INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

LICENCIATURA EM ENGENHARIA DE INFORMÁTICA E COMPUTADORES

SEMESTRE DE VERÃO 2011/2012

Sistemas Distribuídos

Optimistic Data Replication

Grupo:

- G05D

Alunos:

- João Pires 31933
- José Casimiro 32713
- António Dente 33168

Índice

Introdução 3
Replicação Pessimista 3
Replicação Otimista 3
Escolhas de desenho 4
Complexidade e Eficiência 4
Concorrência 5
Eficiência 5
Qualidade 5
Deteção de relações de concorrência e "Happens-Before" 6
Algoritmos de deteção e representação de relações *"happens-before"* 6
Controlo de concorrência e consistência eventual 7
Schedulling 7
Deteção de conflitos 7
Resolução de conflitos 8
Protocolos de compromisso 8
Propagação 9
Propagação de operações 9
Propagação de estado 9
Transferência híbrida 9
Técnicas de transferência por *push* 10
Divergência entre réplicas 10
Ordenação de leituras/escritas 10
Garantias de sessão 10
Limitar as divergências entre réplicas 10
Referências 11

Introdução

Em quase todos os sistemas distribuídos, existe algum tipo de *data* ou informação que é necessário partilhar por vários servidores. Estejam estes armazenados em base de dados ou em memória volátil é imperativo que se mantenham consistentes e coerentes independentemente do local onde são acedidos.

Aqui entra a replicação de dados. Mantendo múltiplas cópias de informação em servidores separados. Deste modo pode-se aceder a um servidor mais próximo de nós mesmo que a informação tenha originado do outro lado do globo, melhorando a velocidade de receção e facilitando a alteração de informação, caso esta funcionalidade seja permitida.

Existem duas técnicas base de replicação de dados, replicação pessimista e otimista. Neste relatório iremos dar uma pequena explicação da replicação pessimista embora nos vamos focar na otimista.

Replicação Pessimista

Numa replicação pessimista (tradicional) tenta-se criar a ilusão de que apenas uma cópia dos dados existe, por exemplo, apenas existe uma base de dados. Para criar esta ilusão não se permite acesso a qualquer informação que não seja provada como a mais atual. Para tal recorre-se a técnicas de *multi locks* e sempre que é realizada uma alteração a um dado esta é replicada para todas as cópias. Daí sere chamada de pessimista.

Esta abordagem, de facto, garante uma grande consistência nos dados mas é irrealista num cenário *web*. Embora em cenários *lan* seja possível a sua concretização, tendo em conta baixos valores de latência e pouca probabilidade de falha mecânica, num cenário *web* estes pressupostos não podem ser garantidos.

Replicação Otimista

As técnicas de replicação otimista, em oposição à pessimista, pressupõem que, qualquer erro ou falha, irá ocorrer poucas vezes, se alguma.

Com base neste pressuposto, uma técnica otimista, permite que os dados sejam acedidos mesmo que não sejam comprovados como os mais atuais. Isto permite um ambiente de trabalho muito mais fluido e “isolado”. Deste modo podemos garantir uma melhor disponibilidade de informação ao custo do ocasional conflito, visto um dado poder ser alterado por duas pessoas diferentes em duas réplicas distintas.

Após uma alteração, ao contrário de numa replicação pessimista, a informação da alteração é partilhada em *background* e caso ocorra um conflito este pode ser resolvido automaticamente ou com ajuda do utilizador.

Qualquer sistema de replicação deve ter o conceito de unidade de replicação mínima, a esta unidade é dado o nome de Objeto. Um Objeto pode ter réplicas por vários locais e um local armazenar réplicas de vários objetos. Com estes objetos é que são partilhadas as alterações realizadas em cada local. Embora a alteração possa ser realizada em apenas uma máquina (single-master / writer) ou em várias (multi-master / writer).

Qualquer sistema que utilize replicação otimista deve também permitir acesso a dados que estejam "desligados" das outras réplicas. Podemos pensar num sistema móvel em que o utilizador se conecta á internet apenas quando necessário. Deste modo o utilizador pode realizar as alterações em modo *offline* e depois realizar a propagação das alterações quando se ligar à internet. Deste modo podem ser armazenadas as operações que foram realizadas sobre um objeto ou apenas o objeto em si, estes dois modos de alteração de objetos / propagação de alterações têm algumas diferenças e iram ser discutidos mais á frente.

Escolhas de desenho

Quando estamos a desenhar uma aplicação que contenha replicação de dados devemos pensar nalguns temas que nos permitem visualizar a complexidade do nosso sistema e como o vamos implementar.

Complexidade e Eficiência

Um dos pontos que devemos analisar é a quantidade de escritores que existe, que réplicas, se alguma, permitem a alteração de dados. Este ponto ajuda a definir a complexidade base do nosso sistema. Quanto mais escritores, maior a possibilidade de conflitos e mais frequente devem ser os *updates* de informação. Num sistema com apenas um escritor, como por exemplo no DNS, não há necessidade de tratamento de conflitos visto que estes não irão ocorrer, basta que, quando uma alteração for realizada, as réplicas sejam informadas dessa alteração. Num sistema com múltiplos escritores é necessário um tratamento de conflitos pois mais do que uma réplica pode ter realizado alguma alteração sobre um objeto.

Outro ponto que afeta diretamente a complexidade do sistema são as operações realizadas, se a alteração de um objeto é informada pelo estado do objeto ou apenas pela operação realizada sobre este. No primeiro caso irão ocorrer mais conflitos que no segundo pois, com o estado do objeto torna-se mais simples indicar apenas que houve uma alteração mas pode ser complicado saber onde, ao saber a operação sabemos exatamente o que foi alterado e pelo que foi alterado. Esta última solução permite um melhor tratamento de

conflitos, seja ao verificar que a alteração ocorreu em locais distintos e portanto não "chocam", ou mesmo a aplicar melhorias na própria ordem em que as operações foram realizadas, embora para esta última situação seja necessário que o sistema conheça a sua semântica.

Concorrência

Também tem de ser pensado como é realizado o scheduling, ou seja, como são ordenadas as operações quando é realizada alguma alteração sobre um objecto. Este scheduling pode ser também sintático ou semântico. Num scheduling sintático apenas é tomado em conta onde, quando, e por quem foi realizada a operação, ou seja, se três operações A, B e C forem realizadas numa máquina, se as três operações fossem realizadas em máquinas distintas mas pela mesma ordem, a ordem pela qual elas iriam ser realizadas nas réplicas seria a mesma. Ao ter um scheduling semântico é tido em conta informações do sistema para melhor ordenar as operações. Por exemplo, se no caso anterior, a realização da operação C antes de B causasse que todas tivessem um resultado positivo enquanto que pela sua ordem original uma delas obtivesse um resultado negativo, se não houver inconvenientes para o sistema, este pode reordenar essas duas operações.

No campo de concorrência temos também o tratamento de conflitos. Como já disse anteriormente, os conflitos podem ser tratados automaticamente ou com input do utilizador. Podemos pensar no caso do SVN ou do GIT em que é realizado um *merge* automático de ficheiros mas em caso de conflitos que o sistema não consiga resolver, é pedido ao utilizador que os resolva e volte a enviar o resultado. Neste caso temos uma combinação das duas soluções.

Eficiência

Em aspetos de eficiência é necessário pensar na estratégia da propagação, como são propagadas as operações entre locais. Também aqui temos várias soluções, podem ser realizados *pull's* manuais (por exemplo em PDA's) ou periódicos (como no caso do DNS) ou por *push*, através de um update periódico. Quanto mais curto o período entre atualizações menor a inconsistência de informação e número de conflitos, mas também aumenta a complexidade do sistema e o overhead.

Qualidade

Aqui entra em conta as garantias que o sistema providencia. Se é garantido apenas o estado final do objeto, podendo o estado intermédio (entre operações) ser um pouco inconsistente, ou se todos os estados (intermédios e final) são garantidos. Aqui entra também em conta o scheduling pois como tinha informado é possível que as operações sejam realizadas por ordens diferentes da realizada pelo utilizador desde que o resultado final seja o mesmo.

Deteção de relações de concorrência e "Happens-Before"

Um sistema de replicação otimista suporta várias operações em simultâneo através do escalonamento das mesmas e da deteção de conflitos entre elas. Para ser possível o escalonamento o sistema tem de saber quais os eventos e a ordem pela qual ocorreram.

Contudo num ambiente distribuído onde o atraso nas comunicações é imprevisível não é possível ordenar de forma absoluta as ocorrências dos eventos.

Para resolver este problema foi proposto por Lamport [1] num artigo de 1978 o conceito de *"happens-before"* que consiste em capturar relações de precedência entre eventos e tentar definir entre eles a ordem de ocorrência. Considere-se duas operações A e B efetuadas sobre os processos pA e pB respetivamente. A operação A *"happens-before"* B quando:

- pA = pB e a operação A for submetida antes de B;

- pA ≠pB e a operação B for submetida após pB receber e executar a operação A;

- Para uma dada operação C, A *"happens-before"* C e C *"happens-before"* B;

No caso de não ser detetada a relação de *"happens-before"* as operações dizem-se concorrentes. Este tipo de relações é usado em vários cenários na replicação otimística como o de ordenação de operações, deteção de conflitos e propagação de operações.

Algoritmos de deteção e representação de relações *"happens-before"*

Existem vários algoritmos para representar e detetar estas relações:

- **Representação Explicita** - Nome das operações precedentes é anexado à operação x;
- **Relógios vetoriais** - Cada processo contem um vetor de relógios onde para cada i é guardado o número de eventos ocorridos no processo Pi. Quando é submetida uma nova operação y no processo Pi este incrementa o seu valor de eventos ocorridos no vetor e anexa o novo vetor à operação y para atualizar os vetores dos outros processos;
- **Relógios lógicos e de tempo-real**
    - **Lógicos** - Cada processo tem um relógio interno e ao ser feita uma operação x esse relógio é incrementado e anexado à operação. O processo destinatário da operação incrementa o seu relógio para um valor maior que o seu ou maior que o anexado na operação;
    - **Tempo-Real** - Comparação dos relógios físicos.
- **Relógios plausíveis** - Combinação de alguns aspetos dos relógios lógicos com vetoriais;

Controlo de concorrência e consistência eventual

Informalmente consistência eventual significa que todas as réplicas irão eventualmente estar sincronizadas quando o sistema se encontra inativo por dado período de tempo. Um objeto replicado está eventualmente consistente quando respeita as seguintes condições assumindo que todas as réplicas partilham o mesmo estado inicial:

- Em qualquer momento, para cada réplica existe um prefixo do *"schedule"* o qual é equivalente ao prefixo do *"schedule"* das restantes réplicas. A esta condição dá-se o nome de *"committed prefix"*.
- O *"committed prefix"* de cada réplica cresce de forma monotónica ao longo do tempo.
- Todas as operações não abortadas no *"committed prefix"* satisfazem as suas pré-condições.

Para atingir a consistência eventual o sistema tem de adotar uma política de scheduling, detetar conflitos e posteriormente resolvê-los.

Schedulling

Existem essencialmente duas políticas de *"scheduling"*:

- **Sintático** - Baseado em informação de caracter geral como quando, onde e por quem foram submetidas operações
- **Semântica** - Baseado nas relações semânticas entre as operações tais como a comutatividade, onde duas operações consecutivas se forem comutativas a sua ordem de execução pode ser trocada mesmo que haja uma relação de *"happens-before".*

Existem também outros tipos de *"scheduling"* semântico tais como ordenação canónica, transformação operacional e abordagem de otimização.

Deteção de conflitos

Uma operação A está em conflito quando a sua pré-condição não é satisfeita, dado o estado da réplica após várias tentativas e de aplicar operação antes de A no *"schedule"* corrente. A gestão de conflitos envolve duas fases, deteção de conflitos e resolução dos mesmos.

Tal como nas políticas de *"scheduling" e*xistem duas abordagens para deteção de conflitos em sistemas otimistas:

- **Abordagem sintática** - Duas operações estão em conflito quando são concorrentes não havendo relação *"happens-before"*.
- **Abordagem semântica** - Duas operações estão em conflito quando ambas têm a mesma semântica. Por exemplo a escrita concorrente de dois ficheiros distintos no mesmo sistema de ficheiros não é um conflito mas a alteração concorrente do mesmo ficheiro é.

Resolução de conflitos

A resolução de conflitos tem como finalidade reescrever ou abortar operações que potenciem conflitos.

A resolução de conflitos pode ser manual ou automática:

Manual - O sistema exclui a operação conflituosa do *"schedule"* e apresenta duas versões do objeto. Cabe ao utilizador criar uma versão nova com *"merging"* das duas versões e voltar a submeter a operação. (**e.g. – Sistema de controlo de versões Git**).

Automático – É chamado um procedimento especifico da aplicação que pega nas duas de um objeto e cria um novo. Por exemplo, atualizações concorrentes a um ficheiro de uma diretoria de e-mail podem ser resolvidas através da computação da união das mensagens de duas réplicas.

Protocolos de compromisso

Para implementar as 3 etapas da consistência eventual existem os protocolos de compromisso onde a permanência das operações é acordada entre os intervenientes nomeadamente a política de *"scheduling"* e a técnica de resolução de conflitos.

Um dos protocolos é o de compromisso implícito por conhecimento comum. Os sistemas que implementam este protocolo fazem uso de timestamps na ordenação das operações de forma determinística e os conflitos ou não existem ou são ignorados.

Existem ainda outros protocolos como o de acordo em segundo plano e por consenso.

Propagação

Um dos paços fundamentais é a propagação, seja ela de estado de um objeto ou de operações, é necessário sempre num sistema com replicação otimista ter uma forma viável de propagação das réplicas.

Como mencionado anteriormente existem dois tipos de propagação:

- Propagação de operações, em que o que é propagado são as operações necessárias para trazer uma réplica local até a um novo estado;
- Propagação de estado, em que o estado completo da réplica é propagado.

Propagação de operações

A forma mais usada para realizar a propagação de operações é através de *vector clocks*. Desta forma é possível identificar violações causais, ter um controlo sobre os eventos de alteração de estado sobre as réplicas, num ambiente distribuído, e poder identificar repetições de operações e possíveis conflitos.

Propagação de estado

A propagação de estado é normalmente realizada enviando todo o conteúdo de uma réplica para outro local, esta forma não é viável num contexto em que um objeto é bastante grande, para tal existem várias técnicas, descritas a seguir, de propagação de estado que permitem a partilha de réplicas, sem a ineficiência de enviar o objeto por completo.

Transferência híbrida

Uma forma de transferência de estado é através de uma transferência híbrida, isso baseia-se na transferência de apenas de uma história de alterações de estado. Ou seja, durante um período de tempo são registadas as alterações realizadas sobre uma réplica, *diffs*, quando for feito um *update* de uma réplica são apenas enviados os *diffs* necessários. Se não existir registo na história da réplica que vai ser atualizada, será então enviado o conteúdo do objecto.

Existem outras técnicas de propagação de estado como dividir um objeto em múltiplos subobjectos, estruturados numa árvore. Durante um *update* é percorrida a árvore à procura de alterações de estado no objeto.

Técnicas de transferência por *push*

As técnicas de propagação anteriormente faladas, admitem que um *site* sabe de alguma forma em que momento será feita a propagação. Existem certos serviços onde a propagação é feita de forma síncrona ou através de um *polling* periódico, ambas as opções trazem algumas desvantagens com o *delay* existente na propagação síncrona e o *overhead* trazido pelo *polling*. As transferências por *push* baseiam-se na lógica de um *site* proactivamente propagar as operações para outro site. Uma forma simples de fazer isto é através de *flooding*, neste caso *blind flooding*, um *site* recebe uma operação nova e envia automaticamente essa operação para os *sites* vizinhos, que usam algoritmos para filtrar duplicados.

Divergência entre réplicas

A propagação apenas trata de implementar a consistência eventual, mas não trata a qualidade de uma réplica. Apesar de existirem sistemas que não precisam de uma garantia de frescura de uma réplica, é necessário arranjar soluções para sistemas que precisam.

Ordenação de leituras/escritas

Um caso que acontece quando não existe um controlo de consistência de réplicas é a aparente noção de um objecto ter "andado para trás no tempo", pois foi feita uma escrita numa réplica e uma leitura noutra réplica antes de ter sido propagada a primeira escrita.

Garantias de sessão

Uma estratégia para resolver este problema é através de políticas de ordenação escolhidas pelo utilizador:

- Read your writes;
- Monotonic reads;
- Writes follows reads;
- Monotonic writes.

Desta forma existe uma ordenação explicita entres as leituras e escritas que forca a existência de *updates* de réplicas determinados pelas ações de leitura e escrita.

Limitar as divergências entre réplicas

Neste cenário é criado um *threshold* que limita as operações que podem ser realizadas sobre uma réplica sem ocorrer uma partilha de atualizações para outras réplicas. Um *bound* numérico simples, seria a limitação de X atualizações sobre uma réplica sem existir uma propagação das atualizações.

Referências

[1] - Time, Clocks and the Ordering of Events in a Distributed System