Um Protocolo de Atomicidade Local Híbrida para Tratar Transações de Políticas Otimista, *One* e *Two-Phase Locking*, com Possibilidade de Extensão

Maria Alice Brito
malice@ime.uerj.br

Paulo Roberto Lobo
prlobo@dba.com.br

DICC – Departamento de Informática e Ciência da Computação
IME – Instituto de Matemática e Estatística
CTC – Centro de Tecnologia e Ciências
UERJ – Universidade do Estado do Rio de Janeiro
Rua São Francisco Xavier, 524 / Bloco B – sala 6020
CEP: 20.550 - 013
Rio de Janeiro, RJ, Brasil

Abstract

*In this work, we allowed the transactions with three distinct policies – optimistic (*O*), one-phase locking (*1PL*) and two-phase locking (*2PL*) – sharing simultaneously each participant application object. Our contributions are extensions from the [7, 6, 1, 16] approaches, with focus on the pessimistic policy. We placed the lock request in the invocation event handler to enable asynchronous mode for interaction messages. This solution restricts the concurrency chances whereas return conditions cannot be used in the conflict table [7]. We have added two measures for compensate this restriction. The* 2PL *transactions dispose of one exclusive version for each participant object, during the first phase, and the validation action was deferred to the end of this phase. These measures have avoided the unnecessary locking time until the execution phase beginning. The* 1PL *transactions conflict table have got two changes: the conflict condition meaning [16] was substituted from update to execution error; and a new third condition U was created and its meaning is a first level operation code with some reference/assign primitives which need be serialized. These measures have enabled the second level derived operations (read/write primitives) keep locked only during their first level operation run time, and when this first level operation response event returns, these read/write primitives are unlocked.*

1. Introdução

Este trabalho se situa no contexto de controle de concorrência e recuperação (CCR), explorando oportunidades para que transações com novas políticas possam ser estendidas (via reuso) e tenham sua coexistência negociada entre as transações com diferentes políticas que venham a compartilhar cada objeto de uma aplicação. Os mecanismos de controle de concorrência (MCCRs) ficam concentrados no meta objeto de uma meta arquitetura, que conta com mais dois elementos: um gerente de transações, que coordena as fases de uma transação, cooperando com os meta objetos, para alcançar atomicidade global; e um gerente de objetos e de transações, que desempenha um papel de armazém. Essa distribuição de responsabilidades tende a facilitar a programação das aplicações que venham a empregar transações avançadas, bem como possibilitar a extensão de novos modelos.

Essas expectativas para o emprego e extensão de modelos avançados de transações têm chances de ser atendidas, como, por exemplo, nas abordagens [1, 16, 18, 8], que aproveitam as seguintes oportunidades: 1) a atomicidade local híbrida [7, 6, 17], que além de explorar a semântica das operações, trazendo mais chances de concorrência, traz oportunidade para a convivência de políticas diferentes concorrerem no mesmo objeto de aplicação; 2) a granularidade do CCR em cada operação de cada objeto [7]; 3) a serialização global [17, 4] e o impedimento da anomalia de herança (Wegner 1987, Kafura e Lee 1989), podendo ser resolvidas pelo uso de reflexão [12, 9], utilizando um meta objeto associado a cada objeto da aplicação; e 4) a gerência da transação sendo programada em classe, prevista uma instância para cada transação que surgir no ambiente de execução.

Dirigindo a atenção para certas características da orientação a objeto, destacamos os dois fatores, a seguir, que contribuem nos MCCRs, permitindo maior intercalação das operações pedidas ao objeto, mantendo a propriedade da capacidade de serialização: 1) um objeto pode ser acessado/atualizado somente por suas operações, e 2) o programador pode registrar na classe do objeto as relações de conflito entre suas operações, que podem ser consultadas pela sua interface. Esses recursos vem sendo explorados tanto para objetos simples quanto para objetos complexos, como observado em [15], ao retratar as técnicas baseadas em semântica, empregadas em controle de concorrência. A separação do critério de correção do modelo de transação e de suas duas dimensões, a estrutura do objeto e a estrutura da transação, facilita o estudo dessas questões [14] e ao mesmo tempo pode auxiliar a construção dos mecanismos separados em módulos cooperativos de acordo com os aspectos a serem tratados.

Com a finalidade de obter maiores chances de

concorrência, explorando as semânticas das operações de mais alto nível sobre objetos, foram desenvolvidos protocolos de controle de concorrência relacionados aos efeitos que as operações produzem sobre os estados dos objetos, como os que podem ser vistos, em [17, 7, 6, 3].

A atomicidade local incorpora os MCCRs locais ao tipo abstrato de dados, explorando modularidade, quer dizer, as semânticas das operações de mais alto nível sobre objetos, que foi desenvolvida por Willian Weihl, em sua tese, em 1984, e publicada em [17]. Por essa abordagem os objetos compartilhados conseguem garantir atomicidade global às transações que os acessam. Eles são programados por tipos atômicos que apresentam dois lados: um que trata a parte seqüencial do objeto e o outro que trata o lado concorrente. Pelo encapsulamento da sincronização e recuperação necessárias para suportar atomicidade na implementação de objetos compartilhados, a modularidade é beneficiada. Em adição, pelo uso das informações sobre as especificações dos objetos compartilhados, a concorrência entre as transações é beneficiada. É necessário que a serialização global da transação seja garantida e, para isso, tem que haver compatibilidade entre as atomicidades locais, como observa Willian Weihl [17].

Em [3], o protocolo é baseado em comutatividade com adição de informações de recuperação à tabela de conflitos, que deve ser derivada diretamente da especificação do tipo do objeto.

A propriedade de atomicidade local, inicialmente, foi especificada por conflitos definidos por comutatividade [17]. A propriedade de atomicidade local híbrida, de acordo com Weihl, em sua tese (1984), caracteriza o comportamento de protocolos que exibem algumas das características de protocolos atômicos dinâmicos e algumas das características de protocolos atômicos estáticos. Protocolos atômicos estáticos funcionam bem em transações *read-only* e protocolos atômicos dinâmicos funcionam melhor quando há predomínio de atualizações. Com o mecanismo de sincronização híbrido, encontrado em [5], os conflitos em um objeto têm a possibilidade de serem tratados de forma otimista ou pessimista, com técnicas que satisfazem a atomicidade local híbrida. As chances de concorrência foram aumentadas, ao serem desenvolvidos novos protocolos [7, 6], que substituíram a comutatividade por requisitos mais frágeis de dependência, contando com a ordem de serialização explícita, com emprego de *timestamp*s, obtidos na fase de finalização de uma transação.

Em [7], a verificação adota uma relação batizada de *locking conflict relation* para detectar o conflito entre cada duas operações, sendo derivada das relações de dependência. Essa verificação é usada por um esquema caracterizado como *two-phase locking* (2PL), cujo protocolo pode ser descrito por uma máquina de estado chamada de *LOCK*, tendo como entradas o seguinte conjunto de eventos gerados pelas gerências de transações: {*inv*, *res*, *commit*, *abort*}.

Em [6], é introduzida uma abordagem para validação baseada em conflitos pré-definidos entre pares de operação, a ser adotada por um MCCR otimista. Essa validação é dirigida por uma *optimistic conflict relation*, abreviada por **OC**. O autor faz notar que essa abordagem é análoga aos mecanismos de *locking*, que usam conflitos pré-definidos para introduzir *delays*, como em [7].

Explorando ainda as propriedades de atomicidade híbrida, em [6], o autor argumenta que as técnicas otimistas propostas são compatíveis com os esquemas pessimistas com atomicidade híbrida, permitindo que os objetos que empregam técnicas otimistas e pessimistas participem de um único sistema, podendo também ser combinadas dentro de um mesmo objeto. Esse esquema misto, chamado de *mixed conflict-based validation*, pode explorar as forças de cada método, usando técnicas pessimistas para prevenir conflitos de alto risco e, reservando técnicas otimistas para detectar conflitos de baixo risco.

Esse esquema misto permite tratar a coexistência entre transações de diferentes políticas, como os MCCRs que localizamos no meta objeto da nossa meta arquitetura, dirigindo nossas atenções para esses protocolos baseados em atomicidade local híbrida. Nossa contribuição principal é a de trazer as discussões e soluções para explorar mais chances de concorrência nos protocolos de controle de concorrência para transações pessimistas (*one* e *two-phase locking*), estendendo as implementações, em [1, 16], que foram baseadas sobre os protocolos, em [7, 6].

Este artigo está organizado, como, a seguir. A seção 2 mostra uma discussão sobre os trabalhos em foi baseada a nossa implementação; a seção 3 descreve nosso modelo de computação; a seção 4 descreve como os nossos mecanismos de controle de concorrência e recuperação ficaram resolvidos; e a seção 5 encerra o artigo com as conclusões.

2. Protocolos e Implementações de Atomicidade Local Híbrida

A validação, em [6], traz a verificação de conflito pessimista para a pré-condição da invocação da operação, melhor que na validação, em [7], em que a verificação foi localizada no evento de resposta. No entanto, a aquisição do bloqueio ainda permanece no evento de resposta. A exclusão mútua em nosso protocolo não é totalmente garantida até o evento de resposta, por causa do assincronismo, e um bloqueio da operação somente mais tarde, no evento de resposta, pode ser frustrado. Pretendemos a mesma granularidade do CCR, adotada no protocolo, em [7], que é "por operação e por objeto", uma mensagem pode ser propagada por objetos da composição do objeto, que também efetuam o CCR. Um conflito detectado mais tarde pode causar uma situação indesejada de *rollback propagation*.

A implementação, em [1], adotou a divisão de comportamento de protocolos de políticas diferentes,

fazendo a verificação de conflitos entre operações com técnicas pessimistas e operações com técnicas otimistas, somente, durante a fase de finalização, e, em caso de conflito, as operações com técnicas pessimistas têm prioridade, sendo a transação otimista abortada. Essa divisão, também adotada, em [16], foi mais adiante, conduzindo a separação dos protocolos em classes de meta objeto distintas.

As transações otimistas, na fase de execução, em [16], contam com um estado compartilhado, cuja composição vai sendo restaurada da memória secundária por demanda. Se houver atualização, por parte de alguma transação, esta adquire um estado exclusivo, da parte atualizada. Essas transações contam com um protocolo com técnicas otimistas, cuja validação adota dois níveis de conflitos: no primeiro nível entre as operações de alto nível; e no segundo nível, a validação verifica se as leituras iniciais dos atributos obtiveram os mesmos valores que serão atualizados pela transação em finalização. Se a validação alcança êxito, a recuperação providencia as atualizações relativas a transação em finalização, no estado permanente. Em caso contrário, as operações da transação são re-executadas. As transações pessimistas, em [16], contam com um outro protocolo com técnicas pessimistas *2PL*, em que as operações são classificadas como operações *read* se elas não causam trocas no estado do objeto, e *write* se causam. O tratamento de uma mensagem no objeto destinatário é efetuado por um meta método, chamado de `Meta_NormalOperation`, que procede a validação e aquisição de bloqueio, na pré-condição da invocação, que, se bem sucedida, invoca efetivamente a operação do objeto seqüencial, aguarda o retorno com seu resultado, que é então devolvido ao emissor. A recuperação é da mesma forma que para as otimistas.

Para a recuperação, a estratégia, nos protocolos [7, 6] e em suas implementações [1], tem sido a *deferred update DU*, contando com uma lista das intenções a serem aplicadas ao estado permanente, durante a fase de finalização. Caso surja alguma necessidade de aborto, essas intenções são apenas descartadas.

Na abordagem [16], é apresentada uma solução que simplifica essa tarefa de aplicação de lista de intenções. Para as transações pessimistas, é adotada a estratégia *update in place* (*UIP*) no estado corrente. Durante a fase de finalização, os valores dos atributos que foram atualizados no estado corrente são copiados para os atributos do estado permanente, para que seja procedida a gravação.

Considerações Fazemos notar que a separação adotada em [1] também foi considerada em nossa implementação, empregando o artifício de classes aninhadas, que permite que os MCCR para os esquemas diferentes fiquem reunidos no mesmo meta objeto, em vez de meta objetos distintos, como em [16]. Entre as operações de transações de esquemas diferentes, aproveitamos a validação de [1], admitindo apenas as classes pessimistas e otimistas, omitindo a classe híbrida, com o propósito de evitar a incompatibilidade entre a atomicidade local e global das transações. A maioria das soluções encontradas, em [16], foram aproveitadas em nossa implementação, como, as seguintes: a obtenção por demanda dos objetos da árvore de composição, sendo resolvida por nós pelo método de persistência rasa [11]; a validação em dois níveis, que trouxemos também para as pessimistas, melhorando as chances de concorrência, que, em [16], ainda ficaram restritas a uma validação baseada na classificação das operações de primeiro nível consideradas como operações *read* se elas não causam trocas no estado do objeto, e *write* se causam. Deixamos o assincronismo totalmente explícito, ao desdobrarmos uma mensagem de interação em quatro meta mensagens: `sendMsg`, `sendRet`, `recMsg` e `recRet`, que constituem um ciclo a ser percorrido, entre os meta objetos da interação, sendo cada uma delas também enviada ao gerente de transação de nossa meta arquitetura, para que esse possa controlar os casamentos de retorno e envio, afim de descobrir os términos da primeira fase, para uma transação *2PL*, e da fase de execução, para todos os tipos de transação.

3. Modelo de Computação

Nossos MCCR contam com uma meta arquitetura, que constitui-se de três elementos principais, programados em classes, como, a seguir: 1) o gerente de objetos e de transações (GOT), que funciona como um armazém de objetos de aplicação e de transações; 2) o gerente de transação (GT), sendo criada uma instância para cada transação, que surja no ambiente de execução, que, em linhas gerais, implementa o modelo de transação orientado a objeto de acordo com a terminologia em OMG's Object Transaction Services (OTS) [13], cujo maior benefício apontado é o de permitir que todo objeto resolva seu próprio CCR. As responsabilidades do GT resumem-se à cooperação com os MCCRs dos objetos de aplicação participantes de sua transação, com a finalidade de preservar a atomicidade global de acordo com a política que sua classe implementa. Uma transação pode surgir por dois caminhos: a) pela execução de um comando `begin transaction`, em um método de um objeto de aplicação, e terminada pelos comandos `abort` ou `commit`, ou b) automaticamente, quando um meta objeto descobre que uma meta mensagem `sendMsg` está circulando sem um contexto de transação associado. Essa transação que surge de forma implícita deve ser encerrada no tratamento do evento de resposta dessa mensagem, a meta mensagem `sendRet`. No caso em que um comando `begin transaction` é executado em um método, que foi propagado, pela composição, a transação criada se torna automaticamente aninhada, quer dizer, implicitamente aninhada. As aninhadas explícitas são aquelas que aparecem em comandos aninhados em blocos de código do próprio método em que a aninhante foi especificada;

e 3) o meta objeto, cuja instância é criada e associada a cada alocação de um objeto de aplicação, em tempo de execução. Nesse meta objeto ficam localizados os MCCRs, que, assim, incorpora os conceitos de objeto ativo [2, 10] e objeto atômico [17], garantindo a atomicidade local de uma transação que o compartilha. O critério de correção, que também faz parte dos MCCRs, fica localizado no objeto seqüencial, porque as relações de conflito precisam ser preenchidas pelo programador da classe. Um outro conjunto de informações que também ajuda os MCCRs é a lista de atributos referenciados/atribuídos, em cada método do primeiro nível, público ou privado (admitimos controle de concorrência em envio a self), que pode ser preenchida pelo compilador. Além dessas localizações no objeto seqüencial, o GOT também assume um discreto papel nos MCCRs, como já observamos, na sua descrição, acima, responsabilizando-se pela detecção e desdobramentos de situações de *deadlocks*. Essa responsabilidade lhe foi delegada, porque ele é o único elemento na meta arquitetura que conhece todos os outros elementos, tanto os objetos de aplicação quanto os gerentes de transação.

As fases em que essa meta mensagem `recMsg` deve chegar podem ser ou a primeira fase de uma transação pessimista *2PL* ou a de execução para qualquer política de transação[1].

Com exceção das meta mensagens para as mensagens de interação entre objetos de aplicação, a maioria das meta mensagens de cooperação entre os elementos da meta arquitetura apresentam-se aos pares, uma para o envio e outra para o retorno, sendo que a meta mensagem de retorno sempre traz a condição da execução da meta mensagem de envio. Esses pares ajudam o elemento emissor a tomar conhecimento do resultado e da conclusão do pedido, lembrando que elas podem ser enviadas no modo assíncrono. As meta mensagens enviadas pelo GT ao meta objeto de cada participante e as conseqüentes meta mensagens de retorno respondidas por cada meta objeto são as seguintes: `firstPhaseEndMsg` e `firstPhaseEndRet` associadas ao final de primeira fase em uma transação *2PL*; `consistencyMsg` e `consistencyRet` associadas à consistência de uma transação otimista, na finalização; `writeMsg` e `writeRet` associadas à gravação de uma transação de qualquer política, na fase de finalização; `writeConfirmationMsg` e `writeConfirmationRet` associadas à confirmação da gravação de uma transação de qualquer política, na fase de finalização.

Os três elementos da meta arquitetura possuem uma caixa em que as meta mensagens chegadas são depositadas e retiradas uma a uma, no momento em que o elemento considerar apropriado, permitindo que todas as meta mensagens que cooperam entre eles sejam trocadas no modo assíncrono, com a intenção de evitar esperas inúteis.

No meta objeto do objeto destinatário, quando o tratamento da meta mensagem `recMsg` começa, a primeira providência tomada é a de criar um *thread* de controle interno à transação, caso a correspondente do contexto que vem agregado à mensagem não seja encontrada entre as transações que já se encontram compartilhando o objeto. Após essas providências, essa meta mensagem desdobra-se no evento de invocação do método pertencente ao objeto seqüencial de diferentes maneiras, dependendo da fase (ou a primeira fase de uma transação pessimista *2PL* ou a de execução para qualquer política de transação) e da política da transação associada à operação, como podemos ver, na seção dedicada aos MCCRs.

Em tempo de finalização de uma transação, o GT notifica cada meta objeto dos participantes da sua transação, enviando uma meta mensagem apropriada a sua fase e política e tratando as meta mensagens de retorno. Com exceção das meta mensagens para as mensagens de interação entre objetos de aplicação, a maioria dessas meta mensagens entre os elementos da meta arquitetura apresentam-se aos pares, uma para o envio e outra para o retorno, sendo que a meta mensagem de retorno sempre traz a condição da execução da meta mensagem de envio. Esses pares ajudam o elemento emissor a tomar conhecimento do resultado e da conclusão do pedido, lembrando que elas são enviadas no modo assíncrono. As meta mensagens enviadas pelo GT ao meta objeto de cada participante e as conseqüentes meta mensagens de retorno respondidas por cada meta objeto são as seguintes: `firstPhaseEndMsg` e `firstPhaseEndRet` associadas ao final de primeira fase em uma transação *2PL*; `consistencyMsg` e `consistencyRet` associadas à consistência de uma transação otimista, na finalização; `writeMsg` e `writeRet` associadas à gravação de uma transação de qualquer política, na fase de finalização; `writeConfirmationMsg` e `writeConfirmationRet` associadas à confirmação da gravação de uma transação de qualquer política, na fase de finalização. O meta objeto, ao receber uma meta mensagem dessas, converte-a em um evento, passando aos seus MCCRs, que providenciam o seu tratamento, devolvendo, no final, a condição de execução desse evento, enviando uma meta mensagem de retorno, ao GT. Quando o GT identifica a situação de retorno de todos os seus participantes de uma determinada notificação, então ele passa a fase seguinte, notificando, novamente, todos os seus participantes, e, assim, por diante.

A cada tratamento de uma meta mensagem `MsgRec`, o meta objeto registra todas as informações que são úteis na validação e na recuperação, de acordo com as fases para cada política. Entre esses registros, os MCCRs contam com os bloqueios das operações para as transações pessimistas, os indicadores de uso das operações para as transações otimistas, o *log* de todas as operações *read/write*, cada uma com as seguintes informações: o nome do atributo, o *timestamp*, a identificação da transação, o valor de leitura e o valor

de escrita. Esse *log* será útil para a validação de segundo nível e para a recuperação. Além desses registros, várias filas de meta mensagens `MsgRec` são alocadas: a da própria caixa de mensagens; a de espera por uma vaga na janela das transações pessimistas, em que todas as mensagens, associadas a cada transação que ainda não conseguiu espaço, vão sendo enfileiradas; a de espera, quando uma operação associada a uma mensagem de uma transação *1PL*, não conseguiu ser bloqueada, devido a conflito; a das mensagens associadas a uma transação *2PL*, durante a primeira fase, que deverão ser consistidas, no final dessa fase; além de outras, que foram criadas por razões de implementação. Os MCCRs ainda contam com várias versões do objeto de aplicação, administradas no meta objeto: a do estado permanente, somente atualizada, após a transação ficar completamente finalizada; a temporária compartilhada pelas transações pessimistas, a exclusiva para a primeira fase de uma transação *2PL*; e a exclusiva para cada transação otimista.

Nosso modelo de computação para os MCCRs é uma máquina de estado, como em [7], cujo estado dessa máquina são essas estruturas e as entradas para as transições são os eventos, como seus tratamentos descritos ao longo da seção seguinte.

4. Implementação do Protocolo

Nesta seção, estaremos, na maioria das vezes, descrevendo o comportamento dos MCCRs, implementados na classe do meta objeto do objeto de aplicação. Assim, quando estivermos descrevendo algum comportamento em algum elemento diferente do meta objeto, identificaremos esse elemento e ainda por razões de espaço, omitiremos o emissor de cada meta mensagem daquelas apresentadas, na seção 3, acima.

A nossa versão básica oferece três políticas diferentes para o CCR: uma otimista e duas pessimistas, *two-phase locking* (*2PL*) e *one-phase locking* (*1PL*), para as quais definimos duas tabelas de conflitos, para as operações de nível 1, denominadas por **OC** (*optimistic conflict relation*) e **PC** (*pessimistic conflict relation*), como em [6]. Essas tabelas de compatibilidade são definidas na programação da classe de um objeto de aplicação, porque os conflitos a serem preenchidos dependem da semântica específica da classe. Nas situações em que o programador não defini-las, as verificações de conflito ficam restritas aos tradicionais *read/write*.

Controle de concorrência para as transações pessimistas Para as transações pessimistas, a validação das operações de primeiro nível emprega a tabela de conflitos **PC1** ou **PC2**, em que os seus elementos ficaram restritos ao nome da operação, como, em [6], por causa da verificação localizada na pré-condição do evento de invocação. Para as transações pessimistas *2PL*, como já observamos na seção 2, vimos que se levássemos em conta os possíveis conflitos entre as operações do segundo nível (*read/write*) que fossem surgindo na primeira fase, poderíamos melhorar as chances de concorrência em relação à consideração, em [16], sobre os conflitos entre as operações do primeiro nível, pelo critério dessas operações causarem ou não atualização no objeto.

Existe ainda uma terceira situação, já apontada, em [3], que sugeriu um terceiro tipo de conflito, em que operações ao serem executadas, simultaneamente, não gerariam erro, mas poderiam gerar pendência de finalização ou de aborto. Um exemplo assim é o caso de dois correntistas *A* e *B* que estejam tentando depositar simultaneamente. O código básico para efetivar essa operação é um simples comando de atribuição "saldo = saldo + depósito". Esse comando se desdobra em três operações de nível mais primitivo, como a seguir: i) a referência ao saldo, no lado da expressão; ii) as referências na expressão aritmética para a soma; e iii) a atribuição ao saldo do resultado da expressão. Se essas operações mais primitivas não forem serializadas, o resultado final da operação de depósito pode alcançar um valor errado para o saldo. Para indicar essa condição, acima, utilizamos um terceiro valor *U* a ser assumido pelo elemento da tabela de conflito. Além disso, para cada método do objeto da aplicação, contamos com uma lista, em que são fornecidas todas as possíveis referências/atribuições aos atributos, a serem efetuadas por esse método, durante a sua invocação, que pode beneficiar os MCCR. Chamamos a atenção aqui para a segunda cooperação por parte do objeto de aplicação aos mecanismos de controle de concorrência, lembrando que a primeira é a consulta a suas tabelas de conflito **PC1** , **PC2** e **OC**. Essa segunda informação útil a ser prestada pela parte seqüencial do objeto, a lista de atributos que podem ser referenciados/atribuídos durante a execução do método deveria ser construída pelo compilador, que tem chances de descobri-las durante a geração de código.

Descrevemos, a seguir, a validação, efetuada em momentos diferentes, para as transações pessimistas *1PL* e *2PL*, para obter o bloqueio de cada operação.

Para uma transação pessimista *1PL*, no momento do atendimento à mensagem, dependendo dos seguintes retornos da verificação de conflito de primeiro nível: 1) se retornar *N* (indica que não há qualquer espécie de conflito entre as operações de primeiro nível, nem atualizações em seus atributos), o bloqueio para a operação de nível 1 será obtido; 2) se retornar *Y* (indica que mesmo que as operações sejam serializadas, a segunda invocação pode causar erro, como, por exemplo, duas retiradas de conta corrente), a operação de nível 1 será enfileirada; e 3) se retornar *U*, ainda na pré-condição dessa invocação, devem ser verificados os conflitos para as operações de segundo nível da lista do método, que, em caso de êxito, os bloqueios devem ser efetuados, e, durante o evento de retorno (meta mensagem `recRet`), os bloqueios devem ser liberados. Em caso de conflito, em uma dessas condições (2) e (3), a meta mensagem `recMsg` correspondente à operação deve ser reenfileirada, bem como as meta mensagens sucessoras associadas à

transação, devendo ser atendidas, mais tarde, quando a outra transação concorrente liberar os bloqueios.

Para uma transação pessimista *2PL*: na primeira fase, são registradas todas as operações de nível 1 e 2, que foram aplicadas sobre a versão exclusiva; no final da primeira fase, o estado exclusivo é descartado e é efetuada a consistência, entre as suas operações, de nível 1 e 2, e as operações de todas as transações pessimistas ativas. Se a consistência alcançar êxito, o bloqueio poderá ser efetuado para todas as operações registradas, de primeiro e segundo nível, quando então poderá ser dado início à fase seguinte, a de execução propriamente dita. Fazemos notar que a lista de referências/atribuições que é usada para bloquear as operações de segundo nível não é aquela utilizada no protocolo *1PL* e sim as referências/atribuições que surgiram durante a primeira fase e foram registradas. Essa segunda fase de execução contará com o estado temporário compartilhado por todas as transações pessimistas.

Recuperação para as transações pessimistas Pela nossa solução, mantivemos o compartilhamento de um estado para as transações pessimistas, na fase de execução, funcionando como uma estratégia *UIP*. Na fase de finalização, as transações pessimistas *2PL* e *1PL* têm o mesmo tratamento, em que uma lista de operações *read/write* com seus *timestamps* que surgiram na execução e foram registradas deve ser consultada, permitindo detectar a pendência de finalização entre a transação em foco e as outras pessimistas ativas ou em finalização. Essa consulta serve para indicar em qual das quatro situações seguintes se encontra a transação: 1) se ela não depende de alguma outra transação ativa ou em fase de finalização; 2) se ela depende de alguma outra transação ativa ou em fase de finalização; 3) se ela depende de alguma outra transação ativa ou em fase de finalização, e essa outra, por sua vez, também depende dela; e 4) se alguma outra transação depende dela.

Após essa consulta, o protocolo toma as seguintes ações, dependendo da condição retornada: condição (1) o tratamento da gravação da transação pode prosseguir; condição (2) o tratamento da gravação da transação somente poderá prosseguir, após a conclusão do tratamento da gravação das transações das quais ela depende; e condição (3) todas as transações pendentes entre si deverão ter tratamento de gravação em conjunto; a condição (4) não faz sentido para as ações preparatórias de uma gravação, mas será útil para as liberações de pendências, após as confirmações de gravações e também para aborto de transações.

Para proceder a gravação da transação ou grupo de transações, em finalização, que é decidida, após a consulta acima, dada a garantia de que não há pendência, o estado permanente deve ser atualizado, representando os efeitos das operações de *write* com os *timestamps* mais novos pertencentes ou a transação que deve ser gravada sozinha ou ao grupo de transações, quando esse for o caso.

Se alguma transação pessimista, por questões de *deadlock* ou de falha, precisar ser abortada, a relação de pendência de finalização também deve ser consultada como acima. Se a consulta retornar as condições (3) ou (4), que indicam que há pendência de finalização de transações em relação a que está sendo abortada, o protocolo deverá providenciar o aborto dessas também.

Para evitar que surjam esperas infinitas pelas finalizações de transações pessimistas que geram pendências, porque estamos adotando esse tipo de estratégia *UIP*, foi preciso limitar o número de transações pessimistas ativas que compartilham um objeto. Esse limite é parecido com a solução de janela, em protocolos de rede. Essa restrição deve ser verificada toda a vez que for detectada a chegada da primeira mensagem pertencente a uma transação pessimista, quer dizer, o começo do compartilhamento do objeto pela transação. Essa decisão ainda não elimina totalmente a chance de alguma espera longa, devendo ser adotado um tratamento especial, ou o tradicional aborto por *time-out*.

A estratégia de recuperação *UIP* aumenta as chances de concorrência para as transações pessimistas, porque as condições de conflito indicadas na tabela, são restritas apenas aquelas que podem causar efetivamente erros. Assim, as que causam pendências de finalização, têm suas operações de segundo nível verificadas na execução e a verificação das intercalações adiada para a fase de finalização, reduzindo as esperas na execução e adiando para a finalização as verificações de pendência de uma ou mais transações pessimistas. Esse comportamento da estratégia *UIP* nos levou a criar a condição *U* na tabela **PC1**, que pode ser aproveitada pelas transações pessimistas *1PL*.

Quando uma transação pessimista é completamente finalizada, ela deve ainda ser consistida com cada transação otimista ativa, com a intenção de evitar desperdício de trabalho, porque se for acusado conflito, a transação otimista deve ser abortada. Essa consistência é efetuada entre as operações da transação otimista ativa e as operações da transação pessimista em finalização, verificando os seguintes conflitos: 1) de primeiro nível; e 2) de segundo nível, verificando se os valores das leituras iniciais da transação otimista ativa coincidem com os valores do estado permanente atualizado, como em [16]. Se for detectado algum conflito nessas verificações (1) e (2), a transação otimista deve ser abortada e a versão descartada. Além disso, cada transação otimista ativa que sobreviver à consistência, deve ter sua versão exclusiva atualizada, obtendo-se uma cópia do estado permanente atualizado e aplicando-se sobre ele as suas operações *write* (de segundo nível) com *timestamp* mais novo.

Controle de concorrência e recuperação para transações otimistas Nossa solução, para a política otimista, adota uma versão exclusiva do objeto para cada transação, sobre a qual são aplicadas as operações, durante a fase de execução. A concorrência e a recuperação ficam resolvidas, na fase de finalização, da

forma tradicional. Nessa fase, todas as mensagens já retornaram, assim, a tabela de conflitos **OC** refinada pode ser adotada, como, em [6], porém, somente na validação entre as transações otimistas. A tabela OC usada em [6], permite que o programador use o nome da operação, e mais duas espécies de informações: a) condições de execução da operação e b) seu próprio resultado, em termos absolutos ou em faixas de valores.

Em primeiro lugar a consistência é efetuada entre as operações da transação otimista em foco e as operações de todas as transações pessimistas ativas, verificando os seguintes conflitos: 1) de primeiro nível, pela tabela **PC1** ou **PC2**; e 2) de segundo nível, verificando se os valores das leituras iniciais, efetuadas pelas transações pessimistas, coincidem com os valores dos últimos atributos atualizados pela transação otimista, porque o estado não é compartilhado, como em [16]. Se for detectado algum conflito nessas verificações (1) e (2), a transação otimista deve ser abortada e a versão descartada.

Em segundo lugar, se a transação sobreviveu à consistência, os efeitos produzidos pela transação otimista sendo finalizada devem ser refletidos no estado permanente do objeto, assim, a atualização deve ser feita pela aplicação nesse estado das operações *write* (de segundo nível) com *timestamp* mais novo da transação otimista sendo finalizada. Além disso, o estado compartilhado pelas transações pessimistas deve ser atualizado, obtendo-se uma cópia do estado permanente atualizado e aplicando-se sobre ele as operações *write* (de segundo nível) com *timestamp* mais novo das transações pessimistas.

Após esses procedimentos, começará o processo de gravação do estado permanente atualizado pela transação otimista, que é efetuado em duas fases, sendo esse processo comum a todas as classes de transações e descrito mais adiante.

Em terceiro lugar, se as gravações dos estados permanentes de cada objeto de aplicação participante da transação forem bem sucedidas, então, deve ser efetuada a consistência entre as transações otimistas ativas, pelo esquema chamado de *forward-oriented concurrency control* (*FOCC*), que significa verificar os conflitos entre as operações da transação em finalização e as de cada transação ativa otimista. Nessas consistências entre as otimistas, são verificados os seguintes conflitos: 1) de primeiro nível, utilizando a tabela **OC**; e 2) de segundo nível, pela comparação dos valores dos atributos inicialmente lidos pela transação ativa com os do estado permanente, recentemente, atualizado, como em [16]. Caso seja detectado conflito nessa consistência, a transação ativa deve ser abortada e a versão descartada. Cada transação ativa que sobreviver à consistência, deve ter sua versão exclusiva atualizada, obtendo-se uma cópia do estado permanente atualizado e aplicando-se sobre ele as suas operações *write* (de segundo nível) com *timestamp* mais novo.

Gravação e Confirmação da Gravação para todos os Tipos de Transação O processo de gravação começa, com o meta objeto pedindo ao *GOT* que grave seu estado permanente mais atualizado, ainda em uma área da memória secundária alternativa. Quando o *GOT* retornar a condição de êxito nessa gravação, o meta objeto, por sua vez deve retornar essa mesma condição ao *GT*. Quando o *GT* receber todos os retornos de gravação de seus participantes com êxito, pedirá a confirmação de gravação a todos os participantes, cujos meta objetos então pedirão ao *GOT* que efetue a gravação do estado permanente na área efetiva da memória secundária. Quando o *GOT* retornar a condição de confirmação da gravação ao meta objeto, este por sua vez deverá retorná-la ao *GT*. Nesse ponto, o *GT* pede aos meta objetos dos participantes que removam os vestígios da transação finalizada, que, terminando, retornam ao *GT*, que, finalmente, pede ao *GOT* que o destrua. Se na remoção dos vestígios, o meta objeto descobre que não há mais transação o compartilhando, ele também pede ao *GOT* a sua destruição.

5. Conclusões

As validações quanto a conflitos nas transações otimistas não são perturbadas pelo assincronismo, enquanto que, para as transações pessimistas *1PL* e *2PL*, é preciso que o pedido de bloqueio fique localizado no evento de invocação da operação, como, em [16]. Essa medida restringe as oportunidades de concorrência, porque na tabela de conflitos **PC1** e **PC2** não se pode explorar as condições de retorno como na tabela **OC**, que é consultada quando todas as respostas já chegaram.

Vimos que podíamos adotar uma versão exclusiva para as transações pessimistas *2PL*, durante a sua primeira fase (a de bloqueio), e deixar a validação para ser efetuada quando essa fase fosse concluída, retirando o tempo de bloqueio inútil das operações até o final dessa fase. Um inconveniente nessa medida é que se o objeto sofrer alguma alteração em seu estado, entre o momento da identificação da operação e o do bloqueio, o caminho na execução de algum método pode invocar alguma operação sem bloqueio. Para contornar tal situação, durante a fase de execução, na pré-condição da invocação, deverá ser verificado se a operação tanto de nível 1 quanto de nível 2 tem bloqueio adquirido para a transação associada. Em caso negativo, se a operação é de nível 1 deverá ser tratada com todos os cuidados dispensados às operações de transação pessimista *1PL*; e se a operação é de nível 2, deverá ser verificado se há ou não conflito do tipo *read/write*, para então a operação ser, respectivamente, reenfileirada ou executada. O problema nessa solução é a oportunidade de *deadlock* em um esquema pessimista *two-phase locking*. Essas possibilidades nos fazem pensar em estender esse esquema pessimista *2PL*, para um segundo esquema *2PL* mais rígido, adotando, nesse segundo, na primeira fase, o bloqueio de todas as operações *read/write* possíveis de acontecer na segunda fase, em vez das somente identificadas.

Melhoramos também as chances de concorrência, nas transações pessimistas, com mudanças na tabela **PC1** e **PC2**: 1) substituímos o significado da condição de conflito, que era baseado nas chances de atualização [16], pela condição de possibilidade de erro na execução, que pode ocorrer mesmo quando as operações são serializadas (um exemplo de operação que pode causar esse tipo de condição é a de retirada de conta corrente); e 2) adicionamos uma terceira condição *U*, que indica que as primitivas referência/atribuição precisam ser serializadas (um exemplo de operação que pode causar esse tipo de condição é a de depósito em conta corrente). Essa condição não foi aproveitada pelas transações pessimistas *2PL*, porque as operações *read/write* que são identificadas na primeira fase ficam bloqueadas durante toda a segunda fase (execução). Mas as transações pessimista *1PL* foram beneficiadas, porque as operações enquadradas nessa situação, podem ter suas operações derivadas de segundo nível bloqueadas, somente enquanto durar a operação, sendo liberadas no evento de resposta, permitindo mais intercalações.

O aproveitamento dessa condição *U*, pelos MCCRs, foi possível porque a estratégia de recuperação *UIP*, usando um estado temporário compartilhado pelas transações pessimistas, em fase de execução, permite a intercalação de operações restritas apenas ao critério de correção. Assim, as pendências de gravação, que forem surgindo na execução, são adiadas para a finalização, podendo ser atendidas de imediato, na execução, caso não haja conflito com suas operações de segundo nível. Com isso, os prejuízos causados por condições de validação *U* ficam reduzidos a espera pelas liberações de operações de segundo nível e do atraso na fase de finalização, devido a espera pela finalização de uma ou mais transações que estejam causando pendência.

Nossa implementação também aderiu à separação [1, 16], em que usamos o artifício de classes aninhadas, para que os diferentes esquemas acessem as estruturas compartilhadas do meta objeto que implementa o objeto atômico, conseguindo uma instância única, que reúne os protocolos para todos os esquemas.

Em um segundo momento, queremos estudar a combinação dos aspectos de tolerância à falha aos MCCRs da nossa implementação, sendo que hoje já temos os objetos e as transações remotas alcançadas, com auxílio do GOT.

Referências

[1] M.S. Atkins and M. Y. Coady. Adaptable concurrency control for atomic data types. *ACM Transactions on Computer Systems*, 10 (3):190-225, August 1992.

[2] Gul Agha. An Overview of Actor Languages. *ACM SIGPLAN Notices*. 21(10), October 1986.

[3] B. R. Badrinath and Krithi Ramamritham. Semantics-based Concurrency Control: Beyond Commutativity. *ACM Transactions on Data Base Systems*. 17 (6). 1992.

[4] Rachid Guerraoui. Modular Atomic Objects. *Theory and Practices of Object Systems*, Wiley & Sons, 2(1), 1995.

[5] Maurice Herlihy. Optimistic Concurrency Control for Abstract Data Types. In *Proceedings of the Fifth ACM Symposium on the Principles of Distributed Computing*. Calgary, Alberta, August 1987.

[6] Maurice Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Transactions on Database Systems*. 15(1), March 1990.

[7] Maurice Herlihy and Willian E. Weihl. Hybrid Concurrency Control for Abstract Data Types. In *Proceedings of the ACM Symposium on Principles of Database Systems*. 1988.

[8] Olivier Jautzy. Highly Customizable Transaction Management for Systems Integration. In *Proceedings of the SDPS World Conference on Integrated Design Process and Technology (IDPT'99)*, Society for Design and Process Science, Jun 1999.

[9] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[10] H. Lieberman. Concurrent object-oriented programming in Act 1. In [YT87].1987.

[11] Paulo Rogério da Motta Junior, Marcelo Germano Alencar, Maria Alice Silveira de Brito. Reflexão e Persistência quando todos os Objetos são Atômicos. In *Simpósio Brasileiro de Linguagens e Programação 2001*. (*SBLP'01*).2001.

[12] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming, System, and Applications (OOPSLA'87)*. 1987.

[13] OMG CORBA – Transaction Service Specification, v1.0. *CORBAservices: Common Object Services Specification*. March 1995.

[14] M. Tamer Özsu. Transactional Models and Transaction Management in Object-Oriented Database Systems. University of Alberta, Department of Computing Science, Edmonton, Alberta, Canada. *http://web.cs.ualberta.ca:80/~ozsu/publications.html*. 1994.

[15] Krithi Ramamritham and Panos K. Chrisanthis. Advances in Concurrency Control and Transaction Processing – An Executive Briefing. *IEEE Computer Society Press*. 1997.

[16] Robert J. Stroud and Z. Wu. Using metobject protocols to implement atomic data types. In *Proceedings of 9th European Conferece on Object-Oriented Programming*, pp 168-189, 1995.

[17] Willian E. Weihl. Local Atomicity Properties: Modular Concurrency Control for abstract Data Types. *ACM Transactions on Programming Languages and Systems*. 11(2), April 1989.

[18] Zhixue Wu and Scarlet Shcwiderski. Reflective Java: Making Java Even More Flexible. Technical Report Report APM.1936.02, APM Limited, Poseidon house, Castle Park, Cambridge, CB30rd, U.K., ANSA Work Programme. 1997.