Difusão Causal Flexível e Escalável para Replicação na Periferia *

Ema Vieira, Pedro Fouto, Nuno Preguiça e João Leitão

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

Resumo Os sistemas de armazenamento distribuídos são essenciais para a operação de serviços *on-line* de grande escala. O desenho destes sistemas tem evoluído no sentido de melhorar a sua latência e disponibilidade, de forma a proporcionar aos utilizadores a melhor experiência possível. Com este objetivo em mente, têm surgido propostas de novos algoritmos de replicação que tiram partido de modelos de coerência fraca, em particular do modelo causal+. Uma forma de construir estas soluções passa por recorrer a um mecanismo de difusão causal para propagar operações de escrita entre réplicas numa ordem que é compatível com a ordem causal. No entanto, os mecanismos de difusão causal existentes apresentam limitações: ou alavancam em topologias em árvore estáticas, não sendo adaptáveis a cenários com falhas, ou requerem uma quantidade de metadados que cresce linearmente com o número de réplicas, sendo pouco escaláveis e não se adequando a cenários de computação na periferia da rede. Neste artigo, propomos um novo esquema de replicação que garante coerência causal+ e tira partido de *Conflict-free Replicated Data Types* (CRDTs) para permitir a configuração de políticas de resolução de conflitos encapsuladas na lógica dos CRDTs. A nossa solução assenta num esquema de difusão causal descentralizado de operações de escrita, baseado numa árvore capaz de se auto-reconfigurar, mantida por uma especialização do protocolo Plumtree. Contrariamente a outras soluções baseadas em árvores, a nossa solução é capaz de lidar automaticamente com reconfigurações da árvore devido à entrada de nós ou a mudanças de ligações entre nós. Avaliámos experimentalmente um protótipo preliminar da nossa solução numa rede emulada que mostra os benefícios da nossa solução quando comparada com soluções alternativas de difusão causal descentralizada.

Keywords: Propagação Causal · Sistemas de Armazenamento Distribuídos · Coerência Causal+ · Replicação em Grande Escala

1 Introdução

Os sistemas de armazenamento são uma componente essencial para a operação de serviços *on-line* de grande escala. Com a massificação destes serviços e o aumento

* Este trabalho foi parcialmente financiado pela FC&T através do NOVA LINCS (UID/CEC/04516/2013) e do projeto NG-STORAGE (PTDC/CCI-INF/32038/2017). As experiências apresentadas foram conduzidas no Grid'5000, que é suportado por um *scientific interest group* do Inria que inclui CNRS, RENATER, várias universidades e outras organizações (https://www.grid5000.fr/).

do seu número de utilizadores, as expectativas dos mesmos relativamente ao desempenho e tempos de resposta viram um aumento significativo. Este facto levou a que muitas aplicações tenham abandonado soluções de armazenamento clássicas com coerência forte (com semânticas transacionais ACID), em favor de bases de dados distribuídas com garantias de coerência fraca, mas que oferecem alta disponibilidade e tempos de resposta consideravelmente mais baixos [3, 6, 11, 16, 13, 14, 5, 2, 1].

Esta tendência tem-se acentuado com o aparecimento de soluções de armazenamento geo-replicadas que permitem tolerar falhas de centros de dados, ao mesmo tempo que oferecem pontos de presença próximos dos utilizadores de aplicações espalhados por todo o mundo. Neste contexto, os primeiros sistemas de armazenamento geo-replicados ofereciam apenas coerência eventual [4]. No entanto, as poucas garantias oferecidas por este modelo de coerência requerem um esforço de desenvolvimento muito grande da parte dos programadores, que têm de lidar explicitamente com as inúmeras anomalias permitidas. Adicionalmente, estas anomalias podem ser visíveis pelos utilizadores, levando a interações contra-intuitivas com o sistema.

De forma a contornar os desafios impostos pelo uso de coerência eventual, vários sistemas recentes optam por oferecer coerência causal+ [13, 14, 5, 2], que permite a disponibildiade do sistema mesmo na presença de partições de rede e com tempos de resposta relativamente rápidos, garantindo ao mesmo tempo que o estado de cada réplica evolui respeitando a causalidade entre operações. Isto permite aos programadores raciocinar facilmente sobre os possíveis estados do sistema e fornece uma semântica adequada para muitas aplicações.

No entanto, algumas destas soluções podem apresentar tempos de resposta elevados [3, 6] quando o número de operações é muito grande, devido à saturação dos canais de rede que permitem aos dispositivos dos utilizadores contactar com os centros de dados na nuvem. Assim, há tendência para criar sistemas em que as réplicas são colocadas ou nos dispositivos dos clientes [11] (não é uma solução adequada para dispositivos com menor capacidade computacional), ou em pontos de presença na periferia da rede, que se encontram mais próximos dos utilizadores finais, diminuindo assim os tempos de resposta das aplicações [15] e distribuindo a carga pelos vários nós de periferia dispersos a nível global.

Infelizmente, muitas das soluções existentes que oferecem coerência causal+ têm baixa escalabilidade devido à quantidade de informação de controlo que precisam de manter e transmitir quando difundem operações entre as réplicas, de forma a codificar as dependências causais de cada operação. Esta informação tende a crescer linearmente com o número de réplicas no sistema [6], o que impossibilita o seu uso em ambientes com milhares de réplicas. Existem soluções que mitigam a quantidade de informação de controlo propagada com as operações ao usarem topologias em árvore [3]. No entanto, estas topologias são computadas *off-line* e não conseguem lidar com mudanças na configuração da árvore devido a falhas ou entrada de réplicas, ou até mesmo devido a mudanças nos padrões de comunicação das mesmas. Assim, um dos grandes desafios dos novos sistemas de armazenamento na periferia é encontrar formas de minimizar o custo de comunicação devido a metadados, de forma a permitir a sua escalabilidade para centenas, ou mesmo milhares, de réplicas.

Neste artigo, partimos da observação de que uma forma simples de construir uma solução de replicação que ofereça coerência causal é através do uso de um mecanismo de difusão causal com garantias de entrega. Com base nesta observação, propomos um novo esquema de replicação que garante coerência causal+ baseando-se em propagação descentralizada de operações utilizando uma estrutura em árvore, sendo uma especialização do protocolo Plumtree [9], no qual integrámos mecanismos que asseguram a entrega causal mesmo em cenários com falhas ou reconfigurações das ligações entre as réplicas. Para que a nossa solução ofereça coerência causal+, alavancamos em *Conflict-free Replicated Data Types* (CRDTs) de forma a ajustar as políticas de resolução de conflitos entre operações concorrentes. Ao usar uma topologia em árvore, a nossa solução não requer a transmissão de metadados que crescem linearmente com o número de réplicas, excepto em passos de sincronização par-a-par pouco frequentes.

Conduzimos uma avaliação experimental de um protótipo da nossa solução que mostra que, comparativamente com outras soluções de replicação baseadas em mecanismos de difusão causal descentralizados, a nossa solução consegue tempos de visibilidade das operações globalmente baixos, com um custo de comunicação baixo, encontrando-se num ponto de equilíbrio interessante relativamente a estes dois critérios e sendo uma solução mais adequada para suportar a replicação na periferia, pois lida com falhas de forma eficiente e correta.

O resto deste artigo encontra-se organizado da seguinte forma: na secção 2 discutimos o trabalho relacionado. A secção 3 descreve a nossa solução, com ênfase no novo mecanismo de difusão causal. A secção 4 apresenta o nosso ambiente experimental e os resultados obtidos, bem como a sua análise. A secção 5 conclui o artigo e, finalmente, a secção 6 discute trabalho futuro.

2 Trabalho Relacionado

Modelos de Coerência. O teorema CAP [7] mostrou que é impossível que um sistema distribuído seja simultaneamente fortemente coerente, tolerante a partições e sempre disponível. Consequentemente, os sistemas de armazenamento distribuídos têm de optar por duas destas três propriedades. Os sistemas que oferecem coerência forte tendem a experienciar latências consideráveis e perdem disponibilidade na presença de partições de rede (ou falhas de números significativos de réplicas). Por outro lado, os sistemas de armazenamento que adotam modelos de coerência fraca são mais responsivos e disponíveis, mas permitem que o estado das réplicas divirja temporariamente.

A coerência eventual garante apenas que as réplicas do sistema convergem em algum momento, quando não existirem mais operações de escrita. Por outro lado, a coerência causal garante que uma operação apenas é executada depois de todas as operações das quais depende. Os sistemas que oferecem este tipo de coerência têm de manter metadados sobre as operações de maneira a conseguirem identificar as suas dependências causais. Um dos tipos de metadados mais utilizados são vetores de versão, que se baseiam na relação de *happens-before* [8] para determinar a ordem causal dos eventos que modificam as réplicas da base de dados. Estes vetores de versão têm de ter uma entrada por cada réplica e, consequentemente, a carga que impõem nos processos de difusão de operações cresce linearmente com o número de réplicas. A coerência causal+ espande a

coerência causal introduzindo convergência das réplicas através de resolução de conflitos usando, p.e., CRDTs ou políticas como *Last Writer Wins*.

Sistemas de Armazenamento Causalmente Coerentes. O grande desafio do desenho de novos sistemas de armazenamento causalmente coerentes é encontrar um equilíbrio adequado entre a latência média de visibilidade das atualizações e o débito do sistema, que depende da quantidade de metadados associados a cada operação disseminada.

Sistemas como o COPS [13] e Eiger [14] fazem verificação explícita de dependências, tendo por isso de manter para cada versão de um objeto de dados o conjunto das suas dependências. Estes sistemas guardam uma quantidade considerável de metadados e tendem a apresentar um débito inferior ao de soluções como o GentleRain [5] que, ao sumarizar as dependências num só escalar, sacrifica a latência de visibilidade das operações por um maior débito. Soluções que armazenam metadados sob a forma de vetores de versão, como o Cure [1], têm um bom balanço entre estas duas caraterísticas mas não foram desenhadas com centenas, ou milhares, de réplicas em mente, não sendo adequadas para uso em sistemas na periferia.

Por fim, existem sistemas que utilizam topologias de rede especializadas para disseminar operações por ordem causal. O Saturn [3] é um sistema de gestão de metadados que pode ser combinado com sistemas de armazenamento pré-existentes e que, internamente, organiza os seus nós numa estrutura em árvore onde os centros de dados são as folhas, conseguindo assim entregar os metadados numa ordem que é compatível com a ordem causal. No entanto, a árvore usada pelo Saturn não é tolerante a falhas, visto ser computada estaticamente e *a priori*, tendo de ser recomputada e reconfigurada sempre que uma réplica fica inacessível ou se junta ao sistema.

Árvores de Difusão Epidémica. O Plumtree [9] é um algoritmo descentralizado de difusão epidémica baseado numa estrutura em árvore emergente de um processo de disseminação aleatório, que consegue reduzir o custo de comunicação do processo de disseminação e manter alta confiabilidade. A árvore de difusão é criada sobre uma rede sobreposta aleatória mantida por um protocolo de gestão de filiação como, por exemplo, o HyParView [10]. Um dos pontos fortes deste algoritmo é conseguir, de forma descentralizada, reconfigurar a árvore com a entrada e saída de nós da rede.

Este algoritmo utiliza dois tipos de difusão epidémica: imediata (*eager push*) e diferida (*lazy push*). A difusão imediata é utilizada para enviar mensagens a nós através dos ramos da árvore. As ligações restantes são utilizadas para anunciar os IDs das mensagens que o nó recebeu. Os ramos iniciais da árvore são definidos pelo caminho de propagação da primeira mensagem enviada, que tende a minimizar a latência para esse emissor. Os nós criam esta árvore podando os ramos através dos quais recebem mensagens duplicadas. Quando um nó deteta atrasos na entrega de uma mensagem, este enxerta um novo ramo na árvore para um dos vizinhos que a anunciou, garantindo que os nós nunca ficam isolados durante longos períodos de tempo. Apesar de ser um protocolo interessante para cenários de computação na periferia, devido à sua natureza descentralizada, esta solução não garante entrega causal.

3 Solução Proposta

Neste trabalho assumimos um sistema composto por várias réplicas que podem estar localizadas em centros de dados, ou em pontos de acesso na periferia. Assumimos que o número total de réplicas pode variar ao longo do tempo, estando na ordem das centenas ou milhares. Consideramos um sistema com replicação total, onde todas as réplicas mantêm o mesmo conjunto de objetos de dados codificados sob a forma de CRDTs.

No nosso modelo, as aplicações interagem com CRDTs na réplica mais próxima. Assumimos que os clientes interagem sempre com a mesma réplica (a migração de réplicas foi já explorada, p.e., em [3]). As operações de leitura podem ser resolvidas localmente de forma imediata. As operações de escrita são executadas pela réplica que recebe a operação do cliente diretamente sobre a sua cópia local do CRDT, sendo posteriormente associadas a metadados de controlo e disseminadas através da árvore de difusão causal.

Note-se que a nossa proposta de algoritmo de difusão garante entrega causal fiável (baseada na relação de *happens-before*) que, aliada à utilização de CRDTs para resolução automática de conflitos, faz com que o nosso sistema ofereça coerência causal+.

3.1 Visão Geral

A nossa solução tem como base um algoritmo de difusão causal que usa uma árvore gerida de forma descentralizada (inspirada no protocolo Plumtree [9] discutido anteriormente). Note-se que, o correto funcionamento do Plumtree depende de dois fatores subjacentes:

- **Vistas parciais simétricas:** Cada nó do sistema não tem conhecimento de todos os outros nós, tendo uma vista parcial. O protocolo de gestão da rede sobreposta deve garantir que as vistas parciais dos nós são simétricas, ou seja, se o nó A pertence à vista parcial do nó B, então o nó B encontra-se na vista parcial do nó A. Só assim é possível que os múltiplos nós da rede partilhem a mesma árvore de disseminação. Na nossa solução, esta propriedade é garantida gerindo a rede sobreposta com o protocolo HyParView [10];
- **Canais de comunicação *First In, First Out* (FIFO):** Os nós da rede devem manter ligações FIFO entre si para que a propagação de mensagens através dos ramos da árvore seja feita pela ordem de receção. Isto pode ser conseguido utilizando ligações TCP entre os vários nós.

Como explicado em [12], qualquer topologia em árvore que assuma comunicação FIFO oferece garantias de entrega causal. Assim, o Plumtree é inerentemente causal enquanto a estrutura da árvore se mantiver inalterada, sendo o grande desafio manter as garantias da causalidade aquando da criação de novos ramos na árvore, quer por entrada de novos nós na rede, quer por reconfiguração da estrutura da árvore.

Para endereçar este desafio, desenvolvemos mecanismos de sincronização baseados na troca de vetores de versão e na retransmissão das operações necessárias para que os nós fiquem atualizados em relação aos seus (novos) vizinhos na árvore. Note-se que, apesar de os nós conseguirem computar um vetor de versão que captura o seu estado local, este não necessita de ser transmitido com

cada operação disseminada através da árvore, pois isso iria incorrer num custo excessivo de comunicação. Para evitar isto, cada réplica mantém um contador local, inicializado a zero, que é incrementado e atribuído (atomicamente) a cada operação de escrita recebida por essa réplica. As operações disseminadas pela árvore contêm o identificador da réplica que recebeu a operação do cliente e o seu contador, permitindo a cada nó computar localmente o seu vetor de versão. Através da troca destes vetores de versão, cujo diferencial identifica as operações em falta em cada nó, é possível fazer a sincronização entre dois nós (passando estes a ser vizinhos na árvore).

Para podermos retransmitir operações, recorremos a uma camada de replicação, à qual nos iremos referir como núcleo de replicação, que mantém a lista causalmente ordenada de todas as operações executadas localmente. Isto permite a cada nó computar as operações que devem ser enviadas para o seu par aquando da sincronização e garantir que o envio dessas mensagens é realizado por uma ordem que não viola a causalidade. De seguida, detalhamos este novo mecanismo de sincronização.

3.2 Mecanismo de Sincronização

A criação e manutenção da árvore de difusão segue uma estratégia similar à proposta pelo protocolo Plumtree (ver Alg. 1 e Alg. 2), utilizando mensagens de poda e enxerto de ramos. Contudo, na nossa solução, qualquer ramo que seja criado tem de passar primeiro por um processo de sincronização bi-direcional, durante o qual todas as mensagens recebidas por outros ramos da árvore são mantidas num *buffer* ordenado, sendo enviadas apenas após a conclusão do mecanimo de sincronização. Este mecanismo de sincronização divide-se em dois passos principais:

1. **Solicitação e envio do vetor de versão:** Quando um nó pretende sincronizar com outro, este envia-lhe uma mensagem de solicitação de vetor de versão (Alg. 2 - linhas 102-106). Quando o vizinho recebe essa mensagem, comunica com o seu núcleo de replicação de CRDTs local de modo a obter o seu vetor de versão atual (Alg. 2 - linhas 60-63) e encaminha-o para o nó que o solicitou (Alg. 1 - linhas 18-19). De maneira a minimizar o número de duplicados transmitidos, cada nó envia o seu vetor de versão a um vizinho de cada vez, mantendo assim uma lista de vizinhos aos quais tem de enviar um vetor de versão após receção das operações de sincronização do vizinho corrente (Alg. 2 - linhas 64-65).
2. **Envio das operações em atraso:** Quando um nó recebe o vetor de versão de um vizinho, este guarda todas as mensagens que recebe a partir desse instante num *buffer* local de forma ordenada (Alg. 2 - linhas 114-116) e comunica com o seu núcleo de replicação (Alg. 2 - linhas 55-57). O núcleo retorna-lhe as operações em falta, ordenadas causalmente, através da análise do vetor de versão recebido. Por fim, o nó envia as operações em falta ao vizinho (seguidas das operações do *buffer*) e adiciona o vizinho à sua árvore de difusão, estando a sincronização completa (Alg. 1 - linhas 20-29) e prosseguindo o processo de disseminação pela árvore de forma regular.

É de notar que o mecanismo descrito anteriormente é uni-direcional, ou seja, no fim do processo, apenas o nó que o iniciou pode enviar mensagens livremente

para o vizinho com quem sincronizou. Por sua vez, esse vizinho teria de iniciar o seu próprio processo de sincronização com o nó para lhe poder enviar mensagens sem quebrar as garantias da causalidade. Existem apenas duas situações em que um ramo da árvore é criado (iniciando o mecanismo de sincronização) e, garantidamente, ambas despoletam a sincronização no sentido oposto:

- **Adição de um novo vizinho:** Aquando da entrada de um nó na rede, ou quando o HyParView modifica os vizinhos do nó local, ambos os nós no extremo da nova ligação recebem uma notificação de adição de um novo vizinho (Alg. 2 - linhas 83-89). Se a árvore já estiver formada, de maneira a não a destabilizar, esta notificação faz com que o novo vizinho seja adicionado ao conjunto de vizinhos `diferidos`. Caso a árvore ainda esteja por formar, a notificação inicia o processo de sincronização nos dois sentidos da ligação;
- **Enxerto de um ramo da árvore:** Quando o temporizador de receção de uma mensagem expira, o nó despoleta a política de envio de enxertos, iniciando a sincronização com um dos nós que anunciou o ID dessa mensagem (Alg. 2 - linhas 76-82). Para não causar instabilidade na árvore aquando da entrada de novos nós no sistema, esta política espaça o envio dos pedidos de enxerto para que uma sincronização tenha tempo de terminar antes do enxerto seguinte ser enviado, evitando assim a criação de ramos desnecessários na árvore. Quando um nó recebe uma mensagem de enxerto, o processo de sincronização é iniciado do seu lado (Alg. 2 - linhas 47-48).

Algorithm 1: Plumtree Causal (1/2)

```
1  Upon Init(idLocal, Δt1, Δt2) do:
2    idLocal ←− idLocal; //ID do nó
3    vistaParcial ←− {}; //Conjunto de todos vizinhos
4    imediatos ←− {}; //Conjunto dos vizinhos pertencentes à árvore
5    diferidos ←− {}; //Conjunto dos vizinhos não pertencentes à árvore
6    sincsAtivas ←− {}; //Conjunto dos vizinhos com os quais estamos a sincronizar
7    vvPendentes ←− {}; //Fila dos vizinhos à espera do envio de vetor de versão
8    vvCorrente ←− (⊥, ⊥); //Par de ID de mensagem de sincronização e ID do vizinho para o qual enviámos o último vetor
9    //de versão
10   emEspera ←− {}; //Mapa de IDs de mensagens para listas de mensagens recebidas entre a receção de um vetor de versão
11   //e o envio das operações em atraso a esse vizinho
12   porReceber ←− {}; //Mapa de IDs de mensagens para os vizinhos que as possuem
13   recebidas ←− {}; //Conjunto de IDs das mensagens recebidas
14   temporizadoresAtivos ←− {}; //Temporizadores ativos

15 Upon PedidoDifusão(mid, conteúdo) do:
16   trigger NotificaçãoEntrega(mid, conteúdo, idLocal);
17   call processarMensagemEpidémica(mid, conteúdo, idLocal);

18 Upon PedidoEnvioVetorVersão(mid, vv, vizinho) do:
19   trigger send MensagemVetorVersão(mid, vv) to vizinho;

20 Upon PedidoEnvioOperaçõesSincronização(mid, ops, vizinho) do:
21   trigger send MensagemOperaçõesSincronização(mid, ops) to vizinho;
22   q ←− remover(mid, emEspera);
23   forall (idMsg, conteúdo, de) ∈ q do:
24     if vizinho ≠ de then:
25       trigger send MensagemEpidémica(idMsg, conteúdo) to vizinho;
26   if vizinho ∈ vistaParcial then:
27     imediatos ←− imediatos ∪ vizinho;
28     sincsAtivas ←− sincsAtivas \ vizinho;
29     diferidos ←− diferidos \ vizinho;

30 Upon Receive MensagemEpidémica(mid, conteúdo) from de do:
31   if mid ∉ recebidas then:
32     trigger NotificaçãoEntrega(mid, conteúdo, de);
33     call processarMensagemEpidémica(mid, conteúdo, de);
34   else:
35     if de ∈ vistaParcial then:
36       imediatos ←− imediatos \ de;
37       call removerVizinhoDeVvPendentes(de);
38       (id, vizinho) ←− vvCorrente;
39       if de = vizinho then:
40         call tentarPróximaSincronização();
41       diferidos ←− diferidos ∪ de;
42       trigger send MensagemPodar() to de;
```

Algorithm 2: Plumtree Causal (2/2)

```
43  Upon Receive MensagemPodar() from de do:
44     if de ∈ imediatos then:
45        imediatos ⟵ imediatos \ de;
46        diferidos ⟵ diferidos ∪ de;

47  Upon Receive MensagemEnxertar() from de do:
48     call iniciarSincronização(de, falso);

49  Upon Receive MensagemAnúncio(mid) from de do:
50     if mid ∉ recebidas then:
51        if temporizadoresAtivos[mid] = ⊥ then:
52           tid ⟵ setup timer TemporizadorAnúncio(mid) with Δt1;
53           temporizadoresAtivos[mid] ⟵ tid;
54        porReceber[mid] ⟵ porReceber[mid] ∪ de;

55  Upon Receive MensagemVetorVersão(mid, vv) from de do:
56     emEspera[mid] ⟵ {};
57     trigger NotificaçãoVetorVersão(mid, vv, de);

58  Upon Receive MensagemSolicitaçãoVetorVersão(mid) from de do:
59     if de ∈ vistaParcial then:
60        (id, vizinho) ⟵ vvCorrente;
61        if vizinho = ⊥ then:
62           vvCorrente ⟵ (mid, de);
63           trigger NotificaçãoSolicitaçãoVetorVersão(mid, de);
64        else:
65           vvPendentes ⟵ vvPendentes ∪ (mid, de);
66     else:
67        trigger NotificaçãoSolicitaçãoVetorVersão(mid, de);

68  Upon Receive MensagemOperaçõesSincronização(mid, ops) from de do:
69     forall op ∈ ops do:
70        mid ⟵ getId(op);
71        conteúdo ⟵ getConteúdo(op);
72        if mid ∉ recebidas then:
73           trigger NotificaçãoEntrega(mid, conteúdo, de);
74           call processarMensagemEpidémica(mid, conteúdo, de);
75     call tentarPróximaSincronização();

76  Upon TemporizadorAnúncio(mid) do:
77     if mid ∉ recebidas then:
78        vizinho ⟵ buscarSeguinte(porReceber[mid]);
79        if vizinho ≠ ⊥ then:
80           tid ⟵ setup timer TemporizadorAnúncio(mid) with Δt2;
81           temporizadoresAtivos[mid] ⟵ tid;
82           call políticaEnvioEnxertos(mid, vizinho);

83  Upon NotificaçãoAdicionarVizinho(vizinho) do:
84     vistaParcial ⟵ vistaParcial ∪ vizinho;
85     call abrirConexão(vizinho);
86     if diferidos ≠ {} then:
87        diferidos ⟵ diferidos ∪ de;
88     else:
89        call iniciarSincronização(vizinho, verdadeiro);

90  Upon NotificaçãoRemoverVizinho(vizinho) do:
91     vistaParcial ⟵ vistaParcial \ vizinho;
92     imediatos ⟵ imediatos \ vizinho;
93     diferidos ⟵ diferidos \ vizinho;
94     sincsAtivas ⟵ sincsAtivas \ vizinho;
95     call removerVizinhoDeVvPendentes(vizinho);
96     forall q ∈ porReceber do:
97        q ⟵ q \ vizinho;
98     (id, v) ⟵ vvCorrente;
99     if vizinho = v then:
100       call tentarPróximaSincronização();
101    call fecharConexão(vizinho);

102 Procedure iniciarSincronização(vizinho, novaEntrada) :
103    if novaEntrada ∨ (vizinho ∈ diferidos ∧ vizinho ∉ sincsAtivas) then:
104       sincsAtivas ⟵ sincsAtivas ∪ vizinho;
105       mid ⟵ gerarNovoId();
106       trigger send MensagemSolicitaçãoVetorVersão(mid) to vizinho;

107 Procedure tentarPróximaSincronização() :
108    (id, vizinho) ⟵ buscarSeguinte(vvPendentes);
109    if vizinho ≠ ⊥ then:
110       vvCorrente ⟵ (id, vizinho);
111       trigger send MensagemSolicitaçãoVetorVersão(id) to vizinho;
112    else:
113       vvCorrente ⟵ (⊥, ⊥);

114 Procedure processarMensagemEpidémica(mid, conteúdo, de) :
115    forall q ∈ emEspera do:
116       q ⟵ q ∪ (mid, conteúdo, de);
117    recebidas ⟵ recebidas ∪ mid;
118    tid ⟵ temporizadoresAtivos[mid];
119    if tid ≠ ⊥ then:
120       temporizadoresAtivos[mid] ⟵ ⊥;
121       cancel timer tid;
122    forall vizinho ∈ imediatos : vizinho ≠ de do:
123       trigger send MensagemEpidémica(mid, conteúdo) to vizinho;
124    forall vizinho ∈ diferidos : vizinho ≠ de do:
125       trigger send MensagemAnúncio(mid) to vizinho;
```

Este mecanismo de sincronização faz com que a árvore de difusão epidémica se adapte às mudanças da rede sem, em nenhum momento, quebrar as garantias de causalidade.

4 Avaliação

De forma a aferir o desempenho da nossa solução, para além da nossa versão causal do Plumtree, implementámos também variantes causais de dois protocolos de disseminação: um de Inundação Causal (*flood*), em que as mensagens são propagadas para todos os vizinhos da vista parcial de cada nó assim que são recebidas pela primeira vez; e um de Sincronização Periódica (*periodic pull*), em que cada nó envia, periodicamente, o seu vetor de versão a um vizinho aleatório que, por sua vez, lhe responde com todas as operações que conhece que estão em falta no vetor. O período para o protocolo de Sincronização Periódica foi de 3 segundos, tendo este valor sido escolhido de maneira a ser o mesmo que o *timeout* do Plumtree Causal antes de fazer um enxerto na árvore.

Note-se que, de forma a garantir a causalidade, aquando da adição de um novo vizinho, o protocolo de Inundação Causal executa o mecanismo de sincronização descrito na secção 3.2. Por outro lado, no protocolo de Sincronização Periódica, a causalidade é garantida fazendo com que as operações sejam enviadas por ordem de receção. Todas as soluções utilizam o HyParView como protocolo de gestão de filiação, sendo o conjunto de vizinhos de cada nó correspondente à vista ativa do HyParView, que foi parametrizada com um tamanho de 5.

4.1 Ambiente Experimental

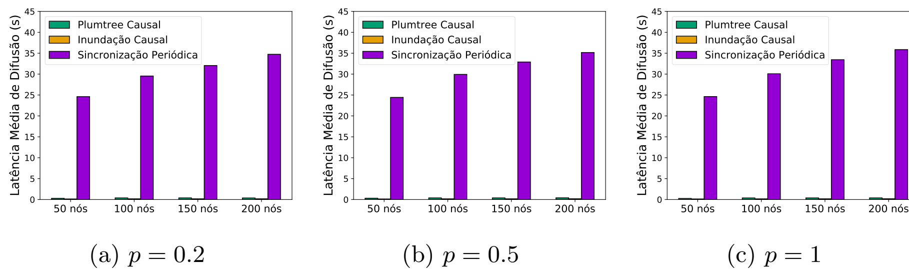
As experiências foram realizadas na plataforma Grid'5000. As máquinas utilizadas têm um CPU de 18 cores (Intel Xeon Gold 5220), 96GB RAM e estão ligadas por uma rede de 25 Gbps. Para avaliar a nossa solução o mais realisticamente possível, recorremos ao *docker swarm*, no qual emulámos uma rede distribuída em que executámos um nó do procotolo de disseminação em cada contentor. Os contentores foram distribuídos pelas máquinas de forma idêntica em toda as experiências.

Nas nossas experiências variámos o número de nós (50, 100, 150 e 200), sendo que a latência entre cada par de nós varia entre 10 e 100 milisegundos. Variámos também a probabilidade ($p = 0.2$, $p = 0.5$ e $p = 1$) de cada nó enviar uma mensagem, de forma a termos cargas de trabalho menos pesadas na rede.

As experiências consistiram em lançar os contentores e esperar até que o protocolo de gestão de filiação (HyParView) estabilizasse. Durante os 5 minutos seguintes, a cada segundo, cada nó gerava duas novas mensagens, com probabilidade p, e disseminava-as de acordo com o protocolo de difusão em uso. Passados esses 5 minutos, os nós deixavam de gerar mensagens. Tendo em conta que o número de mensagens geradas em cada experiência era considerável, esperámos entre 5 a 10 minutos extra para garantir que todas as mensagens eram entregues a todos os nós.

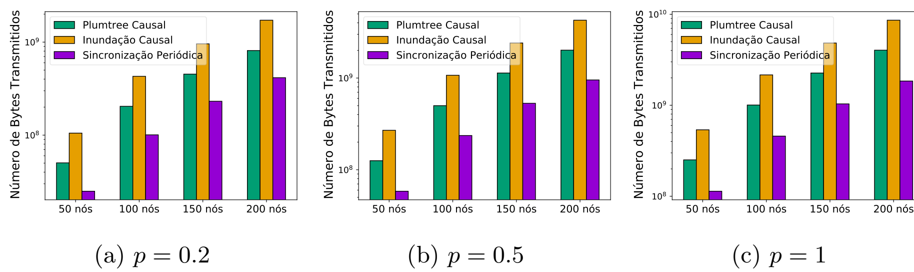
4.2 Resultados

Nesta secção, apresentamos os resultados obtidos, que representam a média das 3 execuções individuais de cada combinação de parâmetros. Das experiências



(a) $p = 0.2$ (b) $p = 0.5$ (c) $p = 1$

Fig. 1: Latência média de difusão de cada protocolo



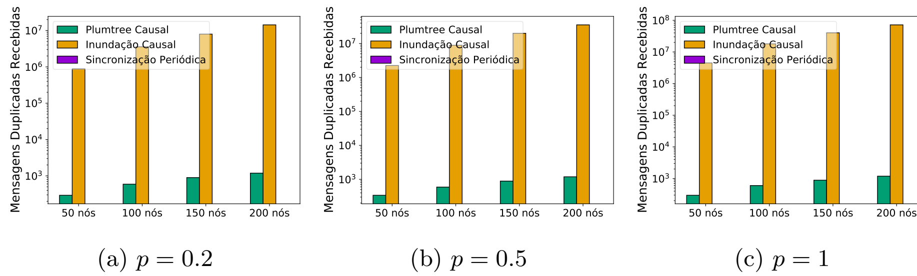
(a) $p = 0.2$ (b) $p = 0.5$ (c) $p = 1$

Fig. 2: Número de bytes transmitidos por cada protocolo

executadas, extraímos o número total de bytes transmitidos pela rede, a latência média de propagação de mensagens e o número total de duplicados recebidos.

As Figuras 1a, 1b e 1c representam a latência média de difusão de mensagens à medida que se aumenta o número de nós do sistema, para probabilidades de geração de operações de 0.2, 0.5 e 1, respetivamente. Tendo em conta que todos os protocolos garantem entrega causal, os valores apresentados também representam os tempos médios de visibilidade de operações de escrita, pois estas podem ser executadas imediatamente após serem entregues. Nestas figuras podemos ver que a nossa solução apresenta latências médias de difusão de mensagens extremamente baixas quando comparadas com o protocolo de Sincronização Periódica. Isto seria de esperar, visto que a nossa solução vai recebendo e executando as operações à medida que são geradas, ao passo que o protocolo de Sincronização Periódica apenas propaga informação no processo de sincronização seguinte. Por outro lado, a nossa solução apresenta latências mais elevadas que o protocolo de Inundação Causal, o que se justifica pelo facto de, na inundação, cada nó enviar as mensagens diretamente para todos os seus vizinhos, reduzindo significativamente número de saltos que uma mensagem tem de dar na rede até que seja entregue a todos os nós. Contudo, a diferença de latências da nossa solução para o protocolo de Inundação Causal é muito ligeira e mantém-se relativamente constante em todos os cenários testados.

As Figuras 2a, 2b e 2c mostram o número total de bytes trasmitidos na rede por cada protocolo à medida que se aumenta o número de nós do sistema, para probabilidades de geração de operações de 0.2, 0.5 e 1, respetivamente. Note-se



(a) $p = 0.2$ (b) $p = 0.5$ (c) $p = 1$

Fig. 3: Número total de duplicados recebidos por cada protocolo

que o eixo do y se encontra em escala logarítmica. A nossa solução apresenta um custo de comunicação significativamente menor que o protocolo de Inundação Causal, visto que o número de bytes por este transmitidos é afetado pela quantidade de nós para os quais as mensagens são propagadas. Contudo, o nosso protocolo envia anúncios para os vizinhos aos quais não envia as mensagens, justificando assim o custo de comunicação superior ao do protocolo de Sincronização Periódica, que não envia nenhum outro tipo de mensagem para além das necessárias para a sincronização (vetor de versão e operações em atraso).

As Figuras 3a, 3b e 3c mostram o número total de duplicados recebidos por cada protocolo à medida que se aumenta o número de nós do sistema, para probabilidades de geração de operações de 0.2, 0.5 e 1, respetivamente. Note-se que o eixo do y se encontra em escala logarítmica. Como seria de esperar, visto que o protocolo de Sincronização Periódica executa apenas uma sincronização de cada vez, este não apresenta mensagens duplicadas. Ao podar ramos da árvore através dos quais são recebidos duplicados, a nossa solução mostra melhorias significativas quando comparada com o protocolo de Inundação Causal que, por se basear numa topologia em grafo, apresenta redundância significativa e, consequentemente, um número de mensagens duplicadas bastante elevado.

5 Conclusão

Neste artigo apresentámos um novo esquema de replicação que oferece garantias de coerência causal+ baseando-se num algoritmo de difusão causal descentralizada e recorrendo a CRDTs para resolução de conflitos entre operações concorrentes. A nossa solução é uma especialização do protocolo Plumtree que é capaz de criar e manter uma árvore de difusão causal de forma eficiente, mesmo quando nós entram e saem da rede, ou quando existem mudanças de ligações entre nós. Introduzimos um novo mecanismo de sincronização que permite manter as garantias da causalidade nos momentos de reconfiguração da árvore e os resultados experimentais mostram que a nossa solução apresenta um bom equilíbrio entre a latência de propagação de mensagens e o custo de comunicação, mostrando-se consideravelmente melhor que outras alternativas descentralizadas.

6 Trabalho Futuro

No protótipo que implementámos, o núcleo de replicação guarda a lista de todas operações executadas no sistema sem qualquer tipo de limitações de espaço.

No futuro, pretendemos explorar a compressão desta lista, criando operações de *checkpoint*. Para além disso, queremos introduzir sincronização por transferência de estado aquando da junção de novos nós à rede. Nas experiências apresentadas, a instabilidade da rede deve-se à mudança das vistas parciais causada pelo HyParView. Contudo, para melhor avaliar o mecanismo de sincronização, iremos testar a nossa solução emulando entradas e falhas de nós. Por fim, planeamos adaptar a nossa solução a cenários com replicação parcial, mantendo várias árvores de difusão causal para cada partição de dados.

Referências

1. Akkoorath, D.D., et al.: Cure: Strong semantics meets high availability and low latency. In: IEEE 36th International Conference on Distributed Computing Systems (2016)
2. Almeida, S., Leitão, J., Rodrigues, L.: Chainreaction: a causal+ consistent datastore based on chain replication. In: Proceedings of the 8th ACM European Conference on Computer Systems (2013)
3. Bravo, M., Rodrigues, L., Van Roy, P.: Saturn: A distributed metadata service for causal consistency. Proceedings of the 12th European Conference on Computer Systems, EuroSys 2017 (2017)
4. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. **41**(6) (Oct 2007)
5. Du, J., Iorgulescu, C., Roy, A., Zwaenepoel, W.: GentleRain: Cheap and scalable causal consistency with physical clocks. Proceedings of the 5th ACM Symposium on Cloud Computing, SOCC 2014 (2014)
6. Fouto, P., Leitão, J., Preguiça, N.: Practical and fast causal consistent partial georeplication. In: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA) (2018)
7. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News **33**(2) (2002)
8. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (Jul 1978)
9. Leitao, J., Pereira, J., Rodrigues, L.: Epidemic broadcast trees. In: 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007) (2007)
10. Leitão, J., Pereira, J., Rodrigues, L.: Hyparview: A membership protocol for reliable gossip-based broadcast (07 2007)
11. van der Linde, A., Fouto, P., Leitão, J.a., Preguiça, N., Castiñeira, S., Bieniusa, A.: Legion: Enriching internet services with peer-to-peer interactions. In: Proceedings of the 26th International Conference on World Wide Web (2017)
12. van der Linde, A., Fouto, P., Leitão, J., Preguiça, N.: The intrinsic cost of causal consistency. Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2020 (2020)
13. 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 (2011)
14. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13) (Apr 2013)
15. Satyanarayanan, M.: The emergence of edge computing. Computer **50**(1) (2017)
16. Zawirski, M., Preguiça, N., Duarte, S., Bieniusa, A., Balegas, V., Shapiro, M.: Write fast, read in the past: Causal consistency for client-side applications. Middleware 2015 - Proceedings of the 16th Annual Middleware Conference (Dc) (2015)