obtido por estabilização consiste em ordenar de forma total as transações, com base numa estampilha temporal que é obtida recorrendo a relógios sincronizados. Uma partição pode satisfazer uma operação de leitura feita por uma transação que se executa no instante t quando sabe que já tomou conhecimento do efeito de todas as transações de escrita que são executadas com estampilhas inferiores a t. Infelizmente, uma vez que é impossível sincronizar os relógios com total precisão, os nós precisam trocar informação para saber quais as estampilhas que estão no passado de todos os outros nós. Por sua vez, os clientes necessitam também contactar pelo menos uma partição para preestabelecer a estampilha que devem usar, de forma a que a sua transação não seja seriada no passado. Exemplos de sistemas que usam esta técnica são o Orbe [7], GentleRain [8] e Cure [2]. O Wren [14] usa estratégias semelhantes embora recorrendo a relógios híbridos. Uma desvantagem destes sistemas é que transações de leitura necessitam sempre de duas rondas, uma vez que é necessária uma ronda só para identificar a estampilha de leitura [12]. Nos sistemas em que o corte coerente é obtido por seriação, é usado um componente centralizado, que toma conhecimento de todas as transações, e as ordena de forma total. Esta solução tem a desvantagem de criar um ponto de estrangulamento no sistema, o que limita a capacidade de escala do mesmo. Um dos sistemas que implementa esta estratégia é o ChainReaction [3].

Finalmente, os sistemas que usam a verificação explícita do corte coerente, obrigam a associar a todas as transações de escrita metadados que capturam o passado causal do cliente. Infelizmente, não é fácil garantir que um cliente consegue ler de um corte coerente numa única ronda e, de facto, sistemas como o Eiger [11] que usam esta estratégia usam no máximo três rondas ou como o COPS[10] e COPS-SNOW[12] que retornam em menos de duas ou uma respetivamente mas não suportam transações de escrita.

Na próxima secção apresentamos o FastCCS, um sistema que combina ideias da estratégia de estabilização e o uso de relógios lógicos para conseguir suportar transações de leitura com um reduzido número de rondas (2 no máximo, mas perto de 1 em média).

4 FastCCS

Em muitas aplicações para a nuvem, as operações de leitura dominam o sistema [12]. Por exemplo, 99.8% das operações da base de dados distribuída do Facebook [6] são leituras e a latência destas leituras é particularmente importante porque um pedido do cliente pode originar milhares de leituras e algumas destas leituras necessitam de ser feitas em sequência, podendo o caminho crítico chegar às dezenas de leituras [1]. Desta forma, é importante oferecer algorit-

mos que suportam leituras não bloqueantes e com o menor número de rondas de comunicação possível. Nesta secção apresentamos um algoritmo, denominado FastCCS, que permite oferecer Coerência Causal Transaccional com baixa latência e preservando a concorrência do sistema.

4.1 Metadados

O FastCCS associa a cada transação de leitura ou de escrita uma estampilha temporal, materializada por um relógio vectorial com N entradas, em que N é o número de partições do centro de dados. Sempre que o valor de uma chave é alterado por uma transação de escrita, é criada uma nova versão dessa chave que é marcada com a estampilha temporal definida ED para a escrita. Os clientes mantêm também a estampilha temporal da operação mais recente que fizeram, que é representado na forma de um vetor de tamanho N que designamos por E_c.

Quando uma nova versão de uma chave é criada esta passa por três estados a saber: *pendente*, *confirmada*, e *visível*. A estampilha temporal associada a uma versão no estado *pendente* é temporária, e atualizada no momento em que a transação passa para o estado *confirmado*. Uma chave passa para o estado *visível* quando todas as chaves modificadas na mesma transação, assim como todas as chaves modificadas pelas transações no passado da transação de escrita, já tiverem sido confirmadas em todas as partições.

Cada partição i mantém um número de sequência, que é incrementado sempre que é gerada uma nova versão de uma chave nessa partição. Designamos este número de sequência por V_i. Cada partição i mantém também uma estampilha temporal LE_i que corresponde a um vetor de tamanho N, em que cada entrada contém o valor máximo de V_j observado pela partição i, designada por *linha de estabilidade*. Todas as chaves modificadas por transações com estampilhas temporais menores ou iguais à linha de estabilidade estão garantidamente confirmadas em todas as partições. Cada partição atualiza o valor de $LE_i[i]$ sempre que uma transação é confirmada nessa partição. Para além disso, as partições trocam periodicamente o seu valor da variável $LE_i[i]$ de forma a atualizar o mesmo com a informação fornecidas pelas restantes partições. Sempre que a linha de estabilidade é atualizada, a partição verifica se existem versões no estado confirmado que possam passar para o estado visível, alterando o seu estado em conformidade.

4.2 Transações Somente de Escrita

Uma transação de escrita T é executada em duas rondas, da seguinte forma. Seja $\mathcal{P}(T)$ o conjunto de partições que armazenam chaves modificadas por T.

Numa primeira ronda, o cliente escolhe uma das partição $co \in \mathcal{P}(T)$ como coordenador da transação. O cliente envia para cada partição $\forall i \in \mathcal{P}(T)$ o novo valor das chaves modificadas e o identificador coordenador da transação. Para o coordenador, o cliente envia também o E_c e o número de participantes da transação e aguarda uma resposta do coordenador. Cada partição, ao receber esta mensagem, incrementa o seu número de sequência V_i e cria uma nova versão da

chave, no estado pendente, à qual atribui uma estampilha temporal temporária ET_i em que $ET_i[j] = -1, i \neq j$ e $ET_i[i] = V_i$. Se a partição for o coordenador da transação, a partição fica a espera da resposta das outras partições em $\mathcal{P}(T)$ caso contrario, O valor de V_i é retornado ao coordenador.

A segunda ronda começa quando o coordenador recebe uma resposta de todas as partições. O coordenador cria uma estampilha temporal definitiva para a transação ED em que $ED[i] = max(E_c[i], V_i), \forall i \in \mathcal{P}(T)$. O coordenador envia este valor para todas as partições em $\mathcal{P}(T)$. Ao receber o valor de ED, a partição passa todas as versões das chaves modificadas pela transação T para o estado confirmado, associando a estas versões o valor da estampilha temporal definitiva. Finalmente, cada partição espera até que $LE_i[i]-1 \geq ED[i]$, momento em que tem a certeza que todas as transações com estampilha inferior a ED já se encontram confirmadas na partição i. Quando esta condição se verifica, uma resposta é enviada ao coordenador. A transação de escrita é considerada terminada quando o coordenador recebe uma resposta de todas as partições, assegurando que todos os valores que a transação escreveu já estão confirmados em todas as partições. E de seguida o coordenador envia o ED da transação para o cliente. O cliente adopta ED como a sua nova estampilha ($E_c = ED$).

4.3 Transações Somente de Leitura

Uma transação de leitura T é executada no máximo em duas rondas, da seguinte forma. Seja $\mathcal{P}(T)$ o conjunto de partições que armazenam chaves lidas por T.

Numa primeira ronda, o cliente envia para cada partição $i \in \mathcal{P}(T)$ as chaves que pretende ler, conjuntamente com a estampilha temporal do cliente E_c e aguarda uma resposta de todas as partições em $\mathcal{P}(T)$. Cada partição, ao receber esta mensagem, atualiza a sua linha de estabilidade fazendo $LE_i[k] = max(LE_i[k], E_c[k]), \forall k \in N$. Isto é possível dado que E_c representa a estampilha máxima que o cliente já viu, e se os efeitos de uma transação já estão visíveis quer dizer que a transação já foi confirmada em todas as partições e tornando assim possível avançar LE até ao E_c em segurança. Isto garante que todos os efeitos de transações que o cliente já observou no passado vão ser visíveis, garantindo assim a atomicidade e a causalidade.

As partições retornam, as versões mais recentes com o $ED \leq LE$, o LE e a estampilha máxima lida EM, onde $EM[k] = max(EM[k], ED[k]), \forall k \in N$.

Quando o cliente recebe uma resposta de todas as partições. O cliente verifica se $LE_i \geq EM_j, \forall i, j \in \mathcal{P}(T) \wedge i \neq j$, o cliente atualiza o seu $E_c = max(EM_i[k], E_c[k]), \forall i \in \mathcal{P}(T) \wedge \forall k \in N$. Se a condição se verificar, o cliente termina a transação e retorna os valores. Se a condição não se verificar, o cliente leu de um corte que pode ou não ser coerente. Assim sendo, o cliente tem de iniciar uma segunda ronda de leituras as partições que não satisfizeram a condição anterior, enviando o seu novo E_c. As partições ao receberem um pedido da segunda ronda de leitura, retornam as versões mais recentes com o $ED \leq E_c$, e atualizam a sua linha de estabilidade à semelhança da primeira ronda.

A segunda ronda garante que vai ser devolvido um corte causal coerente, isto porque, o protocolo de estabilização garante que todas as versões que o cliente

leu na primeira ronda já estão instaladas em todas as partições, não sendo assim necessário bloquear a leitura até a nova versão ser instalada. Como o cliente lê versões que satisfazem $ED \leq E_c$, não vão ser necessárias rondas extra de comunicação porque as leituras devolvidas na primeira ronda têm um $ED \leq E_c$.

4.4 Transações de Leitura e Escrita

No FastCCS as transacções de leitura e escrita são logicamente estruturadas em duas fases, uma fase de leitura seguida de uma fase de escrita, que se executam em sequência usando os algoritmos atrás descritos.

4.5 Progresso da Linha de Estabilidade

Para as versões criadas por transações de escrita tornarem-se visíveis, é necessário que a sua estampilha seja igual ou inferior ao valor da linha de estabilidade. É pois fundamental ter um algoritmo para garantir a progressão do valor desta variável. O FastCCS, implementa o seguinte algoritmo.

Cada partição é responsável por atualizar localmente o seu valor correspondente à sua posição, na sua linha de estabilidade LE_i. Seja T a transação que modificou uma ou mais chaves mantidas pela partição i e cujas versões foi atribuído um número de sequência com o valor igual a $LE_i[i] + 1$. Quando a transação T passar do estado pendente para o estado confirmado na partição i, esta partição incrementa o valor de $LE_i[i]$.

Periodicamente, cada partição envia o valor de $LE_i[i]$ para todas as outras partições. Quando a partição i recebe o valor $LE_j[j]$ da partição j, i atualiza a sua linha de estabilidade fazendo $LE_i[j] = max(LE_i[j], LE_j[j])$.

A informação fornecida pelos clientes, quando apresentam a sua estampilha temporal, é também usada para acelerar a progressão da linha de estabilidade. Isto é possível, uma vez que o cliente só observa versões de transações que já estão confirmadas em todos as partições.

5 Avaliação

Nesta secção avaliamos o desempenho FastCCS em comparação com dois sistemas anteriores que oferecem garantias semelhantes, nomeadamente comparamos com o Eiger [11] e o Wren [14]. A avaliação pretende aferir o débito total do sistema e o número médio de rondas (a latência das operações é proporcional a este número) executado por cada um dos algoritmos. Antes de apresentarmos os resultados, descrevemos a nossa bancada experimental.

5.1 Bancada Experimental

A nossa avaliação é baseada em simulações de execuções do FastCCS, do Eiger [11] e do Wren [14] usando o PeerSim [13], configurado com extensões que

Tabela 2. Parâmetros usados nas experiências.

Parâmetro	Padrão	Variação
Chaves / Transação	4	2-32
Partições	25	-
Chaves	10000	-
Clientes	10000	2500-20000
Fração de escritas	0.05	0.01-0.5

simulam a latência na rede e a largura de banda finita. Os canais entre dois pontos asseguram que a primeira mensagem a ser enviada é a primeira mensagem a ser entregue. Uma vez que o desempenho dos diversos algoritmos depende fundamentalmente do número de rondas e da necessidade de esperar pela obtenção de um corte causal, o simulador não captura a utilização de CPU nem o tempo perdido no acesso ao disco. Cada nó possui um limite de largura de banda de 1 Gb/s e cada mensagem sofre um atraso aleatório cujo valor médio é de 0.5 ms (latência observada entre servidores num centro de dados da AWS [5]).

Os clientes estão configurados para executar pedidos ao sistema em ciclo fechado, esperando pela resposta do pedido anterior antes de fazer um novo pedido. Quando um cliente faz um pedido, escolhe com uma determinada probabilidade se este corresponde a uma transação de escrita ou de leitura (este rácio varia conforme as experiências) e escolhe aleatoriamente as chaves acedidas pela transação (o número de chaves acedidas também varia com a experiência). Os parâmetros utilizados para as experiências encontram-se resumidos na Tabela 2.

5.2 Rondas de Leitura

O número total de rondas necessárias para concluir uma transação de leitura tem um impacto na latência e no débito do sistema. Como se pode observar nas figuras 1a a 1c , o número de rondas necessárias para concluir as transações de leitura do FastCCS é, em média, 33% inferior ao do Eiger. Esta vantagem advém dos metadados usados no FastCCS, que permitem evitar rondas de leitura desnecessárias. O Wren, por outro lado, utiliza sempre 2 rondas para leituras. De seguida, avaliamos o efeito destas diferenças no débito observado.

5.3 Débito do Sistema

Medimos o débito avaliando o número de transações por segundo que cada cliente consegue executar. Como os clientes funcionam em ciclo fechado, quanto menor for a latência de cada transação, mais depressa este pode fazer um novo pedido, e maior será o débito de cada cliente. Esperava-se pois que o FastCCS apresentasse um maior débito, o que se confirma experimentalmente, como se pode verificar pelos resultados apresentados nas figuras 1d a 1f. Avaliamos também o comportamento dos vários sistemas quando se varia o número de clientes, a percentagem de escritas e o tamanho das transações.

Fizemos variar o número de clientes sem nunca saturar a rede. Se as transações fossem independentes, o desempenho de cada cliente manter-se-ia estável.

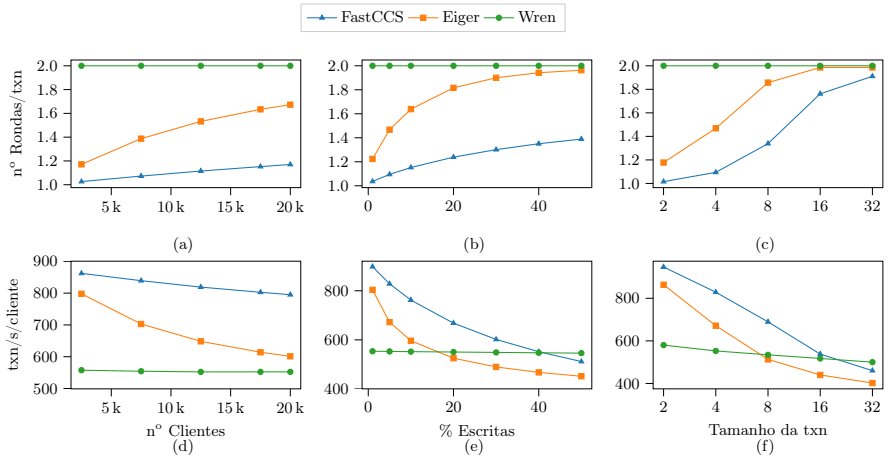

Figura 1. Número de Rondas e Débito

No entanto, ao aumentar o número de clientes (Figura 1d), aumenta a probabilidade das transações entrarem em conflito e, em consequência, o número de rondas das mesmas. Observamos que a penalização do Eiger de usar um relógio lógico por centro de dados, aumenta a incerteza, aumentando o número de rondas e subsequentemente a latência. Este efeito é menos notório no FastCCS, que tem uma maior precisão que o Eiger por utilizar mais metadados o que faz com que tenha o melhor débito e a latência mais baixa, comparativamente aos outros dois sistemas, oferecendo um débito 30% superior ao Eiger e aproximadamente 45% superior ao débito oferecido pelo Wren.

Quando aumentamos a percentagem de escritas (Figura 1e), o FastCCS obtém um débito superior aos outros dois sistemas, mesmo quando a carga é simétrica, onde o débito do FastCCS se aproxima ao do Wren. Conseguimos assim observar que a abordagem do Wren, ao recorrer a duas rondas de leituras, limita o débito do sistema e aumenta a latência dos pedidos.

Quando aumentamos o número de operações que cada transação executa (Figura 1f), observamos um decréscimo mais significativo do desempenho no FastCCS e no Eiger. Esta descida deve-se ao facto destes dois sistemas terem duas rondas de leituras que retornam valores, o que faz aumentar a latência e subsequentemente diminuir o débito. O Wren, ao retornar apenas valores na segunda ronda, é menos sensível a este parâmetro.

6 Conclusão e Trabalho Futuro

Neste artigo apresentamos um novo algoritmo FastCCS que oferece CCT em sistemas de armazenamento particionados usando apenas 2 rondas de comunicação no pior caso. Desenvolvemos um protótipo para aferir o desempenho do

sistema e demonstramos que, quando exposto a perfis de carga dominados por transações de leitura, que executa a maioria destas em apenas 1 ronda. A solução aqui proposta oferece um débito de operações médio 20% mais alto que o Eiger. No futuro, pretendemos implementar e avaliar o FastCCS no Cassandra [9] com suporte para replicação.

Agradecimentos: Este trabalho foi suportado pela FCT – Fundação para a Ciência e a Tecnologia, através dos projectos UID/CEC/50021/2019 e COSMOS (financiado pelo OE com a ref. PTDC/EEI-COM/29271/2017 e pelo Programa Operacional Regional de Lisboa na sua componente FEDER com a ref. Lisboa-01-0145-FEDER-029271).

Referências

1. Ajoux, P., Bronson, N., Kumar, S., Lloyd, W., Veeraraghavan, K.: Challenges to adopting stronger consistency at scale. In: HOTOS. Kartause Ittingen, Switzerland (May 2015)
2. Akkoorath, D., Tomsic, A., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: ICDCS. Nara, Japan (Jun 2016)
3. Almeida, S., Leitão, J.a., Rodrigues, L.: Chainreaction: A causal+ consistent datastore based on chain replication. In: Eurosys. Prague, Czech Republic (Apr 2013)
4. Attiya, H., Ellen, F., Morrison, A.: Limitations of highly-available eventually-consistent data stores. In: PODC. San Sebastián, Spain (Jul 2015)
5. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions: Virtues and limitations. In: VLDB. Trento, Italy (Aug 2013)
6. Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y., Venkataramani, V.: TAO: Facebook's distributed data store for the social graph. In: ATC. San Jose (CA), USA (Jun 2013)
7. Du, J., Elnikety, S., Roy, A., Zwaenepoel, W.: Orbe: Scalable causal consistency using dependency matrices and physical clocks. In: SOCC. San Jose(CA), USA (Oct 2013)
8. Du, J., Iorgulescu, C., Roy, A., Zwaenepoel, W.: Gentlerain: Cheap and scalable causal consistency with physical clocks. In: SOCC. Seattle (WA), USA (Nov 2014)
9. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. In: LADIS. Big Sky (MT), USA (Oct 2009)
10. 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: SOSP. Cascais, Portugal (Oct 2011)
11. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: OSDI. Lombard (IL), USA (Apr 2013)
12. Lu, H., Hodsdon, C., Ngo, K., Mu, S., Lloyd, W.: The SNOW theorem and latency-optimal read-only transactions. In: OSDI. Savannah (GA), USA (Nov 2016)
13. Montresor, A., Jelasity, M.: PeerSim: A scalable P2P simulator. In: P2P. Seattle (WA), USA (Sep 2009)
14. Spirovska, K., Didona, D., Zwaenepoel, W.: Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In: DSN. Luxembourg City, Luxembourg (Jun 2018)
15. Van Renesse, R., Schneider, F.: Chain replication for supporting high throughput and availability. In: OSDI. San Francisco (CA), USA (Dec 2004)