

## 7.5 Protocolos de Consistência

Até aqui, nós nos concentrarmos principalmente em vários modelos de consistência e questões gerais de projeto para protocolos de consistência. Nesta seção, focalizaremos a implementação propriamente dita de modelos de consistência examinando vários protocolos de consistência. Um **protocolo de consistência** descreve uma implementação de um modelo de consistência específico. Seguimos a organização de nossa discussão sobre modelos de consistência examinando, em primeiro lugar, modelos centrados em dados e, em seguida, protocolos para modelos centrados no cliente.

### 7.5.1 Consistência contínua

Como parte de seu trabalho sobre consistência contínua, Yu e Vahdat desenvolveram vários protocolos para lidar com as três formas de consistência. Na discussão a seguir, consideramos várias soluções, omitindo detalhes por questão de clareza.

#### Limitação de desvio numérico

Em primeiro lugar, focalizaremos uma solução para manter o desvio numérico dentro de limites. Mais uma vez, nossa finalidade não é esmiuçar todos os detalhes para cada protocolo, mas dar uma idéia geral. Detalhes sobre limitação de desvio numérico podem ser encontrados em Yu e Vahdat (2000).

Vamos nos concentrar em escritas para um único item de dados  $x$ . Cada escrita  $W(x)$  tem um peso associado que representa o valor numérico pelo qual  $x$  é atualizado, denominado *peso* ( $W(x)$ ), ou simplesmente *peso* ( $W$ ). Para simplificar consideramos que *peso* ( $W$ ) = 0. Cada escrita  $W$  é inicialmente apresentada a um de  $N$  servidores de réplicas disponíveis, caso em que esse servidor se torna a origem da escrita, denotada por *origem* ( $W$ ). Se considerarmos o sistema em um instante específico do tempo, veremos várias escritas apresentadas que ainda precisam ser propagadas para todos os servidores. Com essa finalidade, cada servidor  $S_i$  monitorará um registro  $L_i$  de escritas que ele executou em sua própria cópia local de  $x$ .

Sejam  $TW[i, j]$  as escritas executadas pelo servidor  $S_i$  que se originaram de  $S_j$ :

$$TW[i, j] = \sum \{ \text{peso}(W) | \text{origem}(W) = S_j \text{ e } W \in L_i \}$$

Note que  $TW[i, i]$  representa as escritas agregadas apresentadas a  $S_i$ . Nossa meta é, para qualquer tempo  $t$ , permitir que o valor corrente  $v_i$  no servidor  $S_i$  se desvie dentro de limites em relação ao valor real de  $v(t)$  de  $x$ . Esse valor real é completamente determinado para todas as escritas apresentadas. Isto é, se  $v(0)$  é o valor inicial de  $x$ , então

$$v(t) = v(0) + \sum_{k=1}^N TW[k, k]$$

e

$$v_i = v(0) + \sum_{k=1}^N TW[i, k]$$

Note que  $v_i \leq v(t)$ . Vamos nos concentrar somente em desvios absolutos. Em particular, para cada servidor  $S_i$ , associamos um limite superior  $\delta_i$  tal que precisamos impor:

$$v(t) - v_i \leq \delta_i$$

Escritas apresentadas a um servidor  $S_i$  precisarão ser propagadas para todos os outros servidores. Há modos diferentes de fazer isso, mas o típico é um protocolo epidêmico que permitirá rápida disseminação de atualizações. De qualquer modo, quando um servidor  $S_i$  propaga uma escrita que se origina de  $S_j$  para  $S_k$ , o último poderá tomar conhecimento do valor  $TW[i, j]$  no instante em que a escrita foi enviada. Em outras palavras,  $S_k$  pode manter uma visão  $TW_k[i, j]$  daí que ele acredita que  $S_i$  terá como valor para  $TW[i, j]$ . É óbvio que

$$0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j]$$

A idéia toda é que, quando o servidor  $S_k$  notar que  $S_i$  não está acompanhando o ritmo correto das atualizações que foram apresentadas a  $S_k$ , ele envie escritas de seu registro para  $S_i$ . Na realidade, esse envio *adianta* a visão  $TW_k[i, k]$  que  $S_k$  tem de  $TW[i, k]$ , tornando menor o desvio  $TW[i, k] - TW_k[i, k]$ . Em particular,  $S_k$  adianta sua visão de  $TW[i, k]$  quando uma aplicação apresenta uma nova escrita que aumentaria  $TW[k, k] - TW_k[i, k]$  mais do que  $\delta_i/(N-1)$ . Como exercício, sugerimos que o leitor mostre que o adiantamento sempre garante que  $v(t) - v_i \leq \delta_i$ .

#### Limitação de desvio de idade

Há muitos modos de manter a idade de réplicas dentro de limites especificados. Uma abordagem simples é permitir que o servidor  $S_k$  mantenha um relógio vetorial de tempo real  $RVC_k$  onde  $RVC_k[i] = T(i)$  significa que  $S_k$  viu todas as escritas que foram apresentadas a  $S_i$  até o tempo  $T(i)$ . Nesse caso, consideramos que cada escrita apresentada transporta uma marca de tempo atribuída por seu servidor de origem e que  $T(i)$  é a hora *local* para  $S_i$ .

Se os relógios entre os servidores de réplicas estiverem fracamente sincronizados, um protocolo aceitável para limitar idade seria o seguinte: sempre que o servidor  $S_k$  notar que  $T(k) - RVC_k[i]$  está perto de exceder um limite especificado, ele simplesmente começa a recuperar escritas que se originaram de  $S_i$  que transportem uma marca de tempo mais tardia do que  $RVC_k[i]$ .

Note que, nesse caso, um servidor de réplicas é responsável por manter sua cópia de  $x$  atualizada em relação a escritas que foram emitidas em outros lugares. Ao contrário, quando mantínhamos limites numéricos, seguimos uma abordagem de envio permitindo que um servidor de origem mantivesse réplicas atualizadas por meio

do repasse de escritas. O problema de enviar escritas no caso de idade é que não se pode dar nenhuma garantia de consistência quando não se sabe de antemão qual será o tempo máximo de propagação. Essa situação melhora um pouco com a recuperação de atualizações, porque múltiplos servidores podem ajudar a manter renovada (atualizada) uma cópia de  $x$  de um servidor.

### Limitação de desvio de ordenação

Lembre-se de que desvios de ordenação em consistência contínua são causados pelo fato de que um servidor de réplicas aplica provisoriamente atualizações que lhe foram apresentadas. O resultado é que cada servidor terá uma fila local de escritas provisórias cuja ordem real em que devem ser aplicadas à cópia local de  $x$  ainda precisa ser determinada. O desvio de ordenação é limitado pela especificação de um comprimento máximo da fila de escritas provisórias.

Em decorrência, é simples detectar quando é preciso impor consistência de ordenação: quando o comprimento dessa fila local exceder um comprimento máximo especificado. Nesse ponto, um servidor não aceitará mais nenhuma nova escrita apresentada, mas tentará comprometer escritas provisórias negociando com outros servidores a ordem em que suas escritas devem ser executadas. Em outras palavras, é preciso impor uma ordenação de escritas provisórias globalmente consistente. Há muitos modos de fazer isso, mas ocorre que, na prática, são utilizados os assim chamados protocolos baseados em primários ou em quórum. Discutiremos esses protocolos a seguir.

### 7.5.2 Protocolos baseados em primários

Na prática vemos que, de modo geral, aplicações distribuídas seguem modelos de consistência relativamente fáceis de entender. Entre esses modelos estão os de limitação de desvios de idade e, em menor medida, também os que limitam desvios numéricos. Quando se trata de modelos que manipulam ordenação consistente de operações, temos os modelos de consistência seqüencial. Em particular, são populares aqueles em que as operações podem ser agrupadas por travas ou transações.

Assim que os modelos de consistência ficam um pouco mais difíceis de entender pelos desenvolvedores de aplicações, vemos que são ignorados, ainda que o desempenho pudesse ser melhorado. O resultado líquido é que, se a semântica de um modelo de consistência não for intuitivamente clara, os desenvolvedores de aplicações terão grandes dificuldades para construir aplicações corretas. A simplicidade é apreciada (e talvez isso seja justificável).

No caso de consistência seqüencial prevalecem os protocolos baseados em primários. Nesses protocolos, cada item de dados  $x$  no depósito de dados tem um primário associado, que é responsável por coordenar operações de escrita em  $x$ . Pode-se fazer uma distinção conforme o

primário seja fixo em um servidor remoto ou se as operações de escrita puderem ser realizadas no local depois de mover o primário para o processo em que a operação de escrita é iniciada. Vamos estudar essa classe de protocolos.

### Protocolos de escrita remota

O protocolo mais simples baseado em primário e que suporta replicação é aquele em que as operações de escrita precisam ser enviadas para um único servidor fixo. Operações de leitura podem ser executadas no local. Esses esquemas também são conhecidos como **protocolos de primário e backup** (Budhiraja et al., 1993). Um protocolo de primário e backup funciona como mostra a Figura 7.19. Um processo que quer realizar uma operação de escrita, em um item de dados  $x$ , envia essa operação para o servidor de primários para  $x$ . O servidor primário executa a atualização em sua cópia local de  $x$  e, na sequência, envia a atualização para os servidores de backup. Cada servidor de backup também efetua a atualização e envia um reconhecimento de volta ao servidor primário. Quando todos os servidores de backup tiverem atualizado sua cópia local, o servidor primário envia um reconhecimento de volta ao processo inicial.

Um problema potencial de desempenho desse esquema é que pode levar um tempo relativamente longo antes que o processo que iniciou a atualização tenha permissão para continuar. Na verdade, uma atualização é implementada como uma operação de bloqueio. Uma alternativa é usar uma abordagem não bloqueadora. Tão logo o servidor primário tenha atualizado sua cópia local de  $x$ , ele retorna um reconhecimento. Depois disso, diz ao servidor primário de backup que também efetue a atualização. Protocolos de primário-backup não bloqueadores são discutidos em Budhiraja e Marzullo (1992).

O principal problema de protocolos de primário-backup não bloqueadores tem a ver com tolerância a falha. Com um sistema bloqueador, o processo cliente sabe, com certeza, que a operação de atualização é apoiada por vários outros servidores. Isso não ocorre com uma solução não bloqueadora. A vantagem, é claro, é que operações de escrita podem ser consideravelmente aceleradas. Voltaremos às questões de tolerância a falha no próximo capítulo.

Protocolos de primário e backup proporcionam uma implementação direta de consistência seqüencial porque o servidor primário pode ordenar todas as escritas que entram em uma ordem temporal globalmente exclusiva. Evidentemente, todos os processos vêm todas as operações de escrita na mesma ordem, sem importar qual servidor de backup utilizem para efetuar operações de leitura. Além disso, com protocolos bloqueadores, os processos sempre verão os efeitos de sua operação de escrita mais recente (note que isso não pode ser garantido com um protocolo não bloqueador sem adotar medidas especiais).

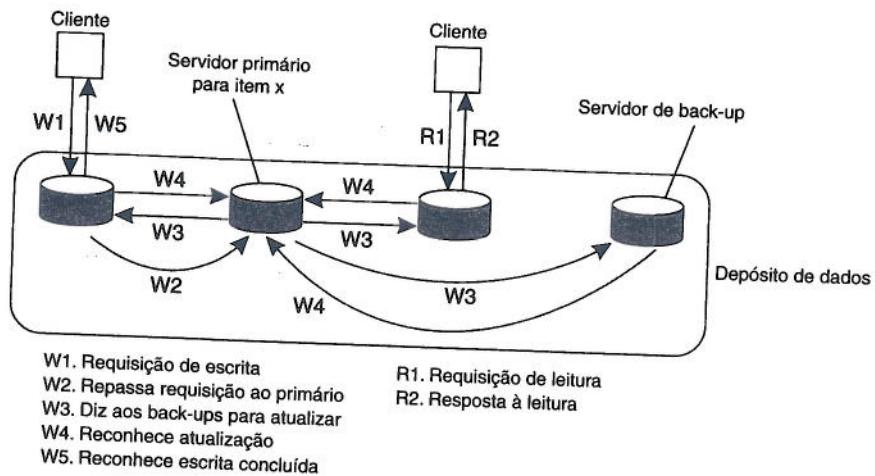


Figura 7.19 Princípio de um protocolo de primário e backup.

### Protocolos de escrita local

Uma variante dos protocolos de primário e backup é aquela em que a cópia primária migra entre processos que desejam realizar uma operação de escrita. Como antes, sempre que um processo quer atualizar o item de dados  $x$ , ele localiza a cópia primária de  $x$  e, na seqüência, move essa cópia para sua própria localização, como mostra a Figura 7.20. A principal vantagem dessa abordagem é que múltiplas operações sucessivas de escrita podem ser executadas no local enquanto processos leitores ainda podem acessar sua cópia local. Contudo, só se pode conseguir tal melhoria se for seguido um protocolo não bloqueador pelo qual as atualizações são propagadas para as réplicas após o servidor primário ter concluído as atualizações realizadas localmente.

Esse protocolo de escrita local de primário e backup também pode ser aplicado a computadores móveis que são capazes de operar em modo desconectado. Antes de

desconectar, o computador móvel torna-se o servidor primário para cada item de dados que ele espera atualizar. Enquanto está desconectado, todas as operações de atualização são executadas localmente, ao mesmo tempo que outros processos ainda podem realizar operações de leitura (mas não atualizações). Mais tarde, quando se conectar novamente, as atualizações são propagadas do primário para os backups, o que leva o depósito de dados novamente a um estado consistente. Voltaremos à operação em modo desconectado no Capítulo 11, quando discutirmos sistemas de arquivos distribuídos.

Como uma última variante desse esquema, protocolos não bloqueadores de escrita local baseados em primários também são usados para sistemas de arquivos distribuídos em geral. Nesse caso, pode haver um servidor central fixo por meio do qual normalmente ocorrem todas as operações de escrita, como no caso de escrita remota no esquema de primário e backup. Contudo, o servidor

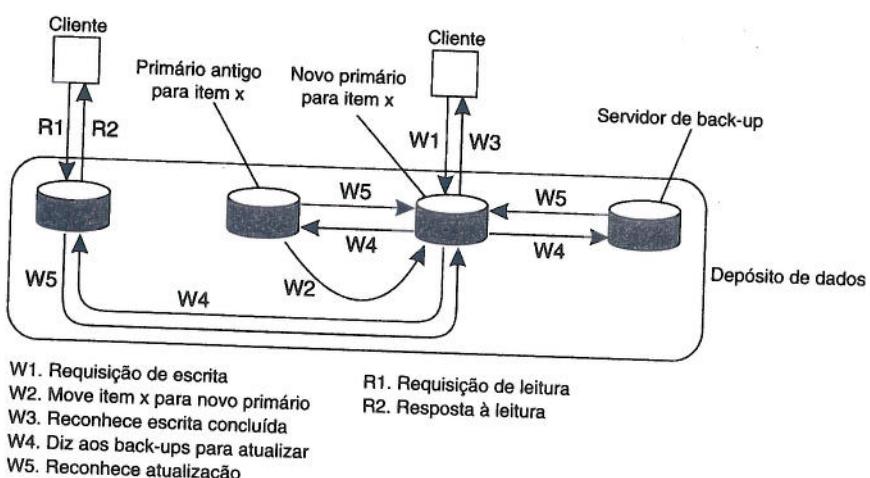


Figura 7.20 Protocolo de primário e backup no qual a cópia primária migra para o processo que quer realizar uma atualização.

permite, temporariamente, que uma das réplicas execute uma série de atualizações locais porque isso pode acelerar consideravelmente o desempenho. Quando o servidor de réplicas concluir, as atualizações são propagadas para o servidor central, a partir do qual elas são distribuídas para os outros servidores de réplicas.

### 7.5.3 Protocolos de escrita replicada

Em protocolos de escrita replicada, operações de escrita podem ser executadas em várias réplicas em vez de em só uma, como no caso de réplicas baseadas em primários. Pode-se fazer uma distinção entre replicação ativa, na qual uma operação é repassada para todas as réplicas, e protocolos de consistência baseados em voto majoritário.

#### Replicação ativa

Em replicação ativa, cada réplica tem um processo associado que realiza as operações de atualização. Ao contrário de outros protocolos, de modo geral, as atualizações são propagadas por meio da operação de escrita que causa a atualização. Em outras palavras, a operação é enviada a cada réplica. Contudo, também é possível enviar a atualização, como discutimos antes.

Um problema da replicação ativa é que as operações precisam ser executadas na mesma ordem em todos os lugares. Por isso, é preciso um mecanismo de multicast totalmente ordenado. Tal multicast pode ser implementado com o uso de relógios lógicos de Lamport, como discutimos no capítulo anterior. Infelizmente, essa implementação de multicast torna-se muito complexa em grandes sistemas distribuídos. Como alternativa, pode-se conseguir ordenação total usando um coordenador central, também denominado sequenciador. Uma abordagem é primeiro repassar cada operação ao sequenciador, que lhe designa um número de seqüência exclusivo, e, logo depois, enviar a operação para todas as réplicas. As operações são executadas na ordem de seu número de seqüência. Claro que essa implementação de multicast totalmente ordenado é muito parecida com protocolos de consistência baseados em primários.

Note que utilizar um sequenciador não resolve o problema de escalabilidade. Na verdade, se for preciso multicast totalmente ordenado, talvez seja necessária uma combinação de multicast simétrico usando marcas de tempo de Lamport e sequenciadores. Tal solução é descrita em Rodrigues et al. (1996).

#### Protocolos baseados em quórum

Uma abordagem diferente para suportar escritas replicadas é usar votação como proposto originalmente por Thomas (1979) e generalizado por Gifford (1979). A idéia básica é exigir que clientes requisitem e adquiram a

permissão de vários servidores antes de ler ou escrever um item de dados replicado.

Como exemplo simples do modo de funcionamento do algoritmo, considere um sistema de arquivos distribuídos e suponha que um arquivo é replicado em  $N$  servidores. Poderíamos criar uma regra determinando que, para atualizar um arquivo, em primeiro lugar um cliente deve contatar no mínimo metade dos servidores mais um (maioria simples) e conseguir que eles concordem em fazer a atualização. Tão logo concordem, o arquivo é alterado e um novo número de versão é associado com o novo arquivo. O número de versão é usado para identificar a versão do arquivo e é o mesmo para todos os arquivos recém-atualizados.

Para ler um arquivo replicado, um cliente também deve contatar no mínimo metade dos servidores mais um e solicitar que eles enviem os números das versões associadas com o arquivo. Se todos os números de versão forem iguais, essa deve ser a versão mais recente, porque uma tentativa de atualizar somente os servidores restantes falharia, já que não há servidores suficientes.

Por exemplo, se houver cinco servidores e um cliente determina que três deles têm a versão 8, é impossível que os outros dois tenham a versão 9. Afinal, qualquer atualização bem-sucedida da versão 8 para a versão 9 requer conseguir que três servidores concordem com isso, e não apenas dois.

Na realidade, o esquema de Gifford é um pouco mais abrangente que isso. Segundo esse esquema, para ler um arquivo do qual existem  $N$  réplicas, um cliente precisa conseguir um **quórum de leitura**, um conjunto arbitrário de quaisquer  $N_R$  servidores, ou mais. De maneira semelhante, para modificar um arquivo, é exigido um **quórum de escrita** de, no mínimo,  $N_W$  servidores. Os valores de  $N_R$  e  $N_W$  estão sujeitos às duas restrições seguintes:

1.  $N_R + N_W \leq N$
2.  $N_W \leq N/2$

A primeira restrição é usada para evitar conflitos leitura-escrita, enquanto a segunda impede conflitos escrita-escrita. Somente após o número adequado de servidores ter concordado em participar é que um arquivo pode ser lido ou escrito.

Para ver como esse algoritmo funciona, considere a Figura 7.21(a), na qual  $N_R = 3$  e  $N_W = 10$ . Imagine que o quórum de escrita mais recente consistiu em 10 servidores, C a L. Todos eles obtêm a nova versão e o novo número de versão. Qualquer quórum de leitura subsequente de três servidores terá de conter, no mínimo, um membro desse conjunto. Quando o cliente vir os números de versão, saberá qual é a mais recente e a adotará.

Na Figura 7.21(b) e (c), vemos mais dois exemplos. Na Figura 7.21(b) pode ocorrer um conflito escrita-escrita porque  $N_W \leq N/2$ . Em particular, se um dos clientes escolher {A,B,C,E,F,G} como seu conjunto de escrita e

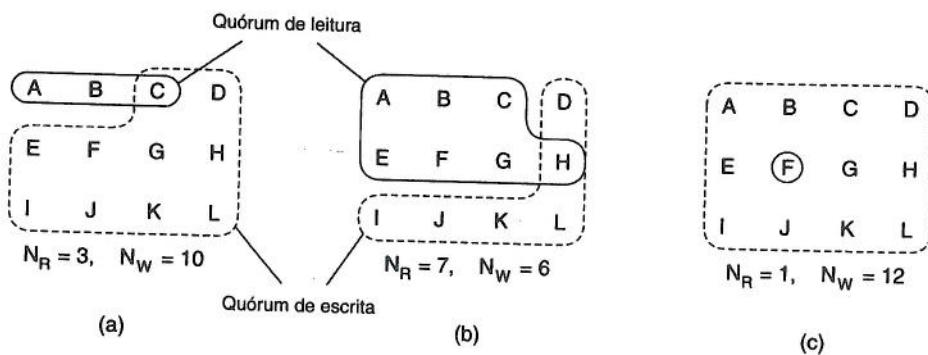


Figura 7.21 Três exemplos do algoritmo de votação. (a) Escolha correta de conjunto de leitura e de escrita. (b) Escolha que pode levar a conflitos escrita–escrita. (c) Escolha correta, conhecida como ROWA (lê uma, escreve todas).

um outro cliente escolher  $\{D,H,I,J,K,L\}$  como seu conjunto de escrita, então estaremos claramente em dificuldades porque ambas as atualizações serão aceitas sem detectar que, na verdade, estão em conflito.

A situação mostrada na Figura 7.21(c) é de especial interesse porque fixa  $N_R$  em um, o que possibilita ler um arquivo replicado descobrindo e usando qualquer cópia. O preço pago por esse bom desempenho de leitura, entretanto, é que as atualizações de escrita precisam adquirir todas as cópias. Esse esquema é geralmente denominado **lê uma, escreve todas** (Read-One, Write-All — ROWA). Há diversas variações de protocolos de replicação baseados em quórum. Uma boa visão geral é apresentada em Jalote (1994).

### 7.5.4 Protocolos de coerência de cache

Caches são um caso especial de replicação, no sentido de que, em geral, são controladas por clientes, em vez de servidores. Contudo, protocolos de coerência de cache, que garantem que uma cache é consistente com as réplicas iniciadas por servidor, em princípio, não são muito diferentes dos protocolos de consistência discutidos até aqui.

Tem havido muita pesquisa nas áreas de projeto e implementação de caches, em especial no contexto de sistemas multiprocessadores de memória compartilhada. Muitas soluções são baseadas em suporte de hardware subjacente, por exemplo, considerando que podem ser feitos escuta ou broadcast eficiente. No contexto de sistemas distribuídos baseados em middleware que são construídos com base em sistemas operacionais de uso geral, soluções para caches baseadas em software são mais interessantes. Nesse caso, quase sempre usam-se dois critérios separados para classificar protocolos de cache (Min e Baer, 1992; Lilja, 1993; Tartalja e Milutinovic, 1997).

Em primeiro lugar, as soluções de cache podem ser diferentes quanto à **estratégia de detecção de coerência**, isto é, quando as inconsistências são realmente detectadas. Em soluções estáticas, a premissa adotada é que um compilador realize a análise necessária anterior à execução e determine quais dados podem realmente levar a

inconsistências porque podem ser colocados em cache. O compilador simplesmente insere instruções que evitam inconsistências. Nos sistemas distribuídos que estudamos neste livro normalmente são aplicadas soluções dinâmicas. Nessas soluções, as inconsistências são detectadas em tempo de execução. Por exemplo, é feita uma verificação no servidor para ver se os dados em cache foram modificados desde que entraram na cache.

No caso de bancos de dados distribuídos, os protocolos baseados em detecção dinâmica ainda podem ser classificados considerando exatamente em que ponto de uma transação a detecção é feita. Franklin et al. (1997) distinguem os três casos seguintes. No primeiro, quando um item de dados em cache é acessado durante uma transação, o cliente precisa verificar se esse item de dados ainda é consistente com a versão armazenada no servidor (possivelmente replicado). A transação não pode prosseguir e usar a versão em cache até que sua consistência tenha sido definitivamente validada.

Na segunda abordagem, a otimista, a transação pode prosseguir enquanto ocorre a verificação. Nesse caso, a premissa adotada é que os dados em cache estavam atualizados quando a transação começou. Se, mais tarde, essa premissa mostrar ser falsa, a transação terá de ser abortada.

A terceira abordagem é verificar se os dados em cache estão atualizados somente quando a transação for comprometida. Essa abordagem é comparável ao esquema otimista de controle de concorrência discutido no capítulo anterior. Na verdade, a transação apenas inicia a operação nos dados em cache e espera que o melhor aconteça. Depois que todo o trabalho foi re-alizado, é verificada a consistência dos dados acessados. Quando forem usados dados antigos, a transação é abortada.

Uma outra questão de projeto para protocolos de coerência de cache é a **estratégia de imposição de coerência**, que determina *como* as caches são mantidas consistentes com as cópias armazenadas em servidores. A solução mais simples é não permitir que dados compartilhados sejam colocados em cache. Em vez disso, dados

compartilhados são guardados somente nos servidores, que mantêm consistência usando um dos protocolos baseados em primários ou de replicação de escrita que já discutimos. Clientes só têm permissão de colocar em cache seus próprios dados privados. É óbvio que essa solução pode oferecer melhorias de desempenho apenas limitadas.

Quando dados compartilhados podem ser colocados em cache, há duas abordagens para impor coerência de cache. A primeira é permitir que um servidor envie uma invalidação a todas as caches sempre que um item de dados for modificado. A segunda é simplesmente propagar a atualização. A maioria dos sistemas de cache usa um desses dois esquemas. Escolha dinâmica entre enviar invalidações ou atualizações às vezes é utilizada em bancos de dados cliente-servidor (Franklin et al., 1997).

Por fim, precisamos considerar também o que acontece quando um processo modifica dados em cache. Quando são usadas caches somente de leitura, operações de atualização só podem ser realizadas por servidores que, na sequência, sigam algum protocolo de distribuição para garantir que as atualizações sejam propagadas para as caches. Em muitos casos é seguida uma abordagem de recuperação de atualizações. Nesse caso, um cliente detecta que sua cache tem itens de dados antigos e requisita uma atualização a um servidor.

Uma abordagem alternativa é permitir que clientes modifiquem diretamente os dados em cache e enviem a atualização aos servidores. Essa abordagem é seguida em **caches de escrita direta**, que costumam ser usadas em sistemas de arquivos distribuídos. Na verdade, a cache de escrita direta é semelhante a um protocolo de escrita local baseado em primários no qual a cache do cliente se tornou um servidor primário temporário. Para garantir consistência (sequencial) é necessário que o cliente tenha recebido permissões exclusivas de escrita, senão podem ocorrer conflitos escrita-escrita.

Caches de escrita direta oferecem, potencialmente, melhor desempenho em comparação com outros esquemas, porque todas as operações podem ser executadas no local. Podem-se conseguir mais melhorias se retardarmos a propagação de atualizações ao permitir que ocorram múltiplas escritas antes de informar aos servidores. Isso resulta no que é conhecido como **cache de escrita retroativa**, que, mais uma vez, é aplicada principalmente em sistemas de arquivos distribuídos.

### 7.5.5 Implementação de consistência centrada no cliente

Como último tópico sobre protocolos de consistência, vamos dirigir nossa atenção à implementação de consistência centrada no cliente. Implementar consistência centrada no cliente é algo relativamente direto se forem ignoradas questões de desempenho. Nas páginas seguintes, em pri-

meiro lugar descrevemos tal implementação e, em seguida, descrevemos uma implementação mais realista.

#### Implementação ingênua

Em uma implementação ingênua de consistência centrada no cliente, a cada operação de escrita  $W$  é designado um identificador globalmente exclusivo. Tal identificador é designado pelo servidor ao qual a escrita foi apresentada. Como no caso da consistência contínua, referimo-nos a esse servidor como a origem de  $W$ . Então, monitoramos dois conjuntos de escritas para cada cliente. O conjunto de leitura para um cliente consiste nas escritas relevantes para as operações de leitura executadas por esse cliente. Da mesma maneira, o conjunto de escrita consiste nas escritas (identificadores das escritas) realizadas pelo cliente.

Consistência de leitura monotônica é implementada como descrevemos a seguir. Quando um cliente realiza uma operação de leitura em um servidor, esse servidor recebe o conjunto de leitura do cliente para verificar se todas as escritas identificadas ocorreram localmente. (O tamanho de tal conjunto pode introduzir um problema de desempenho, para o qual discutiremos uma solução mais adiante.) Se nem todas as leituras ocorreram localmente, ele contata os outros servidores para garantir que ele seja atualizado antes de realizar a operação de leitura. Como alternativa, a operação de leitura é repassada para um servidor no qual as operações de escrita já ocorreram. Após a realização da operação de leitura, as operações de escrita que ocorreram no servidor selecionado e que são relevantes para a operação de leitura são adicionadas ao conjunto de leitura do cliente.

Note que tem de ser possível determinar exatamente onde ocorreram as operações de escrita identificadas no conjunto de leitura. Por exemplo, o identificador de escrita poderia incluir o identificador do servidor ao qual as operações foram apresentadas. Esse servidor deve, por exemplo, registrar a operação de escrita de modo que ela possa ser reproduzida em um outro servidor. Além disso, operações de escrita devem ser executadas na ordem em que foram apresentadas. A ordenação pode ser conseguida ao permitir que o cliente gere um número de sequência globalmente exclusivo que é incluído no identificador de escrita. Se cada item de dados puder ser modificado somente por seu proprietário, este pode fornecer o número de sequência.

Consistência de escrita monotônica é implementada de modo análogo ao das leituras monotônicas. Sempre que um cliente inicia uma nova operação de escrita em um servidor, o servidor recebe o conjunto de escrita do cliente. (Mais uma vez, o tamanho do conjunto pode ser proibitivamente grande em face dos requisitos de desempenho. Uma solução alternativa será discutida mais adiante.) Portanto, ele assegura que as operações de escrita identifica-

das sejam realizadas antes e na ordem correta. Após executar a nova operação, o identificador de escrita daquela operação é adicionado ao conjunto de escrita. Note que atualizar o servidor corrente com o conjunto de escrita do cliente pode introduzir considerável aumento no tempo de resposta do cliente, uma vez que, nesse caso, o cliente tem de esperar que a operação seja totalmente concluída.

Da mesma maneira, consistência leia-susas-escritas requer que o servidor no qual a operação de leitura é executada tenha visto todas as operações de escrita no conjunto de escrita do cliente. As escritas podem ser simplesmente buscadas em outros servidores antes da execução da operação de leitura, embora isso possa resultar em um tempo de resposta pobre. Como alternativa, o software do lado do cliente pode procurar um servidor no qual as operações de escrita identificadas no conjunto de escrita do cliente já foram executadas.

Por fim, consistência escritas-seguem-leituras pode ser implementada primeiro com a atualização do servidor selecionado com as operações de escrita no conjunto de leitura do cliente e, então, mais tarde, com a adição do identificador da operação de escrita ao conjunto de escrita, junto com os identificadores no conjunto de leitura (que, agora, se tornam relevantes para a operação de escrita que acabou de ser executada).

### Como melhorar a eficiência

É fácil observar que o conjunto de leitura e o conjunto de escrita associados com cada cliente podem se tornar muito grandes. Para manter a gerenciabilidade desses conjuntos, as operações de leitura e de escrita de um cliente são agrupadas em sessões. Uma sessão normalmente está associada com uma aplicação: ela é aberta quando a aplicação começa e fechada quando ela sai. Contudo, sessões também podem ser associadas com aplicações que saíram temporariamente, como agentes de usuários para e-mail. Sempre que um cliente fecha uma sessão, os conjuntos são simplesmente removidos. Claro que, se um cliente abrir uma sessão e nunca fechá-la, os conjuntos de leitura e escrita associados ainda podem se tornar muito grandes.

O problema principal da implementação ingênuo se encontra na representação dos conjuntos de leitura e escrita. Cada conjunto consiste em uma quantidade de identificadores para operações de escrita. Sempre que um cliente envia uma requisição de leitura ou de escrita para um servidor, o servidor também recebe um conjunto de identificadores para que ele veja se todas as operações de escrita relevantes para a requisição foram executadas por esse servidor.

Essa informação pode ser representada com mais eficiência por meio de marcas de tempo vetoriais da maneira que descreveremos a seguir. Em primeiro lugar, sempre que um servidor aceita uma nova operação de escrita  $W$ , ele designa a essa operação um identificador globalmente

exclusivo junto com uma marca de tempo  $ts(W)$ . Uma operação de escrita subsequente apresentada àquele servidor recebe uma marca de tempo de valor mais alto. Cada servidor  $S_i$  mantém uma marca de tempo vetorial  $WVC_i$ , onde  $WVC_i[j]$  é igual à marca de tempo da mais recente operação de escrita que se originou de  $S_j$  e que foi processada por  $S_i$ . Por questão de clareza, considere que, para cada servidor, escritas que vêm de  $S_j$  são processadas na ordem em que foram apresentadas.

Sempre que um cliente emite uma requisição para executar uma operação de leitura ou de escrita  $O$  em um servidor específico, esse servidor retorna sua marca de tempo corrente com os resultados de  $O$ . Na sequência, conjuntos de leitura e de escrita são representados por marcas de tempo vetoriais. Mais especificamente, para cada sessão  $A$ , construímos uma marca de tempo vetorial  $SVC_A$  com  $SVC_A[j]$  igual à máxima marca de tempo de todas as operações de escrita em  $A$  que se originam no servidor  $S_i$ :

$$SVC_A[j] = \max\{ ts(W) \mid W \in A \text{ & } \text{origem}(W) = S_j \}$$

Em outras palavras, a marca de tempo de uma sessão sempre representa as últimas operações de escrita que foram vistas pelas aplicações em execução como parte dessa sessão. A compactação é obtida representando todas as operações de escrita observadas que se originaram do mesmo servidor por uma única marca de tempo.

Como exemplo, suponha que um cliente, como parte da sessão  $A$ , registre-se em um servidor  $S_i$ . Com essa finalidade, ele passa  $SVC_A$  para  $S_i$ . Considere que  $SVC_A[j] = WVC_i[j]$ . Isso significa que  $S_i$  ainda não viu todas as escritas que se originaram de  $S_j$  que o cliente viu. Dependendo da consistência requerida, agora o servidor  $S_i$  pode ter de buscar essas escritas antes de poder se reportar consistentemente ao cliente. Tão logo a operação seja realizada, o servidor  $S_i$  retornará sua marca de tempo corrente  $WVC_i$ . Nesse ponto,  $SVC_A$  é ajustada para:

$$SVC_A[j] \leftarrow \max\{ SVC_A[j], WVC_i[j] \}$$

Mais uma vez, vemos como marcas de tempo vetoriais podem proporcionar um modo elegante e compacto de representar históricos em um sistema distribuído.

## 7.6 Resumo

Há duas razões primordiais para replicar dados: melhorar a confiabilidade de um sistema distribuído e melhorar desempenho. A replicação introduz um problema de consistência: sempre que uma réplica é atualizada, ela se torna diferente das outras. Para manter as réplicas consistentes, precisamos propagar atualizações de tal modo que inconsistências temporárias não sejam notadas. Infelizmente, fazer isso degrada seriamente o desempenho, em especial em sistemas distribuídos de grande porte.

A única solução para esse problema é relaxar um pouco a consistência. Existem diferentes modelos de consistência. Para consistência contínua, a meta é estabelecer limites para o desvio numérico entre réplicas, para o desvio entre idades e para desvios entre as ordenações de operações.

Desvio numérico refere-se ao valor da diferença entre réplicas que pode ser tolerado. Esse tipo de desvio é muito dependente de aplicação mas pode, por exemplo, ser usado na replicação de valores de ações. O desvio de idade se refere ao tempo durante o qual uma réplica ainda é considerada consistente, embora as atualizações possam ter ocorrido há algum tempo. O desvio de idade costuma ser usado para caches Web. Por fim, o desvio de ordenação se refere ao número máximo de escritas provisórias que podem ficar pendentes em qualquer servidor sem ter sido sincronizadas com os outros servidores de réplicas.

Há muito tempo que a ordenação consistente de operações forma a base de muitos modelos de consistência. Existem muitas variações, mas parece que apenas algumas predominam entre os desenvolvedores de aplicações. Em essência, a consistência seqüencial fornece a semântica que os programadores esperam em programação concorrente: todas as operações de escrita são vistas por todos na mesma ordem. Menos usada, mas ainda assim relevante, é a consistência causal, que reflete que as operações que são potencialmente dependentes umas das outras sejam executadas na ordem dessa dependência.

Modelos de consistência mais fraca consideram séries de operações de leitura e de escrita. Em particular, eles consideram que cada série é adequadamente ‘limitada’ por operações conjugadas executadas em variáveis de sincronização, como travas. Embora isso requeira esforço explícito dos programadores, de modo geral esses modelos são mais fáceis de implementar com eficiência do que, por exemplo, consistência seqüencial pura.

Ao contrário desses modelos centrados em dados, os pesquisadores da área de bancos de dados distribuídos para usuários móveis definiram vários modelos de consistência centrada no cliente. Esses modelos não consideram o fato de que os dados podem ser compartilhados por diversos usuários, mas se concentram na consistência que deve ser oferecida a um cliente individual. A premissa subjacente é que um cliente se conecte com réplicas diferentes ao longo do tempo, mas que essas diferenças sejam transparentes. Em essência, modelos de consistência centrados no cliente garantem que, sempre que um cliente se conectar com uma nova réplica, essa réplica seja atualizada com os dados que tinham sido manipulados por aquele cliente antes e que possivelmente residam em outros sites de réplicas.

Técnicas diferentes podem ser aplicadas para propagar atualizações. É preciso fazer uma distinção no que diz respeito a o que é exatamente propagado, onde as atualizações são propagadas e por quem a propagação é iniciada. Podemos decidir propagar notificações, operações ou estado. Da mesma maneira, não são todas as réplicas que sempre precisam ser atualizadas imediatamente. Qual das réplicas é atualizada, e quando, depende do protocolo de distribuição. Por fim, pode-se escolher entre enviar atualizações para outras réplicas ou recuperar atualizações de outras réplicas.

Protocolos de consistência descrevem implementações específicas de modelos de consistência. No que diz respeito à consistência seqüencial e suas variantes, pode-se fazer uma distinção entre protocolos baseados em primários e protocolos de escrita replicada. Em protocolos baseados em primários, todas as operações de atualização são repassadas para uma cópia primária que, na seqüência, garante que a atualização seja adequadamente ordenada e repassada. Em protocolos de escrita replicada, uma atualização é repassada a diversas réplicas ao mesmo tempo. Nesse caso, a ordenação correta das operações costuma ficar mais difícil.

## Problemas

1. Acessos a objetos Java compartilhados podem ser serializados declarando seus métodos como sincronizados. Isso é suficiente para garantir serialização quando tal objeto é replicado?
2. Explique, com suas próprias palavras, qual é a principal razão para considerar modelos de consistência fraca.
3. Explique como ocorre a replicação em DNS e por que, na verdade, ela funciona tão bem.
4. Durante a discussão de modelos de consistência, referimo-nos freqüentemente ao contrato entre o software e o depósito de dados. Por que tal contrato é necessário?
5. Dadas as réplicas na Figura 7.2, o que precisaria ser feito para terminar os valores na contagem de modo que ambos, A e B, vejam o mesmo resultado?
6. Na Figura 7.7, 001110 é uma saída legal para uma memória seqüencialmente consistente? Explique sua resposta.
7. Costuma-se argumentar que modelos de consistência fraca impõem uma carga extra aos programadores. Até que ponto essa declaração é verdadeira?
8. O multicast totalmente ordenado por meio de um seqüenciador e por questão de consistência em replicação ativa viola o argumento fim-a-fim no projeto de sistemas?
9. Que tipo de consistência você usaria para implementar um mercado eletrônico de ações? Explique sua resposta.

10. Considere uma caixa postal pessoal para um usuário móvel, implementada como parte de um banco de dados distribuído de longa distância. Que tipo de consistência centrada no cliente seria mais adequado?
11. Descreva uma implementação simples de consistência leia-suas-escritas para apresentar páginas Web que acabaram de ser atualizadas.
12. Para simplificar as coisas, consideramos que não havia conflitos escrita-escrita no Bayou. Claro que essa premissa não é realista. Explique como podem acontecer conflitos.
13. Quando se usa um leasing, é necessário que os relógios de um cliente e do servidor, respectivamente, estejam fortemente sincronizados?
14. Afirmamos que multicast totalmente ordenado que utiliza relógios lógicos de Lamport não é escalável. Explique por quê.
15. Mostre que, no caso de consistência contínua, fazer com que um servidor  $S_k$  adiante sua visão  $TW_k(i,k)$  sempre que receber uma atualização renovada que aumentaria  $TW(k,k) - TW_k(i,k)$  para além de  $\delta_i/(N-1)$  garante que  $v(t) - v_i \leq \delta_i$ .
16. No caso da consistência contínua, consideramos que cada escrita só aumenta o valor do item de dados  $x$ . Elabore uma solução na qual também é possível reduzir o valor de  $x$ .
17. Considere um protocolo de primário e backup não bloqueador usado para garantir consistência seqüencial em um depósito de dados distribuído. Esse depósito de dados sempre fornece consistência leia-suas-escritas?
18. Para que a replicação ativa funcione de modo geral, é necessário que todas as operações sejam executadas na mesma ordem em cada réplica. Essa ordenação é sempre necessária?
19. Para implementar multicast totalmente ordenado por meio de um seqüenciador, uma abordagem seria primeiro repassar uma operação ao seqüenciador, que então lhe designaria um número exclusivo e, na seqüência, faria multicast da operação. Cite duas abordagens alternativas e compare as três soluções.
20. Um arquivo é replicado em dez servidores. Faça uma lista de todas as combinações de quórum de leitura e quórum de escrita que são permitidas pelo algoritmo de votação.
21. Leasings baseados em estado são usados para aliviar a carga de um servidor permitindo que ele concorde em monitorar o menor número possível de clientes. Essa abordagem resultará necessariamente em melhor desempenho?
22. (Tarefa de laboratório) Neste exercício, você deve implementar um sistema simples que suporte multicast RPC. Vimos que há vários servidores replicados e que cada cliente se comunica com um servidor por meio de uma RPC. Contudo, quando se tratar de replicação, um cliente precisará enviar uma requisição RPC a cada réplica. Programe o cliente de modo tal que, para a aplicação, pareça ser enviada uma única RPC. Considere que você está replicando por desempenho, mas que os servidores são suscetíveis a falhas.