Síntese de Artigo: *Atomicity and Isolation for Transactional Processes*

Rodrigo L. M. Flores

25 de outubro de 2010

1 Problema

O problema a ser discutido no artigo é a atomicidade e o isolamento em processos transacionais. Um processo transacional é uma sequência bem definida de passos computacionais executados de maneira coordenada. Exemplos clássicos são operações bancárias (abrir contas, fazer uma transferência, fazer pagamento de um boleto), fazer uma compra em um *e-commerce* ou reservar uma passagem aérea, entre outros. O objetivo do artigo é propôr um modelo que solucione questões de isolamento e atomicidade em processos transacionais.

Por isolamento, entendemos que todo passo deve ser executado como se estivesse executando sozinho, sem a interferência de outras tarefas que por ventura estejam sendo executadas. Por atomicidade, entendemos que todo passo deve ser executado por completo ou abortado. Já existem soluções para ambos para processos simples como a atualização de um valor em uma tabela de banco de dados. Porém o escopo a que nos referimos são processos grandes e que podem demorar mais de um dia para ser executados e muitas vezes dependem de agentes externos.

2 Contextualização do problema

Um exemplo básico e de fácil compreensão é o de um *e-commerce*. Suponha que exista um *e-commerce* que faça vendas de arquivos digitais e que podem ser baixados da internet e cuja finalização de venda seja uma descriptografia de um arquivo. O exemplo consiste inicialmente em receber informações do pagamento, após isso se verifica a validade das informações de pagamento do cliente (se há saldo disponível, se o limite do cartão não foi ultrapassado, etc) e a seguir o sistema recebe do vendedor a chave que descriptografa os arquivos a serem entregues. A partir

daí temos um verificador de *time-out*, que escreve um registro dos recibos de pagamento. Em caso que o verificador ocorreu sem problemas, entregamos a chave criptográfica ao comprador, transferimos o dinheiro do pagamento e enviamos a confirmação ao vendedor (isto é, a transação ocorreu com sucesso). Em caso de problema no verificador, notificamos o comprador, depois o banco e o vendedor (a transação foi abortada).

Perceba que o exemplo tem dependências externas (sistema de cobrança), o que torna pouco viável a alternativa disso ser atômico (não é razoável parar o sistema inteiro aguardando uma resposta de um agente externo). Outro fator é que uma venda não pode influir na outra (como não temos a noção de estoque aqui podemos considerar isso).

3 Soluções existentes e seus limites

Como já discutido, uma solução possível é subdividir o processo em diversos processos atômicos. Mas a implementação disso em uma linguagem com *C* ficaria bastante complexa, manutenção e a compreensão mais difíceis. Um nível mais próximo do banco de dados e com uma lógica mais aparente torna a aplicação mais simples de se manter e de se compreender.

4 Contribuições

Uma grande vantagem de se utilizar processos é que estes mantém explícitas as lógicas de negócio, que, tradicionalmente eram escritas em *C* ou *C++* e por serem implementadas com um lógica de programação, estas são difíceis de se manter, de se evoluir e de se compreender. Utiizando os processos, as informações são escritas em um nível mais alto, facilitando a manutenção, evolução e compreensão do mesmo.

Diferentemente de uma transação, que é atômica e que só pode ser totalmente revertida ou terminada (com um commit), um processo pode fazer reversões (*roll-back*) parciais ou seguir um outro caminho possível. Outra diferença entre transações e processos é que os processos podem ter dependências entre suas etapas e sua execução só pode acontecer de uma determinada ordem.

Processos podem também ter pontos denominados “sem volta”, que após serem concluídos o processo não pode mais voltar e deve ter ou um caminho de escape (aborto) ou pode ser tentado varias vezes um mesmo passo até se obter um sucesso. Isso é um contraponto ao modelo já existente, baseado em commits e aborts.

Exemplificando, o exemplo do e-commerce possui um passo no qual é verificado se haverá fundos para a transferência e em outro passo é feita a transferência. A etapa da transferência é garantida que será feita com sucesso, pois já foi verificada a disponibilidade. O pior dos casos pode acontecer da tentativa ser executada várias vezes.

4.1 Modelo

Processos são normalmente vistos como uma coleção de atividades as quais são, por definição, atômicas e podem ser realizadas ou não com sucesso. Atividades (denotadas por a) podem ou não ter uma outra atividade (denotadas aqui por a^{-1}) tal que a execução das duas não causa nenhum efeito, ou seja, a atividade a^{-1} compensa a atividade a. Se uma atividade não pode ser compensada, ela é chamada então de pivô. Quando um pivô é executado, o processo deve prosseguir, pois não há como compensar a execução do pivô.

Outro tipo possível de atividade é chamada de *retriable* (baseada em tentativas). Este tipo de atividade garante que após um número finito de tentativas, haverá um *commit* e em todas anteriores nenhum *commit* aconteceu. Um bom exemplo de atividade é o recebimento do pagamento após verificar que há saldo. Se há saldo, então o pagamento pode ser feito (independentemente da atividade não ter sido concluída por outro motivo, como uma queda na conexão por exemplo).

Nada impede que um processo realize duas atividades concorrentemente, porém devem haver indicações se uma atividade deve acontecer antes que outra (ordem forte). Pode também acontecer de uma atividade (que chamaremos de a) poder ser executada concorrentemente com outra (que chamaremos de b), porém o resultado esperado deve ser equivalente ao de b acontecer depos de a (ordem fraca).

4.1.1 Estrutura do processo

O primeiro pivô é chamado de pivô primário e nunca são constituídos de mais de uma atividade, o que nos mostra que jamais outra atividade pode ser executada concorrentemente com o pivô. Após o pivô deve sempre haver uma árvore, chamada de árvore de finalização concreta, consistindo apenas de atividades baseadas em tentativas. Em geral um pivô tem filhos nos quais esta mesma estrutura que definimos aqui vale recursivamente.

Um programa de processo é constituído de:

- Um conjunto de nós, que chamaremos de N, de atividades. Um nó pode ser constituído de mais de uma atividades, desde que esta seja compensável. Se

o nó for pivô ou *retriable*, ele deve ser *singleton*, isto é, constituído apenas de uma atividade.

- Um conjunto de arestas E, que juntamente com o conjunto de nós N, formam um árvore. Cada aresta e, representa uma ordem forte.
- As ordens (parciais) fortes.
- As ordens (parciais) fracas
- Os nós pivôs
- As preferências dos pivôs (ou seja, o que fazer após cada pivô). O último item de preferência de cada pivô deve ser sempre uma árvore de finalização concreta.

No exemplo mostrado na seção anterior, os nós pivôs são:

- Nó de verificação de *time-out*.
- Nó de transferência das chaves.

Há dois caminhos a se seguir após a verificação de *time-out*: o de aborto e o de sucesso. No nó de transferência da chave, a decisão é o ramo em que se está. A preferência é o nó de sucesso sobre a árvore de finalização concreta (aborto). Os estados baseados em tentativas são: o estado de recebimentos das chaves pelos vendedores, envio da confirmação para os vendedores em caso de commit e a notificação aos vendedores em caso de falha.

4.1.2 Estados de processos

Ao ser iniciado, um processo pode ter diversos estados. Chegando finalmente em um estado de terminado *commited* ou a um estado de abortado *aborted*. Listamos a seguir os diversos estados possíveis.

- Uma vez iniciado, o estado é executando;
- Antes do pivô primário, um aborto de alguma atividade compensável muda o estado para abortando;
- Após compensar todas as atividades, o processo é determinado abortado;
- O commit de um pivô, muda o estado de executando para terminando. O aborto de qualquer estado após o pivô, faz ele tentar a próxima. Então o estado deve apontar qual é a alternativa que está sendo executada.

- Se uma alternativa termina, o estado final é chamado de terminado (*commited*)

Repare que a sequência de execução de um processo não é necessariamente um caminho na árvore, pois podem haver tentativas e compensações (poderiamos considerar a sequência de execução como sendo um passeio). O efeito atual de um projeto (isto é, todos os itens que executaram corretamente e não foram compensados) é um caminho na árvore.

5 Discussão/conclusão

Aplicações grandes normalmente integram muitos componentes independentes e distribuídos (que podem involver um banco de dados local e uma integração com web-services de bancos ou de cartões de crédito no caso do *e-commerce*) ao invés de depender de somente um banco de dados local. O desenvolvimento destas aplicações deve levar em conta não somente os passos computacionais de um processo, mas sim a união de todos os passos de uma maneira coerente.

Utilizar processos permite um nível maior de especificação, que torna a manutenção, compreensão e evolução mais simples de serem realizadas. A abordagem dada aos processos, permite que eles sempre terminem, seja com um aborto que apenas faça a compensação ou com um commit que termine a execução com sucesso. Generalizamos então a questão de atomicidade do processo. Embora nem sempre ele possa ser apagado (se ele passou pelo pivô ele foi necessariamente registrado). E, diferentemente de outras abordagems, cobrimos tanto a atomicidade como o isolamento fazendo o controle de concorrência e a recuperação no nível adequado, o escalonamento do processos.

Referências

[1] Heiko Schuldt, Gustavo Alonso, Catriel Beeri, and Hans-Jörg Schek. Atomicity and isolation for transactional processes, 2002.