Estudo das Influências de Trocas de Mensagens Assíncronas nos Mecanismos que Implementam a Atomicidade Local

Maria Alice Silveira de Brito
DICC – Departamento de Informática e Ciência da Computação
IME – Instituto de Matemática e Estatística
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
malice@ime.uerj.br

Abstract

We describe in this paper the issues to be risen if we include the asynchronous message passing on the concurrency control and recovery mechanisms based on serial dependency relation. These mechanisms take into accounting the objects are shared by pessimistic and optimistic transactions. The mechanism can't use the results and return arguments in the serial dependency relation under a pessimistic one-phase locking transaction, thus, the concurrency chances are reduced. The optimistic transactions execute without synchronization, relying on commit-time validation to ensure serializability, thus, the asynchronous mode is don't care at run-time. The behavior of pessimistic two-phase locking transactions in object was preserved. These results confirm the chance to include asynchronous message passing in atomic object implementation.

1. Introdução

Este trabalho se situa no contexto de construções em linguagens de programação, em particular, sobre a facilidade de transação a ser oferecida por uma linguagem de programação orientada a objeto, com enfoque no seu protocolo de controle de concorrência e recuperação.

Para a integração da facilidade de transação a linguagens de programação, os modelos, que têm sido adotados, foram estendidos para o processamento das transações diferentes das tradicionais e surgiram, em geral, conduzidos por uma solução específica a cada tipo de necessidade ditada por novos tipos de aplicações ou por novos tipos de recursos. Essa seqüência de novas soluções resultou em uma diversidade de modelos avançados de transações (*ATM*s), ver [13, 29, 34, 5, 14, 16, 41, 6, 24], complicando tanto a escolha quanto o emprego desses modelos, na tarefa de programação da aplicação.

As abordagens, que levaram em conta os dois aspectos seguintes – a) a ferramenta genérica de programação para simplificar o emprego/extensão de *ATM*s e b) a exploração dos recursos de orientação a objeto – aparecem nas três linhas, a seguir: 1) as que possibilitaram extensão de modelos de transação, encontradas em [3, 7, 8, 11, 28, 19], aproveitando parcialmente as oportunidades oferecidas pela orientação a objeto, para alcançar maior concorrência; 2) a encontrada nos ambientes transacionais [39, 12, 31, 32, 26, 36], que permitiu explorar a semântica das operações dos objetos, através do conceito de atomicidade local [38], não oferecendo, no entanto, facilidades para a extensão de modelos de transação, porque a gerência da transação se situa na plataforma sobre a qual a aplicação executa; e 3) as encontradas, em [35, 40, 21], que empregaram os dois aspectos seguintes: 3-a) programação reflexiva [27, 23] para permitir a programação dos modelos de transação, programados em classes, nos módulos da meta arquitetura; e 3-b) atomicidade local nos mecanismos de controle de concorrência, no módulo monitor (meta objeto) que é responsável pela implementação do objeto ativo [25, 2], nessa meta arquitetura.

Essa terceira linha de abordagem aproveitou as seguintes oportunidades: 1) a atomicidade local, que além de explorar a semântica das operações, trazendo mais chances de concorrência, trouxe oportunidade para a convivência de políticas diferentes concorrerem no mesmo objeto de aplicação [17, 20,38, 18, 1, 35]; 2) a serialização global [38, 15] e a anomalia de herança tempo (Wegner 1987, Kafura e Lee 1989), ver [30, 4], podendo ser resolvidas pelo uso de reflexão na implementação do objeto atômico; e 3) a gerência da

transação sendo programada em classe, prevista uma instância para cada transação que surgir no ambiente de execução.

Nosso modelo original de meta arquitetura, em [10], combina duas finalidades principais: 1) integrar a facilidade de transação a linguagens de programação orientada a objeto, e 2) permitir a extensão e simplificar o emprego de *ATM*s, enquadrando-se nessa terceira linha de abordagem das ferramentas genéricas, aproximando-se do modelo encontrado, em [35]. Essa aproximação é identificada pela forma de ter implementado a atomicidade local em um meta objeto que deve ser associado ao objeto de aplicação, para que este assuma a característica de um objeto ativo, além da de objeto atômico, com a diferença de que as particularidades do controle de concorrência de cada política de transação ficaram implementadas juntas nos mecanismos de controle de concorrência e de recuperação, situados no monitor (meta objeto). Assim, em vez de uma classe de meta objeto para cada política de transação, como em [35], todas as particularidades do controle de concorrência e de recuperação de cada política de transação ficam reunidas no mesmo módulo de implementação, devendo, então, ser levado em conta que apenas uma classe de meta objeto serve para instanciar os monitores de todos os objetos de aplicação, no ambiente de execução. Estamos estudando soluções com a facilidade de classes aninhadas, oferecida em linguagens de programação orientadas a objeto, com a intenção de simplificar a extensão (herança) dos mecanismos para modelos de transação diferentes.

Nesses modelos da terceira linha, o mecanismo de controle de concorrência e recuperação deve permitir a negociação entre as diferentes políticas de transações, que compartilhem simultaneamente cada objeto atômico, além de permitir que novas políticas de transações sejam estendidas e tenham sua coexistência negociada entre essas diferentes políticas.

Como essas questões, acima, ficam resolvidas, nos mecanismos de controle de concorrência e recuperação, a incorporação da capacidade de troca de mensagem assíncrona combinada à granularidade de concorrência em cada operação em cada objeto poderia influenciar o funcionamento desses mecanismos. Assim, resolvemos estudar quais seriam essas influências, nos mecanismos que implementam atomicidade local, cujo material predominante foi o protocolo apresentado em [20] e as implementações [1, 35], baseadas em [38, 20, 18]. Cabe, neste ponto, ressaltar que pretendemos estudar os mecanismos apresentados em [18] e talvez chegar a melhores resultados dos que os já alcançados até aqui. Nosso objetivo neste artigo é descrever o processo e os resultados desse estudo inicial, em que já temos um modelo de computação, que está em fase de implementação, cujas características estão apresentadas na seção 4.

Chegando ao final desta seção de introdução, consideramos apropriado apresentar as definições dos dois principais conceitos levados em conta, nessas soluções, os de atomicidade local e objeto ativo, a seguir.

Atomicidade local A atomicidade local incorpora o mecanismo de controle de concorrência local 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 [38]. Essa abordagem propõe que os objetos de dados compartilhados sejam implementados de uma forma que seja garantida atomicidade às transações que os acessam. Esses objetos que devem prover sincronização e recuperação apropriadas são chamados de objetos atômicos e a atomicidade de uma transação somente será garantida se todos os objetos compartilhados por ela forem atômicos. 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. A atomicidade global na transação conta com as atomicidades locais nos objetos, necessitando que a serialização global seja garantida e, para isso, tendo que haver compatibilidade entre as atomicidades locais, como observa Willian Weihl [38].

Objeto ativo O conceito de objeto ativo – inspirado em *Actor languages* [25, 2] – é apresentado como objetos que se comportam, cooperativamente, trocando mensagens. As operações sobre um objeto servidor podem ser invocadas pela execução concorrente de objetos clientes. No modelo de objeto ativo, é assinalado que os próprios objetos servidores têm a responsabilidade de responder aos pedidos concorrentes que são recebidos e enfileirados, provendo uma forma de passagem de mensagem assíncrona.

Este artigo é constituído de mais três seções: a segunda que apresenta os fundamentos e trabalhos relacionados; a terceira, o estudo em si, dividida em três sub-seções dedicadas, como a seguir: em 3.1, sobre o protocolo básico de controle de concorrência e recuperação híbrida, encontrado em [20], indicamos os aspectos que ficam incompatíveis com o assincronismo; em 3.2, sobre a implementação encontrada em [1], que inclui as verificações de conflitos entre as classes de transação, fazemos uma lista dos aspectos de verificação e recuperação e suas influências no assincronismo e na propagação pela composição, além das estratégias de recuperação *update in place* e *deferred update*; e, em 3.3, sobre a implementação encontrada em [35], que inclui o artifício de verificação de conflitos em dois níveis, no nível das operações da interface do objeto e das operações privadas e no nível

das operações primárias de referência/atribuição, além da separação em meta objetos para cada protocolo otimista ou pessimista, apontamos a oportunidade de dois níveis de conflitos para as transações pessimistas também; a quarta seção que indica as características para um novo protocolo, de acordo com os resultados desse estudo; e a quinta seção, as conclusões.

2. Fundamentos e trabalhos relacionados

Dirigindo a atenção para certas características da orientação a objeto, como, seus tipos abstratos, que definem a semântica dos dados, destacamos os dois fatos, a seguir, que contribuem nos mecanismos de controle de concorrência, 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) um objeto pode adquirir conhecimento das relações de conflito entre suas operações. Esse conhecimento das relações de conflito entre as operações pelo próprio tipo abstrato de dados torna a concorrência de operações mais flexível, melhorando o desempenho na execução das transações concorrentes. Esses recursos vêm sendo explorados tanto para objetos simples quanto para objetos complexos, como observado em [33], 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, da estrutura do objeto e da estrutura da transação, facilita o estudo dessas questões 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 [38, 20, 18, 1, 35, 9].

A propriedade de atomicidade local (na dimensão da estrutura do objeto), que combina os próprios recursos de tipos abstratos de dados com as técnicas em controle de concorrência baseadas em semântica, inicialmente, foi especificada por conflitos definidos por comutatividade [Wei1984] e, posteriormente, definidos por invalidação [20].

No trabalho apresentado em [38], são abordadas três tipos de propriedades locais – *dynamic atomicity*, *static atomicity* and *hibrid atomicity* – que caracterizam, respectivamente, o comportamento das três classes comuns de protocolos: *dynamic protocols*, tais como os *two-phase locking*, em que a ordem de serialização de transações é determinada pela ordem em que elas acessam os objetos; *multiversion timestamp-based protocols*, em que a ordem de serialização das transações é baseada sobre uma *predetermined total order*; e *hibrid protocols*, que usa uma combinação dessas técnicas.

Nessa linha, em [20, 18], as chances de concorrência foram aumentadas ao serem desenvolvidos novos protocolos que substituíram a comutatividade. Em [20], é definido um *locking protocol* que é menos restritivo do que o protocolo baseado em comutatividade, sendo mostrado que *lock-conflict relations induced by dependency* podem ser mais frágeis do que *induced by the commutativiy-based protocol*. Em [18], é desenvolvido um mecanismo de controle de concorrência otimista, em que a validação é baseada nos conflitos pré-definidos entre pares de operações. Essa abordagem é análoga a que emprega *locking*, em [20], que também usa conflitos pré-definidos entre pares de operações, para introduzir retardos. Em [9], o protocolo é baseado em comutatividade com adição de *recoverability* à tabela de conflitos, que deve ser derivada diretamente da especificação do tipo do objeto.

A convivência de políticas diferentes em um mesmo objeto, neste parágrafo, é examinada, enfocando os modelos que foram baseados na propriedade de atomicidade local híbrida, e, em particular, no protocolo que se encontra em [20]. A propriedade de atomicidade local híbrida 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 [17], os conflitos em um objeto têm a possibilidade de serem tratados de forma otimista ou pessimista, com técnicas que satisfazem a atomicidade local híbrida [38].

Em [20], é definido um *hibrid locking protocol* que se baseia em uma *lock conflict relation* para determinar se há conflito de uma operação com outras que estiverem *lock* por outras transações, sendo que nesse caso a operação é descartada e tentada mais tarde, quando os *locks* forem liberados. A restrição básica que governa os *lock conflicts* é a noção de *dependency*, quer dizer, uma operação p e outra operação q não podem executar concorrentemente se uma depende da outra. Quando a transação termina, suas intenções, que se encontram numa lista associada a essa transação, são *merged* com o *committed state*, em *timestamp order*. Ao final dessa etapa de *merge*, os *locks* e as intenções dessa transação *committed* são liberadas. Esse protocolo permite mais concorrência do que aqueles baseados em comutatividade. Ele usa *timestamps* gerados quando a transação *commits* para prover mais informações sobre a ordem de serialização das transações.

Em [18], é proposto um mecanismo de controle de concorrência otimista chamado *conflict-based validation*, construído por conflitos pré-definidos entre

pares de operações. Essa abordagem é a otimista análoga aos mecanismos de *locking* [20], que usam conflitos pré-definidos similares, para gerar atrasos nos bloqueios. Em outras palavras, essa invalidação define um conflito entre duas operações se a execução de uma invalida a execução de outra, sendo essa a relação que forma a base para estabelecer a serialização.

Em [9, 34, 8], o protocolo, que é baseado em comutatividade e *recoverability*, inclui na tabela de conflitos também as informações de dependência de *commit*.

Na outra parte da atomicidade, a recuperação, em protocolos híbridos [20], o momento de efetuar o *update* do estado permanente do objeto tem sido tratado pela estratégia *deferred update* (*DU*), embora essa estratégia *DU* dê maiores possibilidades de aborto ao final da transação do que a estratégia *update in place* (*UIP*).

Em [1], foi desenvolvido um servidor[1] para um objeto específico, uma fila, compartilhado por transações. A construção desse método de controle de concorrência híbrida foi baseada em [17, 20], permitindo que controles otimistas ou pessimistas sejam aplicados a um objeto (*per conflict-type granularity*), de forma pré definida ou adaptável dinamicamente. Os conflitos entre transações são resolvidos de forma diferente dependendo da coincidência ou não entre suas classes: (*o*) otimista, (*p*) pessimista e (*h*) híbrido. Se as transações são de uma mesma classe, a verificação é feita apenas pela consulta à tabela de conflitos. Se as transações são de classes diferentes, a verificação é feita, no momento de validação, na fase de *commit*, obedecendo a seguinte linha de prioridade: $p > h > o$. Sobre a estratégia para recuperação, foi mantida a estratégia *deferred update*, porque os autores consideraram que não estava claro como a estratégia *update in place* poderia ser incorporada ao servidor híbrido sem aumentar o número de conflitos e o número de abortos. Assim, cada transação que concorre em um objeto apresenta uma lista de intenções, que é aplicada ao estado permanente do objeto após a validação que é efetuada no momento de *commit*.

Em [35], os tipos de dados atômicos são implementados com o uso de meta objeto *protocol* e a classe do meta objeto é amarrada ao objeto, em tempo de execução, de acordo com a política e protocolo da transação, associada a mensagem recebida. Essa classe pode ser estendida para implementar diferentes protocolos de transações, provendo flexiblidade, e, inicialmente, foram oferecidas duas classes para esse meta objeto: uma que implementa a atomicidade local para as transações de política pessimista e outra para as otimistas.

Para economizar acesso a disco, principalmente em objetos considerados grandes, a representação dos objetos foi projetada como estruturas em árvore, com sub-objetos primitivos nas folhas. Uma cópia de um sub-objeto da árvore estruturada de um objeto somente deve ser obtida do objeto físico, se uma dada invocação necessitar acessá-la, sendo considerada a cópia corrente. Uma cópia dessa corrente, associada a uma transação, deve ser obtida somente se uma dada invocação necessitar atualizá-la. A partir daí, qualquer operação *read/write*, associada a transação que causou essa invocação, deverá ser efetuada nessa segunda cópia.

Para as transações de política pessimista, foi implementado o protocolo *two-phase locking*, em que os conflitos são verificados entre operações *read/write*, na fase de execução. Uma operação adquire um *lock*, dependendo dos *lock*s adquiridos por outras transações. O estado permanente é salvo em *snapshot*s e as operações são realizadas sobre o estado permanente. Durante o *commit*, os *lock*s são apenas liberados e se houver aborto, o estado permanente é restaurado com os *snapshot*s.

Para as transações de política otimista, é adotado um protocolo chamado *dual-level validation* (*DLV*) *method*. Nesse método, o funcionamento do controle de concorrência, na fase de consistência, fica dividido nos dois níveis seguintes: 1) no nível lógico, em que são verificados os conflitos entre as operações do tipo abstrato, consultando a tabela de conflitos; e 2) no nível físico, em que a validação é feita, comparando os valores lidos com os que se encontram no estado permanente (corrente). Caso seja detectado algum conflito em qualquer nível, as operações deverão ser re-executadas.

Na seção seguinte, apresentamos os aspectos que seriam influenciados se adicionássemos a capacidade de troca de mensagem assíncrona, aos três protocolos de controle de concorrência e recuperação, encontrados em [20, 1, 35], indicando as soluções que deveriam ser adotadas.

3. Implementações de protocolo híbrido

3.1. O protocolo básico

O protocolo apresentado em [20] é descrito operacionalmente por uma máquina de estado *LOCK*, cuja linguagem consiste de um conjunto de eventos. A máquina usa uma relação de conflito particular, *Conflict*, para testar se uma operação tem conflito com outra. É assumido que essa relação *Conflict* seja uma matriz simétrica. Para descrever o protocolo, entretanto, não é preciso fazer quaisquer outras suposições sobre a relação de conflito usada. Nesse artigo [20], é mostrado que as relações de conflito derivadas de relações de dependência são suficientes e necessárias para assegurar que a implementação esteja correta, no sentido de que toda história completa em *L* (*LOCK*) é atômica e híbrida.

Um estado ***s*** de *LOCK* consiste de quatro componentes: *s.pending*, *s.intentions*, *s.committed*, and

[1] O termo servidor adotado em [1] pode ser entendido, no contexto do nosso trabalho, como um monitor.

s.aborted.

s.pending é uma função parcial das transações para eventos de invocação;

s.intentions é uma função total das transações para seqüências de operações;

s.committed é uma função parcial das transações para *timestamp*s.

s.aborted é um conjunto de transações.

s.*pending* Esta estrutura registra invocações pendentes para transações. Desde que cada transação seja inicialmente quiescente, s.*pending* é indefinida para todas as transações no estado inicial de LOCK.

s.*intentions* Esta estrutura registra a seqüência de operações executadas por cada transação. No estado inicial de LOCK, *s.intentions* maps cada transação para a seqüência vazia.

s.*committed* Esta estrutura permite-nos contar quais transações foram committed, e para cada transação committed registra seu *timestamp*. S.*committed* é inicialmente indefinida para todas as transações.

s.*aborted* Esta estrutura registra o conjunto de transações que foram abortadas e é incialmente vazia.

Se **s** é um estado de LOCK, definimos s.*completed* como sendo

$\{s.aborted \cup \{a \mid s.committed\ (a) \neq \perp\}$.

s.*completed* Esta estrutura assim consiste de todas as ações que tenham ou committed ou abortado. Se $Q \notin$ s.*completed*, definimos *View* (*Q*, *s*) como a seqüência de operações obtida por concatenação das listas de intenções para otodas as transações committed em ordem de timestamp e então concatenada a lista de intenções de *Q*.

As transições de LOCK são os eventos envolvendo *X*, suas pré-condições e pós-condições são descritas abaixo. Por brevidade, nós assumimos que todas as histórias de entradas são bem formadas (Well-formedness poderia ser verificada explícitamente por adição de mais componentes de estado e pré-condições).

Na descrição das transições, nós escrevemos pré-condições e pós-condições para eventos, usando a convenção que *s*' denota o estado antes do evento indicado e que *s* denota o estado depois do evento. Em adição, um componente do estado que não é mencionado na pós-condição para um evento é assumido a ser destrocado pela ocorrência daquele evento.

Eventos *invocation*, *commit*, e *abort* são entradas controladas pela transação, assim, suas pré-condições são verdadeiras. A transição para cada evento é bem simples, o evento é simplesmente registrado no estado de LOCK.

<inv, X, Q>

Postcondition

s.pending = s'.pending [Q → inv]

<commit (t), X, Q>

Postcondition

s.committed = s'.committed [Q → t]

<abort, X, Q>

Postcondition

s.aborted = s'.aborted ∪ {Q}

<res, X, Q> /*Response events are somewhat more complicated */

Precondition

s'.pending (Q) ≠ ⊥

Q ∉ s'.completed

Let q = <s'.pending(Q),res>

View(Q,s') •q ∈ *Serial*

for all transactions P ∉ s'.completed ∪ {Q},

and for all operations p in s'.intentions (P),

<p,q> ∉ *Conflict*

Postcondition

s.pending = s'.pending [Q → ⊥]

s.intentions =

s'.intentions [Q → s'.intentions(Q)•q]

Controle de Concorrência. Podemos observar, pela especificação acima, que os mecanismos de controle de concorrência se concentram no tratamento do evento *response*. Para retornar um *response* a uma transação, são exigidos vários requisitos. Primeiro, a transação deve ter uma invocação pendente. Segundo, a transação não deve estar ainda completa. Terceiro, a operação (constituída do par <*invocation*, *result*>) deve estar legal em "*view*". Finalmente, a operação não deve apresentar conflito com qualquer outra operação já executada por outra transação ativa. Se todos esses requisitos são atendidos, o evento *response* pode ser aceito, causando a remoção de *pending invocation* do estado e a *intentions lists* para a transação será atualizada, registrando a nova operação.

Recuperação Notar que s.*intentions* é retida para todas as transações, inclusive as transações committed. Assim, o "estado committed" é simplesmente as *intentions lists* para as transações committed, organizado pela ordem de *timestamp*. Essa abordagem é claramente impraticável. No entanto, ela nos permite descrever o protocolo numa forma simples e geral. Todos os outros métodos parecem ser casos especiais do uso dessas *intentions lists*, no sentido de eles não registram qualquer informação a mais sobre o passado no estado. Em adição, alguns outros métodos de recuperação parecem requerer restrições na concorrência mais do que o suficiente para *intentions lists*.

Considerações As necessidades de assincronismo e de controle de concorrência na propagação das mensagens pela composição ou pelos parâmetros, admitidas em nossa meta arquitetura, causam uma certa dificuldade com alguns dos princípios, em que se baseia o protocolo, acima.

Uma delas é a necessidade de cada transação *P* dever esperar pelo evento *response* correspondente à sua última invocação antes de invocar a operação seguinte, exigindo que uma transação tenha uma única pendência de invocação por vez. Se pretendemos admitir pedidos da mesma transação, simultaneamente, dois aspectos, no protocolo, são influenciados e percebidos de imediato: 1) a tabela de conflitos não poderá ter entradas com resultados de operações para as transações pessimistas, porque a verificação de conflitos para o modo assíncrono da mensagem deve ser feito no momento da invocação da operação e não em seu retorno; e, assim, 2) o evento de invocação, para a transação pessimista, deverá examinar o conflito, na pré-condição, antes da mensagem ser atendida.

Podemos observar que as chances de concorrência, para as transações pessimistas, diminuem, porque não poderemos testar o resultado retornado, precisando garantir, previamente, que a operação executará corretamente, provavelmente, restringindo as possibilidades de atendimento da mensagem, que chega ao objeto. Para as transações otimistas, como a consistência é efetuada, no final, os resultados se encontram nos registros e poderão ser examinados. Notar que o protocolo só tem uma espécie de verificação de conflitos, baseada em *lock conflict relation*, para determinar se há conflito de uma operação com outras que estiverem *lock* por outras transações. No caso de conflito, a operação é descartada e tentada mais tarde, quando os *locks* forem liberados. Vamos, a seguir, ver como, em [1], este protocolo foi implementado e tratou as classes pessimista, otimista e híbrida, para o protocolo [20].

3.2. Tratamento das classes pessimistas, otimistas e híbridas

Em [1], que é uma implementação de [20], os conflitos podem ser preenchidos na tabela, específicos a transações pessimistas e a transações otimistas, com a pretensão de ser um servidor híbrido. As ações, durante a detecção de conflitos, são as seguintes: nos conflitos otimistas, as transações devem ser abortadas em tempo de *committing* e nos conflitos pessimistas, as transações devem ter suas operações adiadas. Embora esse trabalho [1] ressalte a oportunidade das transações se adaptarem a melhor política de acordo com o comportamento da aplicação, consideramos que deva ser aproveitada, somente, a alternativa denominada *preassignment*, em que a classe da transação é estabelecida, inicialmente, pelo programador da aplicação que decide qual classe, quer dizer, não aproveitando a oportunidade de adaptação da aplicação à política de transação mais adequada, que poderia ser decidida pelo protocolo.

As classes apresentadas nessa proposta de servidor são as seguintes: otimista (*o*), pessimista (*p*), ou híbrida (*h*). A informação sobre essa classe, associada à transação, vai indicar como os conflitos devem ser tratados, tanto na tabela, quanto na ações posteriores. Para uma versão inicial, estamos considerando que no nosso modelo de protocolo, somente as transações de classe otimista e pessimista devam ser tratadas, excluindo as transações híbridas. Essa decisão foi conduzida pela nossa idéia de desenvolver, no monitor (meta objeto), um protocolo que trate todas as atomicidades locais correspondentes as atomicidades globais das diferentes políticas e protocolos de concorrência das transações, que possam compartilhar o objeto, independentemente, das suas características seqüenciais, para garantir a serialização global.

A seguir, apresentamos como são tratadas a concorrência, a recuperação e as chances de *deadlock*, acompanhadas de discussões sobre as nossas necessidades.

Controle de concorrência As ações relacionadas ao controle de concorrência, de acordo com o servidor original, em [1], são tomadas em tempo de execução, como podemos ver, a seguir:

- Se uma transação Tj, da classe (o), executa uma operação p e qualquer outra transação Tk executa q, existindo um conflito pessimista (p,q), esse conflito não é registrado sobre Tj, se Tj está sendo validada enquanto Tk está ativa. Isso quer dizer que Tj não é abortada ou adiada e Tk não é invalidada.
- Se uma transação Tj, da classe (p), executa uma operação p e qualquer outra transação Tk da classe (o) ou (h) executa q, existindo um conflito pessimista (p,q), nenhum P-lock para q é guardado por Tk e então Tj não é adiada.
- Se uma transação Tj, da classe (p), executa uma operação p e qualquer outra transação Tk da classe (p) executa q, existindo um conflito pessimista (p,q), P-lock para q é guardado por Tk e então Tj é adiada.

Pela nossa proposta, não estaremos considerando a classe híbrida (h), assim, as ações se tornam, como, a seguir:

- Se uma transação Tj, da classe (o), executa uma operação p e qualquer outra transação Tk executa q, existindo um conflito pessimista (p,q), esse conflito não é registrado sobre Tj, se Tj está sendo validada enquanto Tk está ativa. Isso quer dizer que Tj não é abortada ou adiada e Tk não é invalidada.
- Se uma transação Tj, da classe (p), executa uma operação p e qualquer outra transação Tk da classe (o) executa q, existindo um conflito pessimista (p,q), nenhum P-lock para q é guardado por Tk e então Tj não é adiada.
- Se uma transação Tj, da classe (p), executa uma operação p e qualquer outra transação Tk da classe (p) executa q, existindo um conflito pessimista (p,q), P-lock para q é guardado por Tk e então Tj é adiada.

Podemos ver que os conflitos entre operações de transações de classes diferentes não são resolvidos em tempo de execução, ficando para o tempo de *commit*, na fase de validação. As facilidades *P-locks* e *O-flags*, utilizadas em servidores estáticos, foram empregadas no

servidor dinâmico proposto em [1]. Uma outra tabela deve ser criada, em que as entradas podem representar dois tipos de estratégias para a solução de conflito, como, podemos ver, a seguir. Nos procedimentos de *validação* de transações otimistas e dos componentes otimistas de transações híbridas são adotados a estratégia *suicidal* (*S*). Nos procedimentos de *commitment* dos componentes pessimistas das transações otimistas e das transações híbridas são adotados a estratégia *commit-and-kill* (*C&K*). Observar que transações da classe (*h*) primeiro tentam validar de forma otimista, e, se bem sucedidas, *commit* em um simples passo atômico, chamado (*val&com*).

Conflitos entre transações de classes diferentes			
	To (*act*)	*Th* (*act*)	*Tp* (*act*)
To (*val*)		S	*S*
Th (*val&com*)	*C&K*		*S*
Tp (*com*)	*C&K*	*C&K*	

A linha um da tabela indica que uma transação otimista que esteja na fase de *commit* deverá cometer suicídio durante sua validação se ela tiver operações que geram conflitos com operações de transações ativas híbridas e pessimistas.

A linha dois da tabela indica que uma transação híbrida que esteja tentando validar e *commit* será bem sucedida, se ela gera conflito com uma transação ativa otimista, mas a transação da classe (*o*) será *killed*. Se a transação híbrida que esteja tentando validar e *commit* gera conflito com uma transação ativa pessimista, deverá cometer suicídio.

A linha três da tabela indica que uma transação pessimista que esteja na fase de *commit*, sempre será bem sucedida, mas se existir conflitos com operações de transações de classes (*o*) ou (*h*), essas serão *killed*.

Em resumo, os conflitos entre transações de classes diferentes são resolvidos pelo aborto da transação na seguinte ordem de prioridade, em relação a sua classe: $o < h < p$. Essa decisão reduz o disperdício de trabalho.

Pela nossa proposta, em que são consideradas apenas as classes pessimista (*p*) e otimista (*o*), nos procedimentos de validação e de *commitment*, a parte correspondente às transações híbridas também não é considerada, em outras palavras, os componentes tratados são os correspondentes, apenas, às transações otimistas e pessimistas. Em resumo, nos procedimentos de validação de transações otimistas são adotadas a estratégia *suicidal* (*S*); e nos procedimentos de *commitment* dos componentes pessimistas das transações otimistas são adotadas a estratégia *commit-and-kill* (*C&K*). Assim, a tabela que representa a estratégia para a solução de conflito entre classes diferentes é obtida, diretamente, pela remoção dos elementos que representam as transações híbridas, da tabela original, como podemos ver, a seguir.

Conflitos entre transações de classes diferentes

	To (*act*)	*Tp* (*act*)
To (*val*)		*S*
Tp (*com*)	*C&K*	

A linha um da tabela indica que uma transação otimista que esteja na fase de *committing* deverá cometer suicídio durante sua validação se ela tiver operações que geram conflitos com operações de transações ativas pessimistas.

A linha dois da tabela indica que uma transação pessimista que esteja na fase de *committing*, sempre será bem sucedida, mas se existir conflitos com operações de transações de classes (*o*), essas serão *killed*.

Em resumo, os conflitos entre transações de classes diferentes, sem a consideração da classe das transações híbridas, são resolvidos pelo aborto da transação, na seguinte ordem de prioridade: $o < p$, em relação à sua classe.

O estado *s* do protocolo, apresentado em [20], é acrescido dos seguintes componentes, no protocolo apresentado em [1]:

s.O-Flags (*p*) registra as transações otimistas ativas que executaram a operação *p*;

s.P-Locks (*p*) registra as transações pessimistas ativas que executaram a operação *p*;

s.waiting registra {<*inv,X,P*>, (*P,Q*)) | *P* espera pela liberação de um *lock* por *Q* };

s.wakeup registra a invocação <*inv, X, P*>, que será tentada novamente.

Recuperação A recuperação de conflitos é alcançada pela definição de cada objeto constituído de dois componentes: um estado permanente e um conjunto de listas de intenções. Um estado permanente do objeto registra os efeitos de transações que tenham terminado com sucesso. Existe uma lista de intenções, registrando tentativas de trocas, para cada transação ativa que tenha acessado o objeto pela execução de um evento. Quando uma transação termina, trocas na sua *intention's list* são aplicadas ao estado permanente do objeto. Se, no entanto, a transação aborta, sua lista de intenções é simplesmente descartada. Esse método é chamado de *deferred update* (*DU*) *strategy*, em que *intention's list* ou *private workspace* são usados como *buffer modifications* para o estado permanente do objeto até que uma transação termine.

As duas justificativas para a adoção dessa estratégia para recuperação, em [1], são: 1) que *DU recovery* trabalha separado da concorrência, ocorrendo a única chance de interação entre eles, quando a *intention's list* é aplicada; e 2) que o controle de concorrência híbrido funciona bem com a estratégia *DU*. No entanto, é chamada a atenção, nesse trabalho, para o fato de que, em uma abordagem puramente pessimista, uma *update in place* (*UIP*) *recovery method* pode prover acesso mais eficiente para alguns tipos abstratos de dados, aparecendo examinada, no parágrafo seguinte. A política otimista é geralmente escolhida nos casos de

muita leitura e pouca alteração, sendo ainda interessante adotar a estratégia *DU*.

A estratégia *update in place* deveria ser considerada para o protocolo de controle de concorrência de acordo com as nossas necessidades. Essa decisão conduz a um aumento de possibilidades nas operações do nível abstrato que não conflitam, mas que causam dependências entre si. Chamamos a atenção, neste ponto, que para ser possível a detecção das dependências entre as operações, que não causam erro na execução mas causam dependência, o uso de um artifício de validação entre as operações *read/write* causadas pelas transações é adequado. Esse mecanismo aparece na abordagem apresentada em [35], chamado de *dual level validation* (DLV), cuja discussão é retomada na seção dedicada a esse protocolo. mais à frente.

Agora, vamos examinar as ações de recuperação combinadas à situação em que toda a mensagem que chegar a um objeto poderá propagar outras mensagens pela sua composição ou por parâmetros, em transações com política pessimista. Essas mensagens propagadas também deverão ser submetidas ao controle de concorrência localizados nos objetos destinos. Quando uma transação é *committed*, a aplicação da *intention's list* combinada à propagação de mensagens torna-se complicada. Assim, resolvemos examinar as duas estratégias: *deferred update* (*DU*) e *update in place* (*UIP*), tentando evitar a re-execução e levando em conta: a) os conflitos no nível abstrato e as políticas das transações, e b) as dependências causadas nos estados, quer dizer, os conflitos entre as operações *read/write* entre as transações. Para esse exame, apresentamos um exercício simples, no qual consideramos um objeto que possui um atributo *a* e que é compartilhado por duas transações pessimistas T1 e T2.

T1 pode pedir para incrementar *a* e, em seguida, T2 também. Nessa situação, apesar de T2 ser dependente do *committing* de T1, por causa do valor de *a*, que foi incrementado por T1, será incrementado por T2, a operação *increment* no nível abstrato não gera conflito.

1) Na situação seguinte: "T2 *commits* e depois T1 *commits*."

a) Se nós adotarmos a estratégia *deferred update* (*DU*), nós necessitaremos de *redo* de T1, porque, durante a fase de *committing*, o valor de *a* no estado permanente corresponderia ao valor incrementado obtido por T2. Quando T1 invocasse a operação *increment a*, o valor de *a* não seria esse. Nessa estratégia, as transações na fase de *committing* não necessitariam esperar por outra transação que estivesse causando dependência no estado compartilhado.

b) Se nós adotarmos a estratégia *update in place* (*UIP*), o valor de *a* já teria sido atualizado, e, assim, o estado estaria pronto. Nessa estratégia, as transações na fase de *commit* necessitariam esperar pela outra transação que estivesse causando pendência no estado compartilhado.

2) Na seguinte situação: "T2 termina e depois T1 aborta".

a) Se nós adotarmos a estratégia *deferred update* (*DU*), nós apenas necessitaremos descartar a *intentions list* de T1. Nessa estratégia, as transações na fase de *commit* não necessitariam esperar por outra transação que estivesse causando pendência no estado compartilhado.

b) Se nós adotarmos a estratégia update in place (UIP), além das transações na fase de commit necessitarem esperar pela outra transação que estiver causando pendência no estado compartilhado, nós necessitaríamos considerar duas alternativas possíveis:

i) T2 necessitaria de re-execução sobre o estado permanente; ou

ii) T2 poderia também abortar.

Vamos examinar as ações necessárias, acima, levando em conta: as duas estratégias, a existência de conflitos no nível de estado, quer dizer, entre operações *read/write*, e a opção de não re-execução:

Update in place (UIP) strategy Nessa estratégia, durante o *committing* de uma transação, se houver dependência entre transações, poderão surgir as situações, na seguinte ordem:

1) **sempre será necessária** a espera pelo *commit* de todas nessa situação;

2) **não será necessária** a re-execução, se todas forem *committed* com sucesso;

3) **será necessário** o aborto ou a re-execução das transações que dependem de alguma transação que chegar a abortar.

Deferred update (DU) strategy Nessa estratégia, durante o *committing* de uma transação, se houver dependência entre transações, poderão surgir as situações, na seguinte ordem:

1) **não será necessária** a espera pelo *commit* de qualquer transação nessa situação;

2) **sempre será necessária** a re-execução, porque é necessário fazer o *merge* entre a sua *intentions'list* e o estado permanente;

3) **não será necessário** o aborto ou a re-execução das transações que dependem de alguma transação que chegar a abortar.

Se nossa intenção é economizar as re-execuções ou até mesmo evitá-las, a *update in place (UIP) strategy* traz vantagens em relação a *deferred update (DU) strategy*, como podemos ver, nas ações enumeradas acima. A desvantagem, na estratégia *DU*, se deve a necessidade de sempre re-executar no momento de proceder o *merge* entre a sua *intentions'list* e o estado permanente, enquanto que na estratégia *UIP*, somente haverá re-execução ou aborto de alguma transação, caso ela se encontre dependente de outra transação que tenha sido abortada. Se adotarmos, na estratégia UIP, a opção de re-executar em vez de abortar, ainda assim o número de re-execuções será menor do que na estratégia *DU*. Assim, nossas considerações são de que para as transações pessimistas, deve ser adotada a estratégia

UIP com aborto, no caso de alguma transação se encontrar dependente de outra que tenha sido abortada. Em um segundo momento, a opção de re-execução para esse caso com *UIP*, pode ser cautelosamente experimentada.
Considerações. Para garantir a serialização global [38, 15], estamos levando em conta que a atomicidade local deverá refletir a atomicidade global da transação, associada às mensagens que estiverem chegando ao monitor associado ao objeto, em que fica localizado o protocolo. Assim, não deveremos aproveitar, para o nosso modelo, a oportunidade da classe de política de transação (*o*, *p*, ou *h*) poder ser adaptada dinamicamente ao objeto, nos restringindo apenas ao uso do modo *preset*. Além disso, com essa restrição, as classes das transações para as mensagens que chegam aos objetos não deverão depender desse objeto, comportando-se em todos os objetos participantes com a mesma política da transação. Essa consideração elimina a oportunidade da política híbrida *h*, sem, no entanto, impedir a convivência de transações pessimistas e otimistas compartilhando o mesmo objeto.

Quanto às estratégias para recuperação, se a ação de re-execução traz dificuldades, quando o controle está propagado por todos os objetos, a estratégia *udate in place* (*UIP*), para as transações pessimistas, foi considerada a mais apropriada, devendo ser combinada à decisão de abortar, no caso de transações dependerem de alguma transação que chegue a abortar. Para a política otimista, continuaremos ainda com a estratégia *DU*, porque essa política geralmente é adotada, quando a aplicação causa poucas alterações nos dados com os quais ela trabalha.

3.3. Tratamento de conflitos em dois níveis

Este protocolo, descrito em [35], controla a concorrência para as transações otimistas, levando em conta a semântica das operações de alto nível (nível 1). Em resumo, essa solução é a seguinte: a) a cópia particular do estado (representação) do objeto somente é obtida se a transação causa atualização, que desse momento em diante será considerada a versão do objeto, exclusivamente, para essa transação; b) a validação em dois níveis, controla duas classes de operações: b-1) no nível 1, as operações pertencentes a interface do tipo abstrato de dados, e b-2) no nível 2, as operações tradicionais, para referência/atribuição de atributos (*read/write*). Para as transações pessimistas, o controle da concorrência é resolvido pelo bloqueio das operações *read/write*, quer dizer, apenas, no nível 2, assim, começamos esta seção, discutindo a solução para as transações otimistas, porque estamos interessados em explorar seus mecanismos, introduzindo algumas mudanças, como podemos ver no parágrafo a seguir.

Consideramos que a solução, em (a) acima, pode ser empregada para as duas classes de política de transação. Assim, quando uma invocação de operação de escrita surge, uma cópia exclusiva do estado permanente do objeto é alocada, de acordo com a classe de política da transação que gerou a invocação, como a seguir: a) se pessimista, a cópia alocada servirá a todas as transações pessimistas que estiverem compartilhando o objeto, por causa da estratégia *UIP* adotada para essa classe de transação; e b) se otimista, como em [35], a cópia serve, exclusivamente, à transação que causou a invocação.

Na validação entre operações do nível abstrato, o novo protocolo, então, poderia comportar-se como foi descrito na seção dedicada à implementação encontrada em [1], consultando a tabela de conflitos, e depois a tabela de i*nterclass conflict resolution*. Devemos observar que essa validação é efetuada em momentos diferentes, dependendo da classe da política da transação, como a seguir: a) para as pessimistas de uma fase de bloqueio, dinamicamente, durante a chegada da mensagem e, b) para as otimistas, na etapa de consistência da fase de *commit*. Se, nesse nível, não tiver sido detectado qualquer conflito entre as operações, ainda, assim, alguma operação de *write* pode ter produzido algum efeito no estado do objeto, causando dependência entre transações. Essa chance de pendências justifica uma segunda validação, que funciona diferente, nas transações pessimistas e otimistas, que descrevemos, nessa ordem, a seguir.
Transações pessimistas Nas transações pessimistas, sempre que alguma operação de escrita causar uma pendência em alguma outra transação, essa pendência deveria ser identificada pelo protocolo. Dessa forma, ele deveria contar com uma lista, cujas entradas corresponderiam a todas as operações *read/write* das transações pessimistas ativas, que compartilhassem o objeto.

O protocolo, ao consultar essa lista, deveria proceder, como a seguir:
1) *em caso de comitting*:
a) *committing transaction* deveria esperar pelo *committment* de outras, quando fosse detectada uma pendência, quer dizer, essas outras tivessem causado escrita em atributos que a *committing transaction* tivesse lido depois;
b) no caso em que a *committing transaction* se encontrasse sem pendências, o protocolo deveria examinar se ela estaria causando pendência em outras, quer dizer, se essas outras causariam leitura de atributos que a *committing transaction* tivesse escrito antes, desligando essas pendências.
c) quando essas questões de pendências tivessem terminado, seria efetuada a penúltima tarefa, que seria tornar a cópia, considerada válida para as transações pessimistas ativas, como o novo estado permanente e as gravações da transação pessimista *committing*.
d) após essas gravações, se uma transação pessimista fosse considerada totalmente *committed*, deveria ser feita ainda a validação entre essa transação e cada uma transação otimista ativa. No caso de conflito nessa validação, a ação a ser tomada seria aquela da tabela de *interclass conflict resolution*, quer dizer, seria adotada a

estratégia *commit-and-kill*, que produziria o aborto da transação otimista.

e) quando uma transação fosse realmente *committed*, ela poderia liberar pendências em outras transações, podendo, por sua vez, chegar a condição de nenhuma pendência mais. Dessa forma, o protocolo deveria repetir os passos desde (1-a) enquanto houvesse transação a ser *committed* que não se encontrasse mais pendente.

2) *em caso de aborting*: a transação sendo abortada causaria também aborto das transações que possuíssem pendência em relação a ela, quer dizer, essas outras que tivessem tentado leitura em atributos que a transação sendo abortada tivesse escrito antes.

Se esse comportamento de validação no protocolo fosse construído sem limitar o número de transações pessimistas a compartilhar o ambiente de execução, poderiam surgir esperas infinitas pelos *committs* de transações que pudessem gerar pendências. Assim, para evitar esperas desse tipo, deveria ser introduzido um limite para o número de transações ativas pessimistas, a ser controlado por um gerente externo aos objetos e às transações, sendo essa restrição configurada no ambiente de execução. Esse limite é parecido com a solução de janela, em protocolos de rede. Em outras palavras, poderiam ser aceitos vários inícios de transações pessimistas em um gerente de objetos e de transações (*GOT*), até que, em determinado momento, a aceitação deveria ser fechada, as mensagens de transações novas então seriam enfileiradas, esperando pelo *commit* ou aborto de alguma transação ativa. Essa decisão ainda não conseguiria impedir alguma espera longa, sugerindo, nessas situações, um tratamento de aborto por *time-out*, ou uma programação especial para tratamento de transação longa. Em resumo, o protocolo agruparia as transações pessimistas ativas em um determinado número, e, no caso de atingir tempos de espera considerados longos, a transação que estivesse causando essa pendência deveria ser abortada, conseqüentemente, abortando as pendentes também.

Transações otimistas Nas transações otimistas, no momento de *commit*, após a validação no nível abstrato, conforme descrevemos, na seção acima, a validação no nível de estado somente deveria comparar o valor do atributo no estado permanente com o atributo lido pela operação *read*, nas duas situações seguintes: a) aquelas que fossem efetuadas sobre o atributo ainda no estado permanente; e b) aquelas que fossem efetuadas sobre o atributo já na versão alocada especialmente para a transação, mas antes que esse atributo tivesse sofrido qualquer alteração, devido a uma operação de *write* causada por essa transação.

Na fase *commit* de uma transação otimista, sua consistência verificaria os seguintes conflitos: 1) de primeiro nível entre as transações pessimistas ativas, como em [1]; e 2) de segundo nível entre as transações pessimistas ativas, como em nossa proposta acima, para as pessimistas. Caso fosse detectado algum conflito, a transação otimista seria abortada e seu estado exclusivo, se houvesse, deveria ser descartado, sem causar qualquer efeito. Após a consistência, a transação otimista sendo *committed* deveria providenciar as gravações em todos os objetos participantes e para cada objeto participante, seu monitor deveria atualizar o estado permanente e o estado compartilhado pelas pessimistas. Após a gravação com êxito, ainda deveria ser efetuada a consistência entre as transações otimistas ativas, na seguinte ordem: 1) de primeiro nível entre as transações otimistas, como em [20, 1]; e de 2) de segundo nível, pela comparação dos valores dos atributos lidos com os do estado permanente, como em [35]. Nessa segunda consistência, se fosse detectado algum conflito, a transação otimista ativa deveria ser abortada.

4. Características da nova implementação

Resolvemos tomar como base o protocolo apresentado em [20], porque pretendíamos um mecanismo de controle de concorrência que permitisse a negociação entre os diferentes protocolos e políticas de transações, que compartilhassem simultaneamente cada objeto atômico. Além disso, também estamos explorando oportunidades de flexibilidade, para permitir que novas políticas de transações sejam estendidas e tenham sua coexistência negociada entre as diferentes políticas nos mecanismos de controle de concorrência do monitor. Assim, para cada objeto de aplicação que fosse instanciado, deveria ser também instanciado e associado um monitor (meta objeto) de uma classe geral única, em vez das classes distintas para cada política diferente de transação, como em [35]. Essa classe deveria reunir os mecanismos de atomicidade, quer dizer, sincronização e recuperação, que soubessem lidar com todas as políticas e protocolos de transações conhecidos no ambiente de execução.

Por essa nova proposta, as mensagens poderiam ser trocadas no modo assíncrono e o controle de concorrência sobre as mensagens seria exercido por cada objeto, que viesse a ser alcançado pela propagação de uma mensagem, via composição ou referência por parâmetros, em outras palavras, a granularidade da concorrência seria sobre cada operação, em cada objeto, como é no protocolo apresentado em [20]. Essas duas questões, assincronismo e granularidade da concorrência em cada operação, causam uma certa dificuldade com os mecanismos de atomicidade, sincronização e recuperação, do protocolo de controle de concorrência proposto em [20], e, neste trabalho resolvemos estudar como adaptar essas duas questões às soluções encontradas nos protocolos propostos em [20, 18, 1 e 35].

Para a sincronização de mensagens, o protocolo proposto, em [20], conta com as informações de resultados das operações vinculadas a transações pessimistas, adicionando a mensagem que tenha sido retornada corretamente a *intention*'s lista da transação.

O assincronismo das mensagens, em nosso modelo, no entanto, impede que se conte com esses resultados, devendo a validação do critério de correção da sincronização ser resolvida antes da mensagem ser enviada. Assim, para a construção da tabela de conflitos para os objetos que seriam associados aos monitores desenvolvidos para a nossa proposta de protocolo, os seus elementos precisariam conter informações distintas para as transações pessimistas e otimistas. Para as pessimistas, as referências aos elementos dessas tabelas seriam restritas apenas ao nome da operação, enquanto que para as otimistas, o programador poderia usar o refinamento já adotado em [20], adicionando, ao nome da operação, mais duas espécies de informações: a) condições de execução da operação e b) seu próprio resultado, em termos absolutos ou em faixas de valores.

Ainda, em relação à questão da validação do critério de correção da concorrência, no protocolo apresentado em [1], as *particularidades* de uma classe de política de transação – otimista (*o*), pessimista (*p*), ou híbrida (*h*) – poderiam ser tratadas, *especialmente*, no servidor do objeto. Os conflitos entre transações de classes diferentes seriam resolvidos, em tempo de *commit*, pelo aborto da transação, respeitando a seguinte ordem de prioridade, em relação a sua classe: $o < h < p$. Nessa solução, é consultada uma tabela de conflitos entre as mensagens de transações de políticas diferentes, cujo elemento da tabela indica qual a ação que deverá ser efetuada: (*C&K*) *commit and kill*; ou (*S*) *suicidal*. Pela nossa solução, a classe híbrida não foi considerada, sendo desconhecida pelo nosso protocolo, por motivos de compatibilidade entre a serialização global [38, 15] e a serialização local.

Os possíveis conflitos entre as operações do nível abstrato[2], nesses protocolos anteriores, são apenas de uma natureza, indicando que as operações conflitantes podem gerar resultados errados, como por exemplo, dois correntistas de uma conta conjunta que tentam sacar simultaneamente. Um outro tipo de problema pode surgir em operações, que ao serem executadas, simultaneamente, não gerariam erro. Um exemplo assim é o caso de dois correntistas A e B que estivesse tentando depositar simultaneamente. O código básico para efetivar essa operação é "saldo = saldo + depósito". Esse comando de atribuição se desdobra em três operações de nível mais primitivo, como a seguir: i) a referência ao saldo, no lado da expressão; ii) 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. A referência ao saldo, a ser usada na expressão, para a transação TA associada ao correntista A e para a transação TB associada ao correntista B, poderia obter o mesmo valor original do saldo, porque ele poderia não ter sido atualizado. Em vista disso, a expressão de soma, que também seria efetuada para as duas transações, adicionaria os dois depósitos ao mesmo valor e, nas atribuições para as duas transações, o saldo resultante seria o da última atribuição efetuada, com o valor de uma das expressões, sem estar com os dois depósitos acumulados.

Para resolvermos esse tipo de problema, poderíamos considerar as contribuições encontradas em [9, 34, 8], que é uma outra abordagem *semantic-based concurrency control*, em que a informação de *dependency commit* é preenchida, pelo programador, na tabela de conflitos entre as operações do nível abstrato, para ser considerada a priori. Assim, surge um terceiro valor que poderia ser assumido pelo elemento da tabela de conflito, representando a dependência de *commit*, para a operação. Combinado a esse terceiro valor na tabela de conflitos, seria conveniente introduzir uma lista dos atributos do objeto que podem ser referenciados/atribuídos pelo método, podendo ter, então, suas operações *read/write locked*, em tempo de início da execução desse método (operação). Quando o protocolo fosse interceptar a mensagem correspondente a essa operação, associada a alguma transação pessimista de uma fase, e detectasse essa condição de dependência de *commit* por ocasião da consulta a tabela de conflitos, ele deveria providenciar os *lock*s das operações *read/write* para os atributos encontrados nessa lista associada a essa operação. No final da execução da operação, durante o seu retorno, o protocolo liberaria esses *lock*s. Dessa forma, caso duas transações invocassem operações desse tipo, os *lock*s de *read/write* assegurariam a serialização dessas operações, impedindo resultados errados, causados por referências efetuadas antes do momento apropriado.

Para a recuperação, as ações de re-execução, em casos de conflito/falha, ficariam complicadas com a granularidade da concorrência em cada operação de nível abstrato aplicada sobre cada objeto. Essa complicação nos leva a considerar que a ação de descartar a transação seria a mais apropriada, em uma condição de conflito/falha. Em [20, 1], a estratégia *deferred update* é adotada, utilizando-se de uma lista de intenções que é aplicada após o *commit* de cada transação. Dessa forma, a transação que estiver com algum conflito/falha detectado, necessitando ser abortada, terá sua lista de intenções descartada, não produzindo qualquer efeito no estado do objeto, quer dizer, não causando qualquer dependência nas outras transações.

Nossa proposta ficaria um pouco diferente da apresentada em [20, 1], adotando uma espécie de

[2] A expressão "operações do nível abstrato" significa as operações que aparecem na interface do objeto, quer dizer, as operações que caracterizam o seu tipo. Duas outras expressões: "operações de alto nível" e "operações do primeiro nível" têm o mesmo significado neste trabalho. Estas expressões são utilizadas para distinguir essas operações daquelas com atributos derivadas de referência e atribuição, chamadas por "operações read/write" ou "operações do segundo nível". Observamos que as mensagens a *self* podem usar as operações privadas, que, assim, também devem fazer parte deste elenco de operações de primeiro nível, pela nossa abordagem.

update in place strategy (*UIP*) sobre uma cópia do estado permanente, que seria compartilhada pelas transações pessimistas, enquanto que para as transações otimistas continuaríamos com a estratégia *deferrred update* (*DU*). Essa solução pode ser explicada, em detalhes, na seção que discute as modificações sobre o protocolo apresentado em [1], sendo que a principal justificativa para a adoção de *UIP*, em transações pessimistas, é a economia, ou até mesmo eliminação, de re-execuções. Isso simplificaria os mecanismos para o alcance da granularidade da concorrência em cada operação de cada objeto.

A abordagem, em [35], apresenta uma solução que simplifica essa tarefa de aplicação de lista de intenções adotada em [20, 1], sendo diferente para as transações pessimistas e otimistas, como podemos ver a seguir:

1) Para as transações pessimistas, com protocolos *one-phase locking* e *two-phase locking*, é adotada uma cópia compartilhada do estado permanente, e a validação do critério de correção da sincronização é exercida apenas entre as operações *read/write*. Durante o *commit*, como as transações pessimistas têm prioridade, essa cópia compartilhada passa, diretamente, a ser considerada o estado permanente.

2) Para as transações otimistas, o estado permanente do objeto é ser usado, até que a transação cause uma invocação de operação *write*, quando, então, uma cópia exclusiva desse estado permanente é obtida para essa transação. A consistência adota o *dual level validation method*: no primeiro nível, a validação da sincronização entre operações da interface do tipo abstrato de dados é a mesma encontrada em [20, 1]; no segundo nível, a validação da sincronização entre as operações *read/write* é simplesmente uma comparação entre os valores dos atributos que foram lidos do estado permanente original e os que se encontram agora no estado permanente atualizado. Se esses valores comparados coincidem, o estado é considerado válido e essa cópia torna-se o novo estado permanente, senão as operações são re-executadas.

Para nós, essa solução poderia ser estendida, adotando *dual level validation* também para as transações pessimistas, que utilizariam uma versão compartilhada, para que a estratégia *update in place* fosse adotada. A verificação de conflitos, para obter o *locking* de cada operação, deveria ser efetuada em momentos diferentes para a política *one-phase locking* e *two-phase locking*, como podemos ver, a seguir.

Para a política pessimista com protocolo *one-phase locking*, no momento do atendimento da mensagem, se a verificação de conflito de primeiro nível resultar na condição U, devem ser tentados os *locks* para as operações de segundo nível da lista do método, como já descrevemos acima. Essa solução deixa as operações de segundo nível livres por mais tempo, aumentando as chances de concorrência.

A princípio, podemos ficar inseguros, imaginando que intercalações de operações de primeiro nível com conflito Y possam compartilhar operações de segundo nível invocadas por intercalações com conflito N ou U. Nossa estratégia de testes para essa solução deverá obedecer a seguinte ordem: 1º) bloquear somente as operações de segundo nível que estiverem na lista da operação de primeiro nível e que obteve algum conflito de intercalação na condição U; caso não funcione, então, 2º) bloquear também as operações de segundo nível que estiverem na lista da operação de primeiro nível e que obteve algum conflito de intercalação na condição Y; caso não funcione, então, 3º) bloquear também as operações de segundo nível que estiverem na lista da operação de primeiro nível e que obteve no teste de conflito de intercalação a condição N. Podemos observar que com essa terceira opção conseguiríamos operações de segundo nível ainda livres por mais tempo, devido às suas liberações ao final da operação e não ao final da transação, como acontece na política a seguir.

Para a política pessimista com protocolo *two-phase locking*, no final da primeira fase, deve ser efetuada a consistência, entre as transações pessimistas ativas, de todas as operações de primeiro e segundo nível registradas, por ocasião de suas execuções sobre um estado temporário e descartável. Se essa consistência não detectar conflito, o bloqueio poderá ser efetuado para as operações registradas, quando então pode ser dado início a fase seguinte, a de execução propriamente dita.

Devemos notar que a antecipação dos *lock*s, nesta política pessimista com protocolo *two-phase locking*, mantém as chances de concorrência iguais as do protocolo encontrado em [35], porque não podemos utilizar as oportunidades de liberação dos *lock*s de segundo nível, como os feitos para a política pessimista com protocolo *one-phase locking*.

Uma outra questão que surge, nesta política pessimista com protocolo *two-phase locking*, é a possibilidade do estado não ter os mesmos valores, no momento da execução, resultando em caminhos lógicos diferentes, que podem invocar operações de primeiro e segundo nível diferentes, e a operação de primeiro nível não se encontrar bloqueada para a transação, no momento da chegada de sua invocação ao objeto.

Para contornar essa situação de operação invocada e ainda não bloqueada, considerada uma contradição para a política pessimista com protocolo *two-phase locking*, estamos vendo as seguintes alternativas. A primeira delas, mais econômica, seria adotar os procedimentos de bloqueio utilizados para a política pessimista com protocolo *one-phase locking*, no momento de atender a alguma mensagem assim. Se a operação estiver bloqueada por outra transação, uma das duas alternativas seguintes pode ser tomada: a) se a outra transação é pessimista com protocolo *two-phase locking*, então a transação em foco deve ficar a espera, pela liberação da operação; ou b) se a outra transação é pessimista com protocolo *one-phase locking*, então a

transação em foco obtém o bloqueio e a outra transação deveria ser abortada. A segunda delas, ainda econômica, seria bloquear todas as referências encontradas nos códigos, devendo, para isso, exercitar todos os caminhos possíveis de serem seguidos. Devemos notar que essa segunda medida talvez seja impossível, por causa da tipagem forte. Em terceiro lugar, a medida mais cara seria, ainda mantendo a estratégia *UIP*, estabelecer que o estado (representação) do objeto, compartilhado pelas pessimistas, devesse ser congelado durante a primeira fase e na tentativa dos bloqueios, obrigando a suspensão da execução das outras transações pessimistas, enquanto não começasse a fase de execução da transação em foco.

Podemos notar que com a estratégia *UIP* para as transações pessimistas temos mais chances de intercalar as operações, obedecendo apenas a verificação de conflitos, quer dizer, aumentando as chances de concorrência, separadamente da recuperação.

Para a recuperação, durante a fase de *commit* da transação, uma lista de operações *read/write* que foram surgindo na execução deve ser consultada, permitindo detectar a pendência de *commit* entre as transações pessimistas.

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 *committing*; 2) se ela depende de alguma outra transação ativa ou em fase de *committing*; 3) se ela depende de alguma outra transação ativa ou em fase de *committing*, e essa outra também depende dela; e 4) se alguma outra transação depende dela.

Após essa consulta, o protocolo pode tomar as seguintes ações, dependendo da condição retornada: condição (1) a transação pode ser gravada, imediatamente; condição (2) ela somente poderá ser gravada, após a gravação das transações das quais ela depende; e condição (3) todas as transações pendentes entre si deverão ser gravadas juntas; 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 sendo *committed*, que é decidida, após a consulta acima, garantindo que não há pendência de *commit*, 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. Essa ação corresponde a ação de concatenação da lista de intenções da transação sendo *committed*, em [20].

Se alguma transação pessimista, por questões de *deadlock*, ou de algum conflito insolúvel, como no caso acima de transação pessimista com protocolo *two-phase locking*, que na execução encontrou algum objeto com estado diferente, ou de falha, precisar ser abortada, a relação de pendência de *commit* também deve ser consultada como acima. Se a consulta retornar as condições (3) ou (4), que indicam que há pendência de transações em relação a que está sendo abortada, o protocolo deverá providenciar o aborto dessas também, evitando o *undo*.

Em resumo, notamos que a estratégia *UIP* para as transações pessimistas aumenta as possibilidades de intercalação de operações, que deve obedecer apenas a condição da verificação de conflitos, deixando as pendências de gravação, que forem surgindo durante a fase de execução, para serem resolvidas na fase de *committing*, cujo prejuízo é apenas o atraso nessa fase, devido a espera pelo *committing* da transação responsável pela pendência, sem qualquer necessidade de *redo* ou *undo*.

Nas transações otimistas, a concorrência e recuperação ficam resolvidas, na fase de *committing*, como podemos ver nos três parágrafos a seguir.

Em primeiro lugar seria processada a consistência entre as transações pessimistas ativas, verificando os seguintes conflitos: a) de primeiro nível, como em [1]; e b) de segundo nível, como em nossa proposta acima, para as pessimistas. Se fosse detectado algum conflito nessas verificações, a transação otimista seria abortada, e, então, caso tivesse havido uma cópia do estado permanente, ela seria descartada, sem causar qualquer efeito.

Em segundo lugar, se a consistência entre a transação otimista sendo *committed* e as pessimistas tivessem alcançado êxito, então, deveriam ser providenciadas as gravações para essa transação em todos os objetos participantes.

Em terceiro lugar, se essas gravações fossem bem sucedidas, então, deveria ser efetuada a consistência entre as transações otimistas ativas, como em [1], pelo esquema chamado de *forward-oriented concurrency control* (FOCC), que significa verificar os conflitos entre as operações da transação em foco e das transações ativas, uma de cada vez. Nessas consistências entre as otimistas, seriam verificados os seguintes conflitos: a) de primeiro nível, como em [1]; e b) de segundo nível, pela comparação dos valores dos atributos lidos com os do estado permanente, como em [35]. Caso fosse detectado conflito nessa consistência, a transação otimista sendo *committed* seria mantida e a transação otimista ativa seria abortada. As transações otimistas ativas que sobrevivessem à consistência, deveriam ter seus estados atualizados.

Na atualização do estado exclusivo de cada transação otimista ativa, os efeitos produzidos precisariam refletir uma ordem em que as operações da transação sendo *committed* tivessem sido realizadas antes de qualquer operação da transação otimista ativa. Assim, a atualização deveria ser feita pela aplicação das operações *write* (de segundo nível) com *timestamp* mais novo da transação otimista ativa sobre o estado

permanente, que antes deveria ter sido atualizado com as operações da transação otimista sendo *committed*. Após essas aplicações, esse estado seria considerado o estado em vigor para a transação otimista ativa. Observar que esse procedimento deveria ser feito para cada transação otimista ativa, e que o estado permanente atualizado com a transação otimista sendo *committed* não deveria ser alterado, sendo todas essas ações feitas em cópias de rascunho.

Para evitar que surjam esperas infinitas pelos *committs* de transações pessimistas que geram pendências, foi preciso limitar o número de transações ativas que compartilham um objeto. Essa restrição deve ser verificada toda a vez que for detectada a chegada da primeira mensagem pertencente a uma transação, quer dizer, o começo do compartilhamento do objeto pela transação[3]. Essa decisão ainda não elimina totalmente a chance de alguma espera longa pelo fim de alguma transação longa, sugerindo, nessa situação extrema, um tratamento especial, ou o tradicional aborto por *time-out*. Para esse controle de tempo e também de *deadlock*, contamos com a ajuda do gerente de objetos e de transações da meta arquitetura (*GOT*), da qual também faz parte todos os monitores.

Nas três sub seções da seção anterior, a primeira dedicada ao protocolo encontrado em [20], a segunda dedicada à implementação encontrada em [1] e a terceira dedicada à implementação encontrada em [35], na ordem cronológica, é possível ver em maiores detalhes as soluções comentadas aqui. Essas soluções foram modificações que fomos fazendo nas três contribuições para ajustar às necessidades adicionais, que precisa resolver o assincronismo nas mensagens, combinado à granularidade do controle de concorrência em cada operação de cada objeto.

5. Conclusões

O objetivo inicial neste estudo era o de investigar as influências, causadas pela adição da facilidade de troca de mensagem assíncrona, nos mecanismos de controle de concorrência e recuperação localizados nos objetos atômicos, implementados por uma ferramenta genérica de programação.

Essa ferramenta tem o compromisso de simplificar o emprego/extensão de *ATM*s e levar em conta os recursos de orientação a objeto [10], cujo modelo aproxima-se do encontrado, em [35]. Essa aproximação é identificada pelo emprego dos dois aspectos seguintes: a) programação reflexiva [27, 23] para permitir a programação dos modelos de transação, programados em classes, nos módulos de uma meta arquitetura; e b) a implementação dos mecanismos de controle de concorrência e recuperação, no meta objeto, sendo responsável por tornar o objeto da aplicação um objeto atômico.

Esses mecanismos de controle de concorrência e recuperação devem permitir a negociação entre as diferentes políticas de transações, que compartilhem simultaneamente cada objeto atômico, além de permitir que novas políticas de transações sejam estendidas. Para isso, foram levados em conta os aspectos importantes, a seguir: 1) a convivência de transações de políticas diferentes, compartilhando um mesmo objeto de aplicação, ver [17, 20, 38, 18, 1, 35]; 2) a serialização global, ver [38, 15] e a anomalia de herança ver [37, 22], podendo ser resolvidas pelo uso de reflexão na implementação do objeto atômico; e 3) a gerência da transação sendo programada em classe, prevista uma instância para cada transação que surgir no ambiente de execução.

Estamos considerando também que a propagação da mensagem pela composição ou pela passagem de parâmetro, delegando o controle de concorrência e recuperação aos objetos destinatários da mensagem propagada, o que torna a granularidade de concorrência no nível de cada operação em cada objeto. Esse nível de granularidade já é assumido em [20, 35], mas estudamos a sua combinação com a troca de mensagens assíncronas.

Como a adição dessa capacidade poderia influenciar o funcionamento do controle de concorrência e de recuperação, resolvemos estudar o protocolo apresentado em [20] e as contribuições trazidas pelas implementações, encontradas em [1], baseado em [20], e [35], cujo tratamento para as transações otimistas foi baseado em [18].

Um outro aspecto a considerar, pelo nosso modelo, é o fato de que cada objeto de aplicação que seja instanciado, deve ser também instanciado e associado um monitor (meta objeto) de uma classe geral única, em vez das classes distintas para cada política diferente de transação, como em [35]. Assim, essa classe deve reunir os mecanismos de atomicidade, quer dizer, controle de concorrência e recuperação, que saibam lidar com todas as políticas e protocolos de transações conhecidos no ambiente de execução.

Com os resultados desse estudo, vimos a possibilidade de adicionar a troca de mensagem assíncrona a essas implementações dirigidas a objetos atômicos. Essa possibilidade pode causar pequenos prejuízos no desempenho das transações pessimistas que compartilham tais objetos, ficando localizados no impedimento da participação dos resultados das operações, na verificação de conflitos, que deve ser efetuada na invocação (a priori) e não no seu retorno. No entanto, o estudo dessas implementações nos mostrou várias outras oportunidades, além da possibilidade de tratamento da troca de mensagem assíncrona para os objetos atômicos, como podemos ver, a seguir.

Para as transações pessimistas *one-phase locking*, surgem novas chances de concorrência, originadas do esquema de bloqueio temporário de operações de segundo nível, durante somente a execução de

[3] Esse limite é parecido com a solução de janela, em protocolos de rede.

operações de primeiro nível, cuja verificação de conflitos retorne a condição *U*. Para as verificações de segundo nível, é possível contar com o *timestamp* da operação, para verificar conflitos e resolver as pendências de *committing*. A adoção da estratégia *update in place* (*UIP*), para as transações pessimistas, também contribui com o desempenho, porque evita abortos na fase de gravação. Para as transações pessimistas *two-phase locking*, as chances de concorrência são praticamente mantidas, precisando examinar se a opção de efetuar a consistência, somente, no final da primeira fase, traz alguma vantagem. O esquema chamado de *forward-oriented concurrency control* (FOCC), que significa verificar os conflitos entre as operações da transação otimista sendo *committed* e das transações otimistas ativas, também precisa ser examinado, para saber se traz vantagens.

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] A. Biliris, K. Ramamritham, S. Dar, N. Gehani, and H. V. Jagadish. ASSET: A System for Supporting Extended Transactions. In *Proc. ACM SIGMOD Int"l Conf. Management of Data*, ACM Press, New York, N.Y., 1994, pp. 44-53.

[4] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, September 1998.

[5] A. P. Barros and A. H. M. ter Hofstede. Modelling Extensions for Concurrent Workflow Coordination. In *IEEE Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems*. 1998.

[6] Brian Bennett, Bill Hahm, Avraham Left, Thomas Mikalse, Kevin Rasmus, James Rayfield, and Isabelle Rouvellou. *Middleware 2000*, LNCS 1795. Springer –Verlag Heidelberg. 2000.

[7] Roger Barga and Calton Pu. A Practical and modular Method to Implement Extended Transaction Models. In *Proc. 21st International Conference on Very Large Data Bases (VLDB'95)*, Zurich, Switzerland, 1995.

[8] Roger Barga and Calton Pu. The Reflective Transaction Framework. Chapter Three in *Advanced Transaction Models and Architectures*, Editors: S, Jajodia & L. Kershberg, August 1997.

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

[10] Maria Alice Silveira de Brito. Uma Arquitetura Orientada a Objetos para Integração das Facilidades de Concorrência, Transação e Persistência . PhD Dissertation. Pontifícia Universidade Católica do Rio de Janeiro, PUC-Rio, March 1999.

[11] Laurent Daynès, M.P. Atkinson and Patrick Valduriez. Customizable Concurrency Control for Persistent Java. Chapter Seven in *Advanced Transaction Models and Architectures*, Editors: S, Jajodia & L. Kershberg, August 1997.

[12] David L. Detlefs, Maurice Herlihy, and Jeannette M. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*. December 1988.

[13] Ahmed K. Elmagarmid. Database Transaction Models for Advanced Applications. Morgam Kaufmann Publishers, Inc. 2929 Campus Drive, Suite 260, San Mateo, CA 94403. 1992.

[14] M. M. Gore and R. K. Ghosh. Recovery in Distributed Extended Long-lived Transaction Models. In *Proccedings of the 6th International Conference on Database Systems for Advanced Applications*. IEEE. 1998.

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

[16] Paul Grefen, Jochen Vonk, Erik Boertjes and Peter Apers. Semantics and Architecture of Global Transaction Support in Workflow Environments. In *Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems*. IEEE. 1998.

[17] 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.

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

[19] George Heineman and Gail E. Kaiser. The CORD approach to Extensible Concurrency Control. In *Procedings of the 13th International Conference on Data Engineering*. IEEE. 1997.

[20] 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.

[21] 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.

[22] Kafura and Lee 1989, ver [30, 4]

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

[24] Vasudev Krishnamoorthy and Ming-Chien Shan. Virtual Transaction Model to Support Workflow Applications. In *Proceedings of SAC'00* March. ACM. 2000.

[25] H. Lieberman. Concurrent object-oriented programming in Act 1. *Object-Oriented Concurrent Programming – MIT Press* – A. Yonezawa & M. Tokoro Ed. 1987.

[26] M. C. Little and S. K. Shrivastava. Distributed Transaction in Java. *Contribution to High Performance Transaction Systems (HPTS) workshop*, Monterey. 1997.

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

[28] Luigi Mancini, Indrajit Ray, Sushil Jajodia and Elisa Bertino. Flexible Commit Protocols for Advanced Transaction Processing. Chapter Four in *Advanced Transaction Models and Architectures*, Editors: S, Jajodia L. Kershberg, August 1997.

[29] 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.

[30] Michael Papathomas. *Concurrency in Object-Oriented Programming Languages. Object-Oriented Software Composition* - Prentice Hall - Oscar Niertrasz & Dennis Tsichritzis Editors. 1995.

[31] Grahan Parrington. Reliable Distributed programming in C++: The Arjuna Approach. In *Proceedings of USENIX/C++ Conf.*, San Francisco, April 1990.

[32] G.D.Parrington et al. The Design and Implementation of Arjuna. *USENX Computing Systems Journal*. 8 (3). 1995.

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

[34] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6). 1995.

[35] 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.

[36] Paul Taylor, Vinny Cahill and Michael Mock. Combining Object-Oriented Systems and Open Transaction Processing. *The Computer Journal*, 37(6), 1994.

[37] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of the ACM Conference on Object-Oriented Programming, System, and Applications (OOPSLA'87)*. 1987.

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

[39] Willian Weihl and Barbara Liskov. Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems*, 7(2):244-269, 1985.

[40] 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.

[41] Hiroshi Yamamoto, Yuntao Guo, Koichiro Suzuki and Toyohide Watanabe. In *Proceedings of the Third International Conference on Computational Intelligence and Multimedia Applications*. IEEE. 1998.