Consistência Causal em Sistemas Geo-Distribuídos com Replicação Parcial*

Pedro Fouto, João Leitão, and Nuno Preguiça

NOVA LINCS & DI/FCT/NOVA University of Lisbon, Lisboa, Portugal

Resumo Os sistemas de armazenamento de dados distribuídos são um componente fundamental em serviços de Internet de grande escala. De forma a acompanhar as expectativas dos utilizadores em relação à disponibilidade e latência desses serviços, o desenho destes sistemas tem evoluído de forma a conseguir providenciar estas características utilizando técnicas como a replicação parcial, geo-replicação e modelos de consistência fraca. O grande desafio é combinar todas estas técnicas numa solução integrada que seja prática e eficiente.
Neste artigo, propomos um novo esquema de replicação que oferece as garantias do modelo de consistência causal+ num cenário geo-distribuído com replicação parcial, onde os centros de dados replicam apenas fracções de todos os dados. A nossa solução baseia-se numa metodologia proposta recentemente que dissocia a propagação de dados da propagação de meta-dados, e apresenta três contribuições principais. Primeiramente, e ao contrário do estado da arte, o nosso esquema de propagação de informação para garantir causalidade utiliza um modelo de comunicação directa, evitando a complexidade e fragilidade de outras soluções, nomeadamente baseadas em árvores entre os centros de dados. Adicionalmente, a nossa solução permite um maior paralelismo no processamento de operações remotas, aumentando o desempenho do sistema. Por fim, apresentamos uma implementação que integra a nossa solução com o sistema Cassandra. Resultados experimentais obtidos através da execução da nossa implementação e de outras soluções em 9 centros de dados mostram que a nossa solução consegues ter tempos de visibilidade inferiores, aumentando o desempenho das operações remotas.

1 Introdução

As soluções de armazenamento de dados distribuídas [9,16,2,23,22,14,11] tornaram-se componentes chave de muitos serviços de grande escala na Internet. O desempenho destes sistemas afecta significativamente o desempenho dos serviços que os usam, como tal, muitos destes sistemas favorecem a utilização de modelos de consistência fraca [9,16,2,14,11], tipicamente consistência eventual. Juntamente com a geo-replicação, estes modelos possibilitam tempos de resposta pequenos.

A utilização destes modelos, no entanto, tem como desvantagem obrigar os programadores das aplicações a lidar com anomalias na consistência dos dados.

* Trabalho parcialmente financiado pelo projeto europeu LightKone (H2020 grant agreement ID 732505) e NOVA LINCS (FC&T UID/CEC/04516/2013)

Isto levou à exploração de soluções que oferecem consistência causal [18,19,2,13,1,6], visto este modelo proporcionar garantias de consistência mais fortes, mantendo a disponibilidade característica dos modelos de consistência fracos [3,20].

Infelizmente, a maior parte das soluções existentes que oferecem consistência causal+ num ambiente geo-replicado não suportam replicação parcial. O aumento do número de centros de dados e o crescente interesse em aplicações IoT tem aumentado o interesse em procurar soluções com estas características, de forma a reduzir os custos de armazenamento e comunicação em relação às soluções com replicação total.

Existem também desafios relativos à implementação de sistemas com estas caracteristicas, o que levou ao aparecimento de multiplas técnicas. Existe um *trade-off* observável entre a recência dos dados e o débito[5,12] relacionado com os mecanismos usados para codificar (e disseminar) dependências causais. As soluções que utilizam poucos meta-dados[6,11] geralmente perdem precisão na determinação de dependências, piorando a recência de dados, enquanto que soluções que utilizam muitos meta-dados [1,10] sacrificam o débito.

Neste artigo propomos um sistema geo-replicado que suporta replicação parcial e que encontra o balanço no *trade-off* anteriormente descrito, partindo de uma proposta recente feita pelos autores de [6] que consiste em separar a propagação das dependências causais da propagação das operações/dados.

Apresentamos três contribuições principais em relação ao estado da arte: *(i)* utilizamos de um esquema de comunicação directa, simplificando a propagação de meta-dados entre os centros de dados, e evitando as complexidades e fragilidades das soluções que utilizam esquemas mais complexos; *(ii)* ajustámos a quantidade de meta-dados utilizados que, em conjunto com um novo modelo de execução, permite maximizar o paralelismo na execução de operações remotas; *(iii)* apresentamos uma implementação concreta da nossa solução sobre o sistema de armazenamento Cassandra [16], e explicamos os desafios inerentes a este tipo de implementação.

Resultados experimentais, extraídos num ambiente de *cloud* com 9 centros de dados espalhados pelo mundo, revelam que a nossa solução consegue ter um desempenho melhor que o estado da arte, mostrando também um custo adicional aceitável quando comparada com o sistema base com consistência eventual.

2 Trabalho Relacionado

Consistência causal em sistemas de armazenamento na *cloud*. Na ultima década apareceu um grande número de sistemas de armazenamento de dados com geo-replicação, que exploram os *trade-offs* entre a consistência e a disponibilidade. Alguns utilizam modelos de consistência mais fortes [8,21], sacrificando latência e disponibilidade na presença de falhas. Outros adoptam modelos de consistência mais fracos [9,16,18,2], conseguindo melhor latência e disponibilidade mas permitindo que o estado das réplicas divirja ocasionalmente. Nos sistemas com consistência eventual [9,16] apenas é garantido que as réplicas eventualmente irão convergir. Na consistência causal+, além da convergência, também existe a garantia que uma operação apenas se torna visível após as operações da qual depende.

Consistência causal com replicação total. Lamport [17] introduziu a relação de *happens-before* entre eventos num sistema distribuído, esta relação é a base da consistência causal. ISIS [4] e *lazy replication* [15] proporcionam consistência causal, utilizando vectores de versão com uma entrada por cada réplica para monitorizar as dependências. Nas infra-estruturas modernas de *cloud*, compostas por milhares de máquinas em diversos centros de dados, monitorizar a causalidade utilizando uma entrada por máquina não é prático. COPS [18] foi o primeiro sistema a introduzir o conceito de consistência causal+, que combina o modelo causal com a convergência oferecida pela consistência eventual [24]. Este sistema assume *linearizability* dentro de cada centro de dados e monitoriza a causalidade guardando, para cada escrita, a versão do ultimo objecto escrito e todos os objectos lidos desde a ultima escrita. Eiger [19] é uma evolução do COPS que também suporta transacções *read-only* e *write-only*. ChainReaction [2] é um sistema *key-value* geo-distribuído, baseado numa variante do *chain-replication* que garante consistência causal+. O sistema Cure [1] garante consistência causal conseguindo suportar transacções genéricas. Utiliza um vector de versão com uma entrada por centro de dados e um mecanismo de estabilização. O GentleRain [11] utiliza um único escalar, sendo mais eficiente que o Cure, mas sacrificando velocidade de execução das operações remotas. O sistema Eunomia [13] utiliza uma técnica de estabilização em cada centro de dados, combinada com relógios híbridos. Baseando-se nestes relógios, o Eunomia calcula uma ordem total de operações que garante causalidade.

Consistência causal com replicação parcial. O Saturn[6] é o sistema mais parecido com a nossa solução. É um serviço de monitorização de meta-dados que propaga os identificadores de cada operação utilizando uma árvore global de propagação. Ao propagar os identificadores por ordem *FIFO*, estes chegam a cada centro de dados repeitando ordem causal. O Saturn utiliza um algoritmo de optimização para calcular a árvore óptima. Quando um nó falha, é necessário computar uma nova árvore, o que é um processo lento. Além disto, a árvore (ou alguns ramos) podem tornar-se um *bottleneck* para o sistema, visto todos os identificadores passarem por ela. O nosso trabalho lida com estas limitações, adoptando uma abordagem alternativa onde a informação é propagada directamente entre todos os pares de nós. Adicionalmente, propomos um algoritmo que optimiza a concorrência na execução de operações remotas, o que tem que ser feito de forma seriada no Saturn.

3 Desenho

A nossa solução foi desenhada para enriquecer um sistema de armazenamento de dados existente, assumindo um conjunto de propriedades desse sistema:

Consistência: Assumimos que o sistema base oferece consistência eventual entre centros de dados. Dentro de um centro de dados, após uma escrita completar, todas as leituras retornam o valor escrito. Antes da escrita completar, tanto o novo valor como o antigo podem ser retornados, sendo que o mesmo utilizador não pode voltar a ver o antigo após ver o novo. Este comportamento é comum em sistemas de baseados em quórum, os quais retornam a escrita mais recente.

Replicação parcial e geo-replicação: Assumimos que o sistema base oferece replicação parcial e geo-replicação, e que cada centro de dados tem informação sobre os centros de dados que replicam cada objecto.
Sharding: Por fim, assumimos que os dados estão particionados em vários nós em cada centro de dados. Apesar de não ser um requerimento para a validade da solução, é importante pois permite maior paralelismo na execução de operações.

3.1 Arquitectura

O nosso desenho divide o sistema em duas camadas: causalidade e armazenamento. Os utilizadores interagem com a camada de armazenamento, que é responsável por executar as operações localmente e as propagar para centros de dados remotos. Antes de executar uma operação, existe coordenação com a camada da causalidade para evitar violações do modelo de consistência. A camada de causalidade é responsável por propagar a informação de monitorização de causalidade de cada operação, a esta informação chamamos *etiquetas*.

Esta separação de camadas traz vários benefícios: *(i)* é possível isolar a lógica de causalidade, minimizando as alterações necessárias ao sistema de armazenamento base. *(ii)* apenas é necessário guardar e processar informação com tamanho reduzido na camada de causalidade, aumentando a sua eficiência.

Na nossa solução, cada centro de dados contém uma instância da camada de causalidade, responsável por gerir a informação de causalidade desse local e por propagar essa informação para locais remotos. Cada uma destas instâncias comunica directamente com todas as outras. Em relação ao estado da arte, a nossa solução diferencia-se por não utilizar estruturas complexas na interconexão dos centros de dados, melhorando a tolerância a falhas ao evitar que cada *etiqueta* tenha que ser propagada por (vários) nós intermédios.

3.2 API

O desenho do nosso sistema lida com três tipos de operações: *(i)* operações de *leitura*; *(ii)* operações de *escrita*; *(iii)* operações de *migração*, que permitem ao cliente mudar o centro de dados no qual estão a executar operações. As operações de leitura e escrita utilizadas são as fornecidas pelo sistema base.

Em cada momento, cada cliente tem um centro de dados local, e apenas pode executar operações nesse centro de dados. Para poder aceder a dados que não estejam nesse centro de dados, é necessário executar uma operação de migração.

3.3 Garantias de Causalidade

O objectivo da camada de causalidade é manter informação necessária para garantir que qualquer operação é executada respeitando a consistência causal. Visto as operações de leitura serem mais frequentes do que as de escrita, estas são favorecidas na nossa solução, sendo executadas directamente na camada de armazenamento, sem coordenação com a camada de causalidade.

No nosso modelo de sistema enquanto uma escrita está em progresso, o valor retornado por uma leitura pode ser tanto o novo como o antigo. De forma a garantir consistência causal, necessitamos de garantir que: *(i)* para uma escrita, $w(o)$, após um cliente executar uma leitura que retorna $w(o)$, esse cliente não pode voltar a ler um valor mais antigo *(ii)* para duas escritas, $w_1(o_1)$ e $w_2(0_2)$, onde w_1 precede causalmente w_2, se um utilizador lê o resultado da escrita w_2, todas as leituras seguintes a o_1 devem retornar uma versão que reflicta w_1.

A primeira propriedade é satisfeita garantindo que um cliente lê cada objecto sempre do mesmo quórum de réplicas. Para a segunda propriedade é necessário manter informação sobre dependências das operações e garantir que operações dependentes são executadas pela ordem correcta. Esta estratégia é implementada utilizando a camada de causalidade. Esta camada mantêm: um *contador de operações*, utilizado para atribuir *timestamps* às operações; um *relógio de operações concluídas (RoC)*, que consiste num vector com uma entrada por cada centro de dados que corresponde ao *timestamp* da operação mais recente executada proveniente desse centro de dados; um *relógio de operações em execução (RoE)*, que tem o mesmo formato que o *RoC*, mas guarda os *timestamps* das operações mais recentes em execução.

Escritas. Quando a aplicação executa uma operação de escrita, essa operação é enviada para a camada de armazenamento, a qual remete a operação para a instância da camada de causalidade local e propaga a operação para os nós responsáveis por a executar. A operação não é imediatamente executada em nenhum nó. Quando a camada de causalidade recebe informação sobre uma operação de escrita local, incluindo o seu identificador único e o conjunto de centros de dados onde será executada, o *contador de operações* local é aumentado e utilizado para atribuir um *timestamp* a essa operação. As dependências dessa operação são também atribuídas utilizando o valor actual do *RoE*. O conjunto desta informação forma a *etiqueta* da operação. O *RoE* é necessário visto não ser possível determinar se um utilizador já leu o resultado de operações em execução. Como tal, a única abordagem segura é assumir que uma escrita depende de todas as outras que estão em execução. De seguida, a camada de causalidade coloca a *etiqueta* num registo de operações pendentes e propaga-a para as instâncias da camada de causalidade dos outros centros de dados relevantes. Uma operação é executada assim que as suas dependências sejam satisfeitas, ou seja, quando todas as entradas do *RoC* local sejam maiores ou iguais às entradas do relógio de dependências da *etiqueta*. Quando isto acontece, a camada de causalidade notifica os nós locais responsáveis por executar a operação. Esses nós confirmam essa execução, quando terminada, de volta à camada de causalidade.

O conjunto de todas estas abordagens garante que uma operação apenas é executada após execução de todas as operações da qual depende.

Migrações. Quando é necessário aceder a um objecto que não está replicado no centro de dados local de um cliente, é necessário que a operação de mudança de centro de dados só termine após o novo centro de dados tenha executado todas as operações observadas/executadas pelo cliente. Temos que assumir que o cliente já observou uma versão da base de dados que reflicta todas as operações em execução. Na nossa solução, recorremos a uma operação de migração, que permite a um cliente alterar o seu centro de dados local. A execução desta operação é semelhante à execução de uma escrita, com a diferença de que não necessita de *timestamp* e é propagada apenas para um centro de dados.

3.4 Discussão

A nossa solução utiliza uma abordagem cuidadosa na atribuição das dependências causais às operações, no entanto, consegue alcançar um elevado grau de con-

corrência. Quando comparada com soluções que monitorizam dependências causais com mais precisão, como o COPS[18], a nossa abordagem tem a vantagem de não necessitar de informação extra guardada no sistema de armazenamento. Quando comparado com outras soluções que utilizam a abordagem da separação de camadas, como o Saturn[6] e o Eunomia[13], o nosso sistema consegue operar com um modelo de consistência mais fraco dentro de cada centro de dados, oferece maior concorrência na execução de operações, não necessita de informação adicional na camada de armazenamento, e evita guardar dependências no cliente. Tanto o Saturn como o Eunomia assumem que o sistema de armazenamento base implementa *linearizability* dentro de um centro de dados. Apesar de poder ser implementada eficientemente, os sistemas actuais que utilizam replicação baseada em quórum frequentemente adoptam modelos mais fracos (e mais rápidos). Em relação à concorrência, a propagação de dependências mais precisas na nossa solução permite executar um número elevado de operações remotas em paralelo. Em oposição, no Saturn/Eunomia a serialização das operações obriga a execução serial de operações remotas.

4 Integração com o Cassandra

Nesta secção explicamos como integrámos a nossa solução, C^3 com o Cassandra. Resumidamente, necessitamos do seguinte: *(i)* garantir que o Cassandra se enquadra ao nosso modelo de sistema; *(ii)* modificar a execução de operações de forma a integrar no nosso esquema de replicação.

Para garantir o primeiro requerimento, precisamos de garantir que após uma escrita concluir num centro de dados, todas as leituras subsequentes observam essa escrita. Isto é alcançado escolhendo o nível de consistência do Cassandra *LOCAL_QUORUM*. Após uma escrita concluir, pelas propriedades do quórum local, é garantido que qualquer quórum local de leitura a irá interceptar.

Para integrar o nosso esquema de replicação, é necessário fazer alterações à execução das escritas. No Cassandra, uma escrita começa com o cliente a enviar a operação para um nó denominado coordenador, o qual propaga a operação para os nós relevantes e aguarda confirmação de um subconjunto desses nós antes de responder ao cliente. Nesta lógica de execução fazemos as seguintes modificações: *(i)* o coordenador envia também a informação da operação para a camada de causalidade; *(ii)* os nós locais esperarem pela *etiqueta* antes de executarem a operação e confirmam a execuação à camada de causalidade; *(iii)* o mecanismo de *read repair* apenas opera dentro de cada centro de dados.

5 Avaliação

Nesta secção discutimos a nossa implementação do Saturn, seguida da descrição do ambiente de avaliação e dos resultados obtidos.

5.1 Saturn

De forma a podermos comparar a nossa solução com o estado da arte, implementámos uma versão do Saturn sobre o Cassandra.

Seguindo o desenho apresentado em [6], utilizamos o coordenador do Cassandra como *frontend*, visto já ser responsável por mediar o acesso dos clientes ao sistema. Como *gears*, utilizamos os nós do Cassandra, mudando o seu comportamento para gerar e propagar para a camada de causalidade uma *etiqueta*

por cada operação de escrita recebida. As *etiquetas* geradas são guardadas na base de dados com o objecto escrito. Os componentes *label sink* e *remote proxy* do Saturn, que correspondem à camada de causalidade do Saturn, foram implementados adaptando a nossa camada de causalidade.

Estas alterações foram feitas tentando respeitar estritamente a lógica descrita em [6]. Notamos que a nossa implementação do Saturn não utiliza *linearizability*, o que resulta numa implementação *mais eficiente*, tendo como desvantagem a não garantia de causalidade em todas as situações (*correção*).

5.2 Avaliação Experimental

Ambiente. Nas nossas experiências, utilizamos a plataforma de *cloud* Microsoft Azure. Utilizamos 9 centros de dados espalhados pelo mundo[1]. Visto também querermos estudar os efeitos de *sharding*, criámos 4 máquinas virtuais em cada centro de dados, com cada uma a executar uma instância do Cassandra. As máquinas utilizadas são do tipo *A2 v2* com 2 CPUs e 4GB de RAM. De forma a modelar a replicação parcial, criámos 9 partições de dados, cada uma associada a um centro de dados e replicada entre 2 e 4 centros de dados extra. Dentro de um centro de dados, cada partição é replicada em 3 dos 4 nós.

YCSB. Nas nossas experiências utilizamos o *Yahoo! Cloud System Benchmark*[7] para emular múltiplos clientes a executar operações no sistema. Os clientes podem operar em dois modos: *(i) locais*, onde só acedem a objectos do centro de dados primário (o mais perto); *(ii) remoto*, onde alternam entre o centro de dados primário e outro aleatório (que vai variando). Para o modo *remoto* adaptámos o YCSB para suportar a operação de migração e gerir quando um cliente deve migrar e quantas operações deve executar em cada centro de dados, sendo que estas alterações apenas são relevantes na nossa solução ou no Saturn. Também adaptamos o YCSB para emular a biblioteca de cliente do Saturn.

Paramêtros Experimentais. De forma a estudar a forma como a nossa solução se compara com outras, executámos experiências utilizando 4 termos de comparação, 3 delas correspondendo a configurações do Cassandra (garantindo consistência eventual) e a ultima correspondendo à nossa implementação do Saturn:
Cassandra - *E-LQ*: Esta configuração utiliza o sistema Cassandra, onde os clientes executam operações utilizando o nível de consistência *LOCAL_QUORUM*. Quando um cliente precisa de executar operações sobre dados em centros de dados remotos utiliza o nível de consistência *TWO*, resultando na maior parte dos casos num quórum de nós no centro de dados remoto mais próximo. Esta configuração é interessante pois permite perceber como seria o desempenho do nosso sistema caso o custo adicional de adicionar causalidade fosse zero.
Cassandra - *E-Q*: Nesta configuração os clientes utilizam o nível de consistência *QUORUM*, no qual necessitam de esperar por uma maioria do número *total* de nós que replicam o objecto a ser acedido.
Cassandra - *E-EQ*: Nesta configuração os clientes utilizam o nível de consistência *EACH_QUORUM* para escritas, tendo que esperar pela resposta de

[1] As localizações destes centros de dados são as seguintes: EUA Oeste, EUA Este, Japão, Austrália, Sudoeste Asiático, Índia, Canadá, Europa e Brasil

um quórum (2 nós) em *cada* centro de dados que replique o objecto. Os níveis de consistência *LOCAL_QUORUM* e *TWO* são utilizados nas leituras.

Nas configurações E-Q e E-EQ, os quórums de escrita interceptam sempre os quórums de leituras. Aproximando-se das garantias de consistência causal.

Saturn - *C-SAT*: Esta configuração utiliza a nossa implementação do Saturn.

Quando utilizamos uma das soluções que proporcionam causalidade (a nossa ou o Saturn), uma das máquinas de cada centro de dados necessita de executar uma instância da camada de causalidade. Para evitar que essa máquina fique com demasiada carga, diminuímos o seu número de *tokens* do Cassandra.

Em relação à configuração dos clientes, criámos 4 máquinas virtuais extra em cada centro de dados. Em cada uma destas máquinas executamos uma instância do YCSB com um número de clientes que varia entre 50 e 350 (no total, entre 1.800 e 12.600). O número de operações executadas em cada instância do YCSB é 25.000 (900.000 no total), que são divididas entre o número de clientes. Cada cliente executa um número igual de escritas e leituras. Quando utilizamos o modo de operação *remote*, a distribuição de operações no centro de dados primário e nos remotos é de 95% e 5%, respectivamente.

5.3 Resultados

Começamos por apresentar as experiências comparando a nossa solução (C^3) com configurações do Cassandra que permitem medir a carga adicional introduzida pela consistência causal. De seguida comparamos a nossa solução com o Saturn. As métricas que identificámos como mais importantes foram: *(i) débito do sistema:* que representa o número de operações que o sistema consegue executar por unidade de tempo; *(ii) latência observada pelos clientes:* que representa o tempo de espera do utilizador; *(iii) tempo de visibilidade dos dados:* importante devido ao trade-off entre recência dos dados e o débito (ver § 1).

Desempenho vs Cassandra As figuras 1a e 1d mostram a relação entre o número de clientes, o débito do sistema, e a latência observada pelos clientes quando comparando a nossa solução com as diferentes configurações do Cassandra. Cada ponto em cada linha representa o número de clientes a executar operações simultaneamente. A figura 1a representa os clientes a executar apenas operações locais, enquanto a figura 1d representa tanto operações locais como remotas. Estes resultados não são surpreendentes, sendo melhores quanto mais perto e menos forem os nós que necessitam de responder ao cliente. O nosso sistema mostra desempenho inferior devido à carga adicional necessária para garantir consistência causal, que consideramos ser aceitável.

As figuras 1b e 1c reportam a média e o percentil 95 da latência observada pelos clientes quando utilizando operações locais, enquanto as figuras 1e e 1f mostram o mesmo mas para operações locais e remotas. Estas figuras mostram os resultados de experiências com 9000 clientes (quinto ponto nas figuras 1a e 1d). Seleccionámos estes pontos por serem os que mostram débito máximo nas soluções de causalidade (visível na figura 2).

A latência de leituras locais são semelhantes em todas as configurações, excepto na configuração que utiliza quórum global do Cassandra, visto ser a única que precisa de respostas de centros de dados remotos. Nas operações de escrita, as configurações do Cassandra têm latências proporcionais ao número e distância

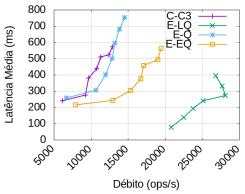
dos nós necessários ao quórum. A nossa solução é a que mostra maior latência, devido coordenação com a camada de causalidade em cada escrita.

Com operações locais e remotas, o Cassandra exibe resultados semelhantes aos anteriores visto a execução de operações remotas seguir a mesmo lógica que as locais. O comportamento na nossa solução altera-se consideravelmente visto os clientes precisarem de executar operações de migração. As latências de leitura são menores, devido aos centros de dados estarem menos carregados, no entanto as operações de migração apresentam uma latência não desprezavel, visto esperarem pela execução das operações de escrita das quais dependem.

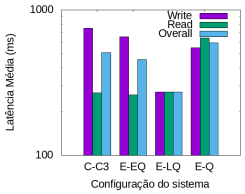
Desempenho vs Saturn As figuras 2a a 2f representam as mesmas métricas que as figuras 1a a 1f, mas comparamos a nossa solução com o Saturn.

Nas figuras de débito e latência, no cenário com apenas operações locais o Saturn aparentemente tem desempenho bastante melhor que o C^3. Isto acontece porque o Saturn consegue executar operações de escrita localmente rápidamente ao evitar coordenar com a camada de causalidade. No entanto, como mostraremos mais tarde, isto também resulta no Saturn ser muito ineficiente a executar operações remotas.

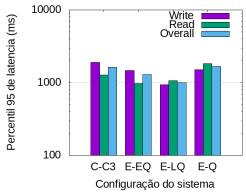
No cenário com operações remotas, o C^3 mostra desempenho bastante melhor que o Saturn, com o desempenho do Saturn a diminuir drasticamente. Isto acontece porque os clientes precisam de utilizar a camada de causalidade para migrar entre centros de dados. Visto as operações de migração necessitarem de esperar pela execução das operações de escrita anteriores, o ritmo lento ao qual o Saturn executa operações remotas levam as operações de migração a esperar bastante tempo antes de serem executadas. Em contraste, na nossa solução, ao user mais meta-dados, permite executar muitas mais operações remotas em con-



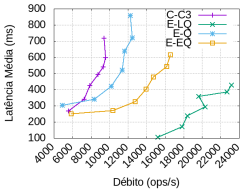
(a) Débito e latência com operações locais



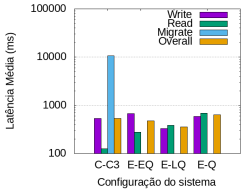
(b) Latências médias com operações locais



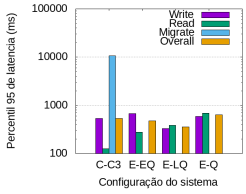
(c) Percentil 95 de latências com operações locais



(d) Débito e latência com operações locais e remotas



(e) Latências médias com operações locais e remotas



(f) Percentil 95 de latências com operações locais e remotas

Figura 1: Comparação de desempenho entre o C^3 e configurações do Cassandra

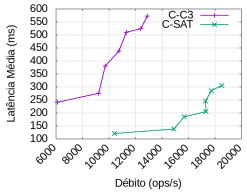
corrência, evitamos longas filas de espera e, como tal, as operações de migração são mais rápidas, evitando deixar os clientes inactivos durante muito tempo.

Olhando para as latências das operações, e começando com o cenário local, a figura 2b apresenta um separação grande entre as latências dos dois sistemas. Como explicado anteriormente, isto é devido às operações de escrita no Saturn não coordenarem com a camada de causalidade, o que limita a execução remota.

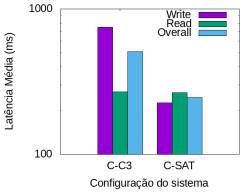
Olhando para o cenário com operações locais e remotas (figura 2b), podemos ver resultados bastante diferentes. As escritas continuam a ter latências menores no Saturn. No entanto, o mais interessante de notar é a latência das operações de migração, particularmente elevada no Saturn. Isto deve-se a esta ser a única operação que necessita de utilizar a camada de causalidade antes do cliente receber a resposta. Nesta operação, a diferença entre o Saturn e a nossa solução é muito significativa (lembramos que o eixo do Y está em escala logarítmica), resultando na latência geral ser maior no Saturn.

Tempos de visibilidade Durante as discussões anteriores, mencionámos a diferença nos tempos de execução de operações remotas entre a nossa solução e o Saturn. Na figura 3 reportamos esses valores. Os resultados mostram a média de tempos de visibilidade das operações de escrita (i.e quanto tempo demora uma operação de escrita a ser executada em todos os centros de dados). Mais uma vez, realçamos que o eixo Y tem escala logaritmica.

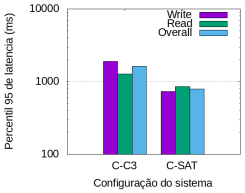
Visto os tempos de visibilidade não serem afectados pelos clientes executarem operações locais ou remotas, os resultados são semelhantes em ambas as figuras. No entanto, conseguimos ver uma diferença considerável entre o C^3 e o Saturn, o que justifica o nosso argumento de que os tempos de visibilidade do Saturn



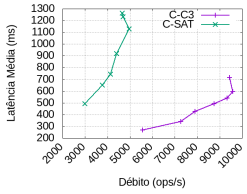
(a) Débito e latência com operações locais



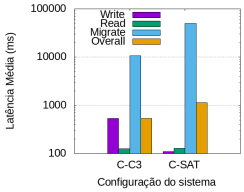
(b) Latências médias com operações locais



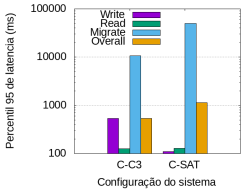
(c) Percentil 95 de latêncisa com operações locais



(d) Débito e latência com operações locais e remotas

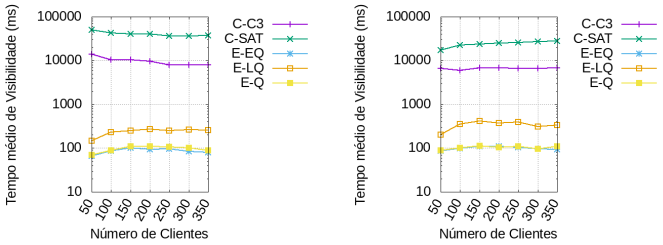


(e) Latências médias com operações locais e remotas



(f) Percentil 95 de latências com operações locais e remotas

Figura 2: Comparação de desempenho entre o C^3 e o Saturn



(a) Com apenas operações locais

(b) Com operações locais e remotas

Figura 3: Tempos de visilibidade de cada configuração

são negativamente afectados pelas limitações de concorrência na execução de operações remotas, o que, por sua vez, degrada o desempenho do sistema.

5.4 Análise de Resultados

Comparando com o Cassandra, o custo de adicionar consistências causal+ é visível no desempenho mais baixo da nossa solução. No entanto, acreditamos ser um custo razoável, especialmente quando comparando com a configuração do Cassandra que utiliza o nível de consistência *EACH_QUORUM*, que pode ser visto como tendo garantias mais próximas da consistência causal.

Comparando com o Saturn, a nossa solução não parece objectivamente melhor em todos os cenários. No entanto, visto a migração de clientes ser inevitável quando se utiliza replicação parcial, a nossa abordagem, ao permitir balançar a execução de operações locais e remotas, mostra resultados bastante melhores. Isto leva-nos a concluir que o uso de meta-dados adicionais para monitorizar dependências causais entre operações é um factor importante, visto possibilitar maior paralelismo na execução de operações remotas.

6 Conclusão

Neste artigo apresentámos um estudo dos desafios associados à criação de um protocolo capaz de suportar consistência causal+ num cenário com replicação parcial e geo-replicação. Explorámos uma abordagem recente que consiste em separar a camada de armazenamento da camada que lida com as dependências causais, permitindo à camada da causalidade operar com meta-dados pequenos, diminuindo a sua carga. Como resultado deste estudo, uma nova solução de consistência causal foi criada, tendo em conta um sistema de armazenamento de dados existente, suportando geo-replicação e replicação parcial, e necessitando de alterações pequenas ao sistema de armazenamento. Além desta solução, também apresentámos uma implementação concreta sobre o Cassandra, um sistema popular que proporciona consistência eventual e descrevemos as alterações necessárias para implementar a nossa solução sobre este sistema. Os resultados experimentais, retirados de um ambiente realista, geo-replicado e com replicação parcial, mostram que o nosso protocolo é capaz de manter um bom balanço entre a execução de operações locais e remotas, utilizando meta-dados suficientes de forma a permitir concorrência na execução das operações remotas.

Referências

1. Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: Proc. of IEEE ICDCS. pp. 405–414. IEEE (2016)
2. Almeida, S., Leitão, J., Rodrigues, L.: Chainreaction: a causal+ consistent datastore based on chain replication. In: Proc. of Eurosys. pp. 85–98. ACM (2013)
3. Attiya, H., Ellen, F., Morrison, A.: Limitations of highly-available eventually-consistent data stores. IEEE TPDS **28**(1), 141–155 (2017)
4. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Trans. Comput. Syst. **5**(1), 47–76 (Jan 1987)
5. Bravo, M., Diegues, N., Zeng, J., Romano, P., Rodrigues, L.E.: On the use of clocks to enforce consistency in the cloud. IEEE Data Eng. Bull. **38**(1), 18–31 (2015)
6. Bravo, M., Rodrigues, L., Roy, P.V.: Saturn: a distributed metadata service for causal consistency. In: Proceedings of the 12nd Eurosys. ACM (2017)
7. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proc. of SOCC. pp. 143–154. ACM (2010)
8. Corbett, J.C., et. al.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. **31**(3), 8:1–8:22 (Aug 2013)
9. DeCandia, G., et. al.: Dynamo: amazon's highly available key-value store. ACM SIGOPS op. sys. review **41**(6), 205–220 (2007)
10. Du, J., Elnikety, S., Roy, A., Zwaenepoel, W.: Orbe: Scalable causal consistency using dependency matrices and physical clocks. In: Proc. of SOCC (2013)
11. Du, J., Iorgulescu, C., Roy, A., Zwaenepoel, W.: Gentlerain: Cheap and scalable causal consistency with physical clocks. In: Proc. of SOCC. pp. 1–13. ACM (2014)
12. Guerraoui, R., Pavlovic, M., Seredinschi, D.A.: Trade-offs in replicated systems. IEEE Data Engineering Bulletin **39**, 14–26 (2016)
13. Gunawardhana, C., Bravo, M., Rodrigues, L.: Unobtrusive deferred update stabilization for efficient geo-replication. In: Proc. of USENIX ATC. pp. 83–95 (2017)
14. Klophaus, R.: Riak core: Building distributed applications without shared state. In: ACM SIGPLAN Commercial Users of Func. Program. p. 14. ACM (2010)
15. Ladin, R., Liskov, B., Shrira, L.: Lazy replication: Exploiting the semantics of distributed services. In: Proc. of PODC '90. pp. 43–57. New York, NY, USA (1990)
16. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (2010)
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (Jul 1978)
18. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In: Proc. of ACM SOSP. pp. 401–416. ACM (2011)
19. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: NSDI. vol. 13, pp. 313–328 (2013)
20. Mahajan, P., Alvisi, L., Dahlin, M., et al.: Consistency, availability, and convergence. University of Texas at Austin Tech Report **11** (2011)
21. Mahmoud, H., Nawab, F., Pucher, A., Agrawal, D., El Abbadi, A.: Low-latency multi-datacenter databases using replicated commit. VLDB Endow. **6**(9) (Jul 2013)
22. Moniz, H., Leitão, J., Dias, R.J., Gehrke, J., Preguiça, N., Rodrigues, R.: Blotter: Low latency transactions for geo-replicated storage. In: Proc. of WWW (2017)
23. Silberstein, A., et. al.: PNUTS in flight: Web-scale data serving at yahoo. IEEE Internet Computing **16**(1), 13–23 (Jan 2012)
24. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (Jan 2009)