Semântica das Mensagens que Constituem as Interfaces dos Três Principais Elementos da Meta Arquitetura Alana

Maria Alice Brito malice@ime.uerj.br
Alfredo Menezes alfredomenezes@hotmail.com
Daniel Hess dfhess@ig.com.br
Érico F. de Almeida ericofa@terra.com.br
Maurício O.de Freitas olmo@terra.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 paper, we describe the Alana meta architecture, whose model encloses two major purposes: 1) to integrate the transaction feature to an object oriented programming language, and 2) to allow the extension and simplify the use of ATMs, fitting itself in the generic tools line. This meta architecture combines four paradigms, as follows: object oriented programming, concurrence, persistence and transaction. We believe that the presentation task of this meta architecture becomes difficult because this diversity of paradigms. Thus, to attempt make easier its explanation, we decided to describe how this meta architecture works by the interface semantic of their three major elements. For this, we classified their messages in three groups in accordance to function: 1) interaction messages; 2) persistence messages; and 3) global and local atomicity messages. For each group, we presented the behavior of these messages received by elements of meta architecture.

1. Introdução

No contexto de construções em linguagens de programação orientada a objeto, em particular, nos mecanismos que integram a essas linguagens as facilidades de sistemas, tais como concorrência, transação e persistência, especificamos um modelo de meta arquitetura [7].

Essa meta arquitetura integrada à linguagem permite que uma implementação de aplicação possa contar com tipos de transações, que variam numa ampla faixa, desde as mais simples até as enquadradas no contexto de longa duração e em alguns casos consistindo de componentes associados a transações, que podem finalizar individualmente (*workflow*), apresentando características cooperativas em vez das tradicionais características competitivas.

Nessa última faixa de tipos de transações, a complexidade das aplicações tem aumentado e para isso muitas ferramentas foram propostas e desenvolvidas, porém, duas principais necessidades ainda precisavam de solução: 1) facilidades de programação e de emprego de modelos de transações novos, incluindo o compartilhamento de transações com políticas diferentes sobre um mesmo objeto, e 2) a capacidade de interação entre os objetos da aplicação e as transações e entre as transações. Algumas ferramentas ofereciam parte dessas soluções mas continuavam apresentando dificuldades de uso e muitas vezes não se adaptavam aos paradigmas – de programação orientada a objeto, concorrência, persistência e transação. Assim, os modelos que tiveram mais chances, para o alcance da capacidade de programação esperada, foram aqueles que exploraram os aspectos de tipos abstratos de dados e de herança, encapsulando os mecanismos para transações, que foram integrados às aplicações. Nesses modelos, como por exemplo, os encontrados em [21, 13], os mecanismos foram programados com reflexão, localizados em uma meta arquitetura, e tiveram incorporados os conceitos de objeto ativo e de atomicidade local.

Nessa linha, também desenvolvemos nosso modelo de meta arquitetura, baseada em três classes principais, que especificam e implementam os seguintes elementos: 1) gerente de objetos e transações (*GOT*); 2) gerente de transação (*GT*), instanciado para cada transação que surgir no ambiente de execução; e 3) o meta objeto (*MO*), instanciado e associado a cada objeto de aplicação.

Um dos papéis que o *GOT* assume é o de armazém

dos objetos de aplicação, tendo o conhecimento da condição de vida desses objetos. Assim, no momento da liberação de cada objeto de aplicação, o *GOT* providencia o salvamento/remoção de seus controles no armazém (memória secundária), dependendo se esse objeto é persistente/transiente, respectivamente.

A principal oportunidade para facilitar a convivência de transações com políticas diferentes foi trazida pelo isolamento da sincronização de cooperação, nos mecanismos de controle de concorrência, que ficam localizados no meta objeto, podendo representar as negociações entre essas políticas.

A principal oportunidade para a interação entre transações e entre objetos e transações foi trazida pela permissão de extensão, tanto das funcionalidades quanto das interfaces, e dispensa a criação de novas primitivas nas linguagens de programação, conduzindo à desconfortável situação de ASSET [5], cujo código do compilador sofre alteração toda vez que precisa integrar um novo recurso de interface a cada nova necessidade surgida.

A interação entre objetos pode ser feita através de troca de mensagens assíncronas e aliada a essa oportunidade de assincronismo, na propagação da mensagem pela composição ou pela passagem de parâmetro, é delegado o controle de concorrência e recuperação aos objetos destinatários da mensagem propagada, e, assim, por diante. Essa delegação refina a granularidade do controle de concorrência e recuperação a cada operação de cada objeto, aumentando as chances de concorrência.

Nossa motivação para descrever o funcionamento dessa meta arquitetura, através da cooperação entre os seus principais elementos, surgiu durante a implementação da biblioteca de classes das mensagens de suas interfaces. Como nossa meta arquitetura envolve vários paradigmas: o de programação orientada a objeto, o de concorrência, o de persistência e o de transação, que foram empregados seguindo um compromisso de respeito entre si, sempre que tentamos explicar nosso modelo, encontramos dificuldades, acreditando que o principal motivo para isso seja a diversidade desses paradigmas envolvidos. No exercício da implementação dessas mensagens, em que foi necessário acompanhar seus ciclos percorridos, entre os elementos da meta arquitetura, percebemos uma oportunidade de simplificar o entendimento de nossa meta arquitetura e do seu funcionamento.

Assim, nosso objetivo neste trabalho é o de descrever como os três elementos da meta arquitetura cooperam no sentido de fazer funcionar as facilidades de concorrência, persistência e transação, ao tratarem as mensagens que lhes chegam. Essa descrição se torna uma contribuição aqueles que tiverem interesse em estender um modelo de transação avançado, adotando a meta arquitetura Alana.

Este trabalho é constituído de mais seis seções, como a seguir: na seção 2, apresentamos os trabalhos relacionados ao modelo de meta arquitetura Alana; na seção 3, apresentamos os aspectos gerais e necessários à implementação das mensagens; nas três seções seguintes: 4, 5 e 6, apresentamos a cooperação entre os três elementos da meta arquitetura para fazer funcionar as facilidades, pelos ciclos percorridos pelas mensagens. Essas três seções foram dividas em grupos distintos pelos seus papéis exercidos, como a seguir: seção 4 – interações; seção 5 – persistência; e seção 6 – atomicidade global e local. Finalmente, na seção 7 apresentamos as conclusões sobre este trabalho.

2. Fundamentos e trabalhos relacionados

A meta arquitetura Alana está relacionada com as ferramentas que deram ênfase a facilidades para o emprego/extensão de modelos de transação e/ou de integração de transações ao paradigma de orientação a objeto. Essas ferramentas se enquadram nas três linhas de abordagem, a seguir: 1) a das ferramentas que possibilitaram extensão de modelos de transação [5, 6, 10, 16], aproveitando parcialmente as oportunidades oferecidas pelos tipos abstratos de dados para alcançar maior concorrência; 2) a dos ambientes transacionais [24, 11, 19, 22], que permitiram explorar a semântica das operações dos objetos, através do conceito de atomicidade local, 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) a das abordagens encontradas, em [21, 13], que empregaram os dois aspectos seguintes: 3-a) programação reflexiva 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, situados no meta-objeto que deve ser associado a cada objeto de aplicação.

O modelo de Alana fica mais próximo da abordagem encontrada, em [21], em que a atomicidade local foi implementada 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. Antes de abordarmos essas soluções, apresentamos, brevemente, os conceitos de atomicidade local e de objeto ativo, nos parágrafos seguintes.

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, foi desenvolvida por Willian Weihl, em sua tese, em 1984, e publicada em [23]. 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 [23].

Objeto ativo O conceito de objeto ativo – inspirado em *Actor languages* [14, 3] – 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.

No modelo encontrado em [21], a separação da parte seqüencial do objeto de aplicação da parte que trata o controle de concorrência e recuperação, que é implementada em um meta objeto, conseguiu evitar a anomalia de herança, que pode surgir, nos sistemas da segunda linha acima, aqueles que utilizaram a atomicidade local pela primeira vez. Esse problema pode surgir porque o código do controle de concorrência e de recuperação é programado juntamente com a funcionalidade do objeto e, quando uma classe assim é herdada, esse código é trazido e poderá não fazer sentido na semântica dessa nova classe. Além disso, uma outra questão, alertada por William Weihl, em [23], a compatibilidade entre as atomicidades locais, foi resolvida, assegurando a serialização global, porque a amarração da atomicidade local é feita à transação que acessa o objeto em vez de ser feita às características do objeto.

A gerência de transação, em [21], é programada em classe, possibilitando a extensão de ATMs e simplificando o seu emprego pela aplicação. Nesse trabalho, um meta objeto pode ser instanciado e amarrado dinamicamente a um objeto da aplicação, e a escolha da classe desse meta objeto depende de qual modelo é a transação agregada a mensagem que está chegando ao objeto de aplicação, no momento. A classe de meta objeto que corresponde ao tratamento dos mecanismos de controle de concorrência para uma transação otimista explora a concorrência nas operações de alto nível entre as transações otimistas, como, em [12], além de verificar se o estado está consistente (verificação no segundo nível), enquanto a correspondente para uma transação pessimista, embora empregue, na validação, uma relação de conflitos entre as operações de alto nível, leva em conta as oportunidades dessas operações causarem ou não atualização no estado do objeto. A verificação de conflitos entre as transações de políticas diferentes, se limita a conflitos *read/write* detectados por alteração de estado.

3. Recursos gerais para a implementação da biblioteca de mensagens

Introduzimos nesta seção as características gerais de uma mensagem, o significado e a importância do seu contexto, a facilidade de programação oferecida pelas classes aninhadas (*inner classes in Java*), os três principais elementos da meta arquitetura: o *GOT*, o gerente de transação (*GT*), e o meta objeto associado ao objeto de aplicação, a classe *Elemento*, que é a classe ascendente das classes desses três elementos principais da meta arquitetura, com sua caixa de mensagens e o contexto de transação, sendo que este último assume papel relevante no controle do *thread* lógico (transação).

O tratamento das mensagens trocadas entre os elementos foi pensado, inicialmente, para ser efetuado totalmente por esses, porém, dada a variedade de tipos de mensagens que circulam no sistema, consideramos apropriado dividir esse trabalho com a própria mensagem, melhorando o nível de abstração. Assim, a mensagem foi projetada como um objeto, adquirindo a capacidade de tratar a si própria, assumindo formas diferentes nesse tratamento de acordo com o tipo do elemento que a recebeu.

Raiz da hierarquia de classes de mensagens Caso algum programador estiver implementando algum novo modelo de transação e necessitar de alguma mensagem diferente, ele deverá herdar alguma classe da hierarquia, cuja raiz é a classe *MsgGal*. Essa hierarquia de classes representa mais um ponto de *hot-spot* para a programação com nossa meta arquitetura.

Contexto de mensagem Essas mensagens podem ser processadas, remotamente, como uma *remote procedure call* (*RPC*), que não pode ter acesso ao contexto local, precisando carregar informações necessárias ao seu funcionamento. Assim, a maioria das mensagens especializadas possui um atributo de contexto, cujo tipo corresponde ao tipo da mensagem. Esses contextos trazem as informações particulares a suas mensagens, as quais são consultadas efetivamente nos métodos de tratamento de mensagem. Além disso, se tornam contextos no meta objeto, auxiliando suas tarefas, principalmente, a de controle de concorrência e transação e na cooperação entre os elementos.

Esses contextos são transmitidos com as mensagens para que possam ser encontrados no retorno ou usados pelos outros elementos. Antes que a mensagem seja

transmitida para o destinatário, o emissor providencia uma transformação no contexto da mensagem que denominamos de contexto externo, em que os atributos que possuem informações sem interesse para o elemento destinatário são preenchidos com valor *NULL*. Essa medida revela uma intenção de compromisso com a proteção dos elementos da meta arquitetura, restringindo as informações que as mensagens devem transportar, para os elementos destinatários.

Métodos nas classes de mensagens As classes de mensagem possuem métodos de duas espécies, uma espécie que trata mensagem com prefixo *TrataMensagemNo*:

- *TrataMensagemNoMO*,
- *TrataMensagemNoGT* e
- *TrataMensagemNoGOT*

e outra espécie que retrata mensagem com prefixo *RetrataMensagemNo*:

- *RetrataMensagemNoMO*,
- *RetrataMensagemNoGT* e
- *RetrataMensagemNoGOT*.

Na classe raiz, *GalMsg*, esses métodos são declarados como virtuais e seus atributos são *id_EmissorMS* (objeto seqüencial que invocou alguma mensagem seqüencial que desdobrou nessa mensagem), *original_* (elemento emissor dessa mensagem) , e *c_Trans* (o contexto de transação que se encontra agregado a essa mensagem).

Um elemento emissor envia uma mensagem a um elemento destinatário, empacotando-a e depositando esse pacote na caixa de mensagens do destinatário. No momento em que esse pacote é retirado da fila de mensagens da caixa, a mensagem é extraída do pacote, e é enfileirada em uma outra fila chamada *q_Ms*, pertencente ao elemento destinatário. O método *MsgHandle*, comum aos elementos, pode se servir da fila *q_Ms*, cujas mensagens não foram ainda tratadas. As mensagens já tratadas, mas que por motivos de sincronização e/ou serialização não conseguiram ser respondidas, precisaram ser reinfileiradas, para serem retratadas em algum momento mais tarde, justificando a especificação dos métodos que tratam mensagens já tratadas mas não respondidas, identificados pelo prefixo *RetrataMensagemNo*. Podemos notar pelo sufixo dos nomes desses métodos que o tratamento é diferente, dependendo de qual elemento esteja recebendo a mensagem. Em outras palavras, as principais utilidades dos métodos de prefixo *TrataMensagemNo* são as de criar ou atualizar os contextos associados e depois passar as mensagens para o elemento destinatário. Enquanto que a única utilidade dos métodos de prefixo *RetrataMensagemNo* é a de passar a mensagem para o elemento destinatário.

Nem todos os tipos de mensagens são submetidas ao controle de concorrência (sincronização e à serialiazação) e, assim, a maioria dessas classes não redefine métodos identificados pelo prefixo *RetrataMensagemNo*. É importante notar que apenas a mensagem *MsgMr* será submetida ao controle de concorrência, no protocolo do meta objeto, assim sendo, sua classe foi a única a redefinir esse tipo de método, o *RetrataMensagemNoMO*.

Nem todos os tipos de mensagens são enviadas a todos os tipos de elementos, e, assim, algumas dessas classes não redefinem, exatamente, todos os três métodos identificados pelo prefixo *TrataMensagemNo*, bem como, exatamente, todos os três métodos identificados pelo prefixo *RetrataMensagemNo*.

Artifícios para a interface dos elementos Os elementos da meta arquitetura possuem os métodos correspondentes às mensagens que estamos apresentando neste trabalho, e, quando a mensagem é tratada no elemento, o método correspondente é invocado, fazendo parecer que as mensagens é que formam a interface do elemento.

Em maiores detalhes, quando uma mensagem é recebida pelo elemento em sua caixa de mensagens, ela é, inicialmente, tratada por um dos métodos de prefixo *TrataMensagemNo* da própria mensagem. A última ação nesses métodos é um simples envio ao elemento destinatário de uma mensagem (instrução da linguagem de programação orientada a objeto). A execução dessa instrução de envio invoca o método do elemento que implementa a semântica da mensagem.

Classe aninhada O elemento gerente de transação (*GT*) possui uma máquina de estado, cujos argumentos na função de transição: (entrada, estado seguinte), representam, respectivamente, (mensagem, fase da transação). Como as mensagens podem chegar em mais de uma fase da transação, de acordo com a fase, o tratamento de uma mensagem pode ser diferente, assim, a implementação desses métodos precisou recorrer ao artifício de classes aninhadas.

Esse artifício permite que os estados sejam programados por classes aninhadas e cada classe dessas pode ter métodos redefinidos da classe da instância em questão. Quando uma fase da transação deve ser trocada (estado seguinte), é efetuada uma coerção no estado com a nova classe aninhada que o representa, em tempo de execução, permitindo trocar a classe aninhada que passa a vigorar. Essa facilidade explora o polimorfismo, porque os métodos redefinidos na classe aninhada que está vigorando serão os alcançados.

Assincronismo nas mensagens de interação Os contextos de mensagem recebida *MsgMr* e de mensagem enviada *MsgMe* cooperam nos mecanismos que tratam o *envio/retorno* em nossa meta arquitetura, permitindo que o assincronismo se mantenha em chamadas com modo assíncrono, sem precisar recorrer às saídas adotadas ou no modelo OMG, que admite apenas invocações síncronas ou *deferred synchronous*, ou a de [CLS 96], em que os autores substituem dinamicamente todas as invocações assíncronas por

deferred synchronous. Como as mensagens do tipo *MsgMr* ou *MsgMe* geram contextos que aguardam, respectivamente, *MsgRr* ou *MsgRe*, em algum momento, as execuções, mesmo com propagações, ficam livres para retornarem quando for conveniente aos objetos da aplicação.

Nossa intenção é deixar os objetos da aplicação com o máximo de liberdade, e, se houver necessidade de algum controle sobre o tempo de retorno – que concretamente fica refletido no tempo da transação, e que é detectado pela necessidade de uma outra transação – então são tomadas iniciativas, como, por exemplo, abortar por *time-out* a transação causadora de algum engarrafamento. Nesse caso, é providenciada a remoção dos contextos de mensagens associadas à transação abortada, em todos os meta objetos de seus participantes.

Elementos da meta arquitetura Alana A meta arquitetura Alana consiste basicamente de classes modelando três elementos: o meta objeto de cada *objeto de aplicação*, o *gerente de transação* e o *gerente de objetos e transações* (*GOT*), que podem cooperar entre si, através de trocas de mensagens.

A divisão de atribuições entre esses três elementos foi conduzida pela intenção de alcançar maior autonomia para o elemento *MO* do objeto de aplicação, economizando ao máximo os recursos externos. Assim, os dois outros elementos, *GT* e *GOT*, receberam somente as atribuições que foram surgindo a medida que as soluções para a integração das facilidades não podiam manter-se localizadas no meta objeto.

Classe Elemento e caixa de mensagens Fazemos notar que todos os três elementos, o gerente de objetos, a transação e o meta objeto herdam uma classe *Elemento* e, além disso, a comunicação entre os três elementos é feita por trocas de mensagens que são depositadas em suas caixas de mensagens.

A classe *Elemento* fatora propriedades das classes *MetaObjeto*, *GerenteDeTransação* e de *GerenteDeObjetosEDeTransações*.

A classe *CxMsg* é a caixa de mensagens que recebe qualquer mensagem de qualquer um outro elemento, sendo o elo entre esses elementos, uma transmissão de mensagem de um elemento para outro significa enviar uma mensagem *escrever* com o argumento *Pacote*. Quando esse método *escrever* é invocado na caixa de mensagem do destinatário, ele enfileira o pacote na fila de mensagens, e avisa ao elemento associado que pode acordar, porque chegou mensagem. Ao mesmo tempo, o elemento destinatário associado à caixa pode lhe pedir uma mensagem, para ser tratada por ele. Para isso, o elemento invoca o método *ler* da caixa, que ao ser executado, remove um pacote da fila e retorna esse pacote como argumento. Quando o pacote é retornado da caixa de mensagens, ele é desempacotado e a mensagem é extraída, via método **`getMensagem ()`**, que é uma instância de uma das classes de mensagens,

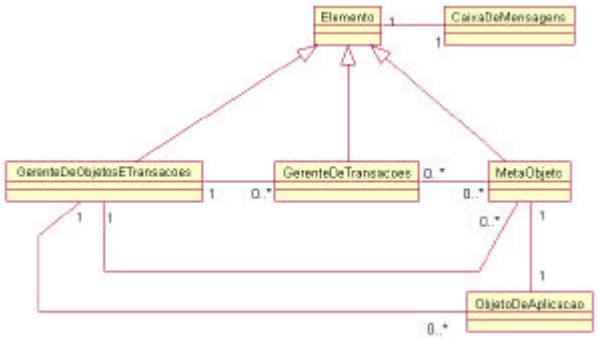

Figura 1: Visão estática da meta arquitetura.

cuja raiz é a classe *MsgGal*.

Em todos os elementos, o método *RecebePacote* desempacota o pacote, extraindo uma mensagem que é uma instância de uma classe que se encontra em uma hierarquia de classes de mensagens, cuja raiz é a classe *MsgGal*. Nós resolvemos empacotar as mensagens por causa dos endereços de origem e destino que não deveriam ser atributos da mensagem.

Gerente de objetos e de transações Uma aplicação integrada à meta arquitetura Alana pode envolver várias transações e objetos de aplicação que precisam ser administrados por um outro elemento que sirva para as necessidades, a seguir. Os objetos de aplicação precisam ser criados, ativados, receber mensagens, armazenados, desativados e distribuídos, configurando-se como tarefas que colaboram nas facilidades de persistência e distribuição para o elemento objeto de aplicação. As transações precisam ser criadas, receber mensagens e distribuídas, configurando-se como tarefas que colaboram na facilidade de transação no nível global e na facilidade de distribuição para o elemento gerente de transação. Todas essas tarefas de administração mais centralizadas, que não podem ser resolvidas por cada objeto e nem por cada gerente de transação, couberam ao elemento gerente de objetos e de transações de nossa arquitetura.

Gerente de transação A facilidade de transação pode envolver, em geral, vários objetos de aplicação, precisando de um elemento especial para administrá-la nesses objetos participantes e controlar as ações de finalização nesses objetos participantes. Atribuímos esse papel de gerência da transação ao elemento gerente de transação, que dessa forma colabora na facilidade de transação, garantindo a atomicidade global. A gerência de uma transação conta com uma *lista de participantes*, cuja entrada identifica o objeto de aplicação *participante*. O elemento gerente de transação deverá ser criado como uma instância da classe gerente de transação pelo gerente de objetos a pedido de um objeto de aplicação cliente. Como transações existem, somente, em tempo de execução, elas não são persistentes, porém podem ser distribuídas, porque um objeto pode precisar enviar-lhe mensagem, ou mesmo criá-la, encontrando-se em outro

ambiente, administrada por outro *GOT*.

Meta objeto Para resolvermos a parte do controle de concorrência relacionada ao objeto de aplicação, desenvolvemos um meta objeto, que, em tempo de ativação, é associado ao objeto seqüencial. Em seguida adicionamos as propriedades de atomicidade local [23, 12, 21] às tarefas de concorrência nesse meta objeto.

Os mecanismos, no meta-objeto, que resolvem as facilidades de concorrência e de atomicidade local, contam com objetos, principalmente, das classes para *contexto de transação cliente* e *lista de transações clientes*. Para cada transação que concorre sobre um objeto de aplicação, o seu meta objeto a identifica por um *contexto de transação cliente*. Esse contexto deve ser associado a uma *versão do objeto seqüencial* e ele é uma entrada na classe *lista de transações clientes*, cuja instância serve para o meta objeto controlar as suas transações concorrentes.

O meta objeto também possui mecanismos de controle exercido sobre as transações iniciadas pelo seu objeto de aplicação, que possuem um *contexto da transação local* que deve ser incluído como entrada na *lista de transações locais*. Esses contextos participam da facilidade de transação no nível global. Um objeto de aplicação é criado pelo gerente de objetos e de transações, a pedido de um outro objeto de aplicação.

Os três elementos – gerente de objetos e de transações, gerente de transação e meta objeto associado ao objeto de aplicação – são responsáveis pelas tarefas necessárias às facilidades de concorrência, transação e de persistência, que podemos abreviar por facilidade de transação. A parte seqüencial do objeto da aplicação, quer dizer, a classe que foi programada para esse objeto, fica responsável principalmente pela parte semântica da aplicação, participando sutilmente no controle de concorrência nas transações, particularmente, no momento de validar uma invocação ao objeto de aplicação.

Embora em um sistema que empregue o nosso modelo de arquitetura convivam objetos seqüenciais e objetos com características *transacionais*, sendo ambos considerados objetos de aplicação, durante a execução de uma aplicação em um ambiente de execução, em que essa meta arquitetura se encontra integrada, todos os objetos de aplicação são considerados como se tivessem as características transacionais, para que o *thread* lógico não seja perdido. No momento de persistir definitivamente é que há uma diferença, enquanto que o objeto persistente permanece armazenado na memória secundária, o objeto transiente é removido.

Em resumo, o gerente de objetos e de transações é responsável pelas transmissões de mensagens, inclusive para gerentes externos (facilidade de distribuição), facilidade de persistência dos objetos de aplicação, e pela criação e liberação das transações; a gerência da transação controla a atomicidade global com a colaboração de seus objetos participantes; e o meta objeto do objeto de aplicação fica praticamente responsável pela concorrência e pela atomicidade local.

Contexto de transação É importante observarmos que a transação é a nossa unidade principal de concorrência, que identifica cada *thread lógico* que ocorre nesse ambiente de execução. Para que esse *thread lógico* pudesse exercer esse papel, criamos o contexto de transação para ser agregado às mensagens, que fica registrado como um rastro no elemento destinatário da mensagem. Esse contexto de transação deverá ser conhecido, no tratamento de mensagem do lado do objeto seqüencial também. Cada mensagem possui dois atributos do tipo contexto de transação, um externo CTGal e outro interno CTrans, fazendo com que qualquer mensagem que circule pelo ambiente de execução esteja garantidamente vinculada a uma transação, que dessa forma contribui para o rastreamento do *thread* lógico.

No começo de uma aplicação, a primeira instância da classe *CTGal* (classe de contexto de transação geral) a surgir deve ser criada pelo gerente de objetos e transações (*GOT*), ao selecionar (ativar) o objeto *principal* da aplicação. Esse contexto de transação não se encontra associado a qualquer transação e, assim, seu atributo *id_Transacao* tem valor **NULL**.

Esse contexto é agregado a primeira mensagem *MsgMr* transmitida pelo *GOT*, que deve chegar no objeto *principal* da aplicação. Essa situação de objeto *principal* (*top level*) é especial e ocorre em apenas um objeto na aplicação e todos os outros objetos da aplicação se encontram no papel de servidor e cliente muitas vezes. Essa mensagem é tratada no meta objeto, que invoca o método do objeto de aplicação, que por sua vez poderá desdobrar-se em uma ou mais mensagens enviadas, que antes de ser enviadas aos seus objetos destinatários, são interceptadas pelo meta objeto do objeto emissor. Quando o meta objeto trata essas mensagens enviadas e descobre que o atributo *id_Transacao* está com valor **NULL**, ele providencia a criação de uma transação otimista implícita. A partir dessa mensagem, todas as mensagens circularão com um contexto de transação preenchido.

4. Mensagens usadas na interação entre objetos da aplicação

As mensagens consideradas mais freqüentes e importantes são as que tratam as interações entre os objetos da aplicação e deverão ser instâncias da classe *MsgMe* (mensagem enviada pelo objeto seqüencial a outro objeto seqüencial), *MsgMr* (mensagem recebida por um objeto seqüencial de outro objeto seqüencial), *MsgRe* (retorno da mensagem enviada), *MsgRr* (retorno da mensagem recebida). Durante um envio de mensagem de um objeto de aplicação para outro objeto, ela se desdobra nessas quatro mensagens

intermediárias, que podem passar pelos três elementos da meta arquitetura, devendo cada um destes colaborar dando um tratamento apropriado. Cada interação que surgir entre dois objetos de aplicação, quer dizer, um envio de mensagem de um objeto para outro, um ciclo efetuado com essas quatro mensagens intermediárias deverá ser desenvolvido.

Ilustramos esse ciclo, na figura 2, sendo desdobrado nas quatro meta mensagens *MsgMe*, *MsgMr*, *MsgRe* e *MsgRr*, mostrando o caminho que elas percorrem pelos elementos da meta arquitetura. Cada elemento da meta arquitetura, ao descobrir uma mensagem dessas na sua caixa, vai cooperar de uma forma específica para que esse ciclo seja concluído e para que sejam registradas informações úteis a serem utilizadas nas fases apropriadas de uma transação.

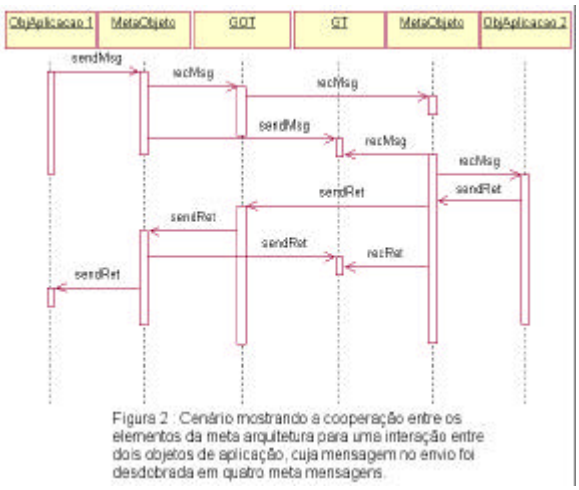

Figura 2 : Cenário mostrando a cooperação entre os elementos da meta arquitetura para uma interação entre dois objetos de aplicação, cuja mensagem no envio foi desdobrada em quatro meta mensagens.

A seguir descrevemos a cooperação entre os elementos para o desenvolvimento desse ciclo de interação entre objetos da aplicação:

a) No papel de cliente (emissor), ainda, no mecanismo de tratamento de mensagem do código gerado do método do objeto seqüencial, a mensagem de aplicação (*ms*) é transformada em uma mensagem enviada (*MsgMe*), sendo considerada uma das primeiras etapas de reflexão, em que as informações que caracterizam a mensagem, como nome do método, lista de parâmetros com seus tipos, e o retorno da mensagem são *reificadas* [15] e guardadas no contexto da mensagem enviada.

b) Quando a *MsgMe* fica pronta, esse tratamento de mensagem do código do objeto seqüencial providencia sua colocação na caixa de mensagens do meta objeto associado a esse objeto de aplicação emissor.

c) Quando o meta objeto associado ao objeto de aplicação emissor apanha essa mensagem de sua caixa, a primeira providência a ser tomada por ele é a de guardar o contexto dessa mensagem em uma lista das mensagens enviadas, para que posteriormente o retorno enviado pelo servidor dessa mensagem possa ser casado a essa mensagem enviada, indicando que o ciclo da interação está sendo concluído.

d) A seguinte providência tomada pelo meta objeto associado ao objeto de aplicação emissor é a de enviar essa mesma mensagem enviada *MsgMe* ao gerente de transação, que é identificado pelo atributo de contexto de transação, encontrado no contexto de mensagem enviada.

e) Quando o gerente de transação apanha essa mensagem *MsgMe* de sua caixa, ele verifica se esse objeto emissor já foi registrado como participante da transação, devendo registrá-lo, caso ainda não tenha sido. Em seguida, ele guarda o contexto dessa mensagem enviada para depois poder casar com o retorno dessa mesma mensagem. O gerente precisa controlar esse casamento, para detectar a verdadeira finalização de certas fases de uma transação, como, por exemplo, a primeira fase de uma transação pessimista *two-phase locking*, a fase de execução de todos os tipos de transação: otimista, ou pessimista *one-phase locking*, ou pessimista *two-phase locking*, além de, futuramente, ajudar em falha de nó de rede.

f) A seguinte providência tomada pelo meta objeto associado ao objeto de aplicação emissor é a criação da mensagem *MsgMr* a partir de informações obtidas dessa mensagem *MsgMe*, que estando pronta deve ser transmitida ao meta objeto do objeto de aplicação destinatário (no papel de servidor). Devemos notar que, para essa transmissão, tanto o objeto de aplicação destinatário quanto o seu meta objeto associado podem ainda não ter sido ativados (não encontrarem-se selecionados na memória *heap*). Essa é a situação mais comum, nesse ambiente de execução, em que cada objeto é selecionado somente quando uma instrução que o referenciar for executada. Nessa situação, o meta objeto do objeto cliente deve solicitar ao gerente de objetos e transação (*GOT*) que transmita essa mensagem ao destinatário. Assim, o meta objeto envia a mensagem *MsgMr* ao *GOT*, quer dizer, coloca a mensagem *MsgMr* na caixa de mensagens do *GOT*.

g) Quando o *GOT* apanha essa mensagem *MsgMr*, ele verifica se o par (meta objeto, objeto seqüencial) encontram-se ou não selecionados, para, em caso negativo, providenciar essa seleção. Com a garantia de que esse par encontra-se selecionado, o *GOT* coloca a mensagem *MsgMr*, na caixa de mensagens desse meta objeto destinatário.

h) Quando o meta objeto, no papel de servidor (destinatário), apanha a mensagem *MsgMr*, ele a coloca em uma lista de *MsgMr*, para ficar aguardando o retorno.

i) Depois, essa mensagem *MsgMr* deve ser submetida ao controle de concorrência, efetuado pelo protocolo, situado no meta objeto do objeto de aplicação destinatário. Esse protocolo vai agir de acordo com o tipo da política da transação agregada na mensagem, como, a seguir: i-i) se a transação é otimista, a

mensagem é despachada sem verificação de conflito, que é feito na fase de consistência; i-ii) se a transação é pessimista *2PL*, e o estado da transação agregada está na **`TPPrimFase`**, a mensagem é transformada em mensagem de bloqueio, para ser propagada aos objetos da composição atingidos pelo caminho lógico. É possível duas transações pessimistas compartilharem o mesmo objeto se elas não efetuam operações conflitantes. Quando há conflito, a transação menos prioritária é abortada e toda as tentativas de bloqueio, para essa transação abortada, são novamente efetuadas; i-iii) se a transação é pessimista *2PL*, e o estado da transação agregada está na **`TPSegundaFase`**, a mensagem é despachada para a versão compartilhada pelas transações pessimistas.

É apropriado chamarmos a atenção para esse item (i), porque ele é o item que garante um dos aspectos da atomicidade local, que é o controle de concorrência no objeto, lembrando que atomicidade conjuga controle de concorrência com recuperação. Essa garantia da atomicidade local em cada objeto participante de uma transação garante a atomicidade global da transação.

j) Tomadas as devidas providências no sentido do controle da concorrência e da recuperação, se não surgir qualquer conflito, o protocolo transforma, por reflexão, a *MsgMr* na *ms* original, com as informações guardadas na caixa de contexto de *MsgMr*, para então, verdadeiramente, despachar para a versão, invocando o método correspondente do objeto seqüencial destinatário;

k) Neste momento, o meta objeto do objeto de aplicação destinatário também deve colocar essa mesma mensagem *MsgMr* na caixa de mensagem do gerente de transação, identificado pelo contexto de transação, que está guardado no contexto da mensagem recebida.

l) Quando o gerente de transação apanha essa mensagem *MsgMr* de sua caixa de mensagem, ele deve proceder da mesma forma que com a *MsgMe* em (e), acima.

m) Quando o método for retornar (*rs*), o código do método do objeto seqüencial destinatário transforma esse retorno em um retorno de mensagem recebida (*MsgRr*) e providencia a sua colocação na caixa de mensagem de seu meta objeto.

n) Quando o meta objeto do objeto de aplicação destinatário, apanha esse retorno de mensagem recebida (*MsgRr*), ele providencia a sua colocação na caixa de mensagens do gerente de transação.

o) Quando o gerente de transação apanha esse retorno de mensagem recebida (*MsgRr*), ele guarda seu contexto nas estruturas que ajudam a fazer o controle do casamento, e se conseguir casar com a mensagem *MsgMr* correspondente, registra esse casamento também, que vai ser útil em certas fases da transação, como já foi observado em (e).

p) Ainda, no meta objeto do objeto de aplicação destinatário, esse retorno de mensagem recebida (*MsgRr*) deve ser casado com a mensagem *MsgMr*, significando incluí-lo na mesma lista em que foi guardada a mensagem *MsgMr*.

q) No meta objeto do objeto de aplicação destinatário, ainda, um retorno de mensagem enviada (*MsgRe*) é criado com informações obtidas desse retorno de mensagem recebida (*MsgRr*), para então ser enviado ao meta objeto do objeto de aplicação cliente (emissor), identificado por informações contidas no contexto do retorno de mensagem enviada.

r) Quando o meta objeto do objeto de aplicação cliente (emissor) apanha esse retorno de mensagem enviada (*MsgRe*) da sua caixa de mensagens, em primeiro lugar, ele envia esse retorno de mensagem enviada (*MsgRe*) ao gerente de transação, identificado pelo contexto de transação agregado ao contexto do retorno de mensagem enviada, da mesma forma como em (d), (k) e (n).

s) Quando o gerente de transação apanha esse retorno de mensagem enviada (*MsgRe*) da sua caixa de mensagens, da mesma forma que com o retorno de mensagem recebida (*MsgRr*), em (o), ele guarda esse retorno e se encontrar a *MsgMe* correspondente, registra o casamento também.

Ainda, no meta objeto do objeto de aplicação cliente, por reflexão, o retorno de mensagem enviada (*MsgRe*) é transformado no retorno da mensagem do objeto seqüencial *rs*, com as informações obtidas da caixa de contexto de *MsgRe*, para então, verdadeiramente, retornar ao objeto seqüencial do lado do cliente, terminando o ciclo de uma interação.

Considerações Na descrição, acima, do ciclo desenvolvido, durante uma interação entre dois objetos, neste ambiente de execução que estamos abordando, achamos conveniente ressaltar as cooperações mais importantes que surgem entre os elementos da meta arquitetura, a seguir: 1) o tratamento para tornar o modo assíncrono possível; 2) as providências para o controle de concorrência e recuperação; e 3) as providências para a persistência dos objetos de aplicação.

Aspectos da troca de mensagem assíncrona durante a interação O principal instrumento para prover o assincronismo é a caixa de mensagens, que vai registrar cada mensagem recebida que é destinada ao seu elemento associado, o qual solicita sua retirada em momento considerado apropriado para ele. Esse comportamento permite o assincronismo para o cliente e ao mesmo tempo colabora na solução da exclusão mútua (sincronização de competição) do objeto destinatário.

Na troca de mensagens assíncronas entre os objetos da aplicação, uma de suas maiores complicações é a descoberta, na lista de mensagens de envio (*MsgMe*'s)

que aguardam o retorno, qual delas corresponde a mensagem de retorno recentemente chegada e por isso devem ser casadas. Para tornarmos isso possível, incluímos, na especificação dessas mensagens, um atributo que guarda um contexto particular a cada tipo. Em outras palavras, foram criados os quatro contextos, a seguir:

1) um contexto de mensagem enviada que é atributo de *MsgMe*,
2) um contexto de retorno da mensagem enviada que é atributo de *MsgRe*,
3) um contexto de mensagem recebida que é atributo de *MsgMr*, e
4) um contexto de retorno da recebida que é atributo de *MsgRr*.

Esses contextos possuem, entre outras informações, a identificação necessária, para que os retornos possam ser combinados aos envios. Vimos, pelos itens (c), (d), (f), (h), (k), (l), (n), (o), (p), (r), e (s), acima, o empenho dos meta objetos e do gerente de transação, para que esse casamento fique resolvido, ajudando tanto a garantir que o retorno combine com a sua mensagem original quanto o gerente de transação detecte o final de cada ciclo de interação entre os objetos que participam da transação administrada por ele. Isso justifica a redundância desse casamento, que é efetuado tanto nos meta objetos quanto no gerente de transação. Com os nossos mecanismos, não foi preciso recorrermos às soluções adotadas ou no modelo OMG, que admite apenas invocações síncronas ou *deferred synchronous*, ou a de [9], em que os autores substituem dinamicamente todas as invocações assíncronas por *deferred synchronous*.

Aspectos do controle de concorrência e recuperação durante a interação No aspecto do controle de concorrência e de recuperação, podemos observar, pelo item (i), que somente a *MsgMr* é submetida a tal controle, pelo protocolo do meta objeto do objeto destinatário. Essa exclusividade para a *MsgMr* é devido a essa mensagem ser recebida pelo objeto destinatário, devendo então seu meta objeto verificar se a intercalação dessa mensagem com as outras que já chegaram e as que ainda estão por chegar não afeta a atomicidade local. Não é nossa intenção, no escopo deste trabalho, descrever esses mecanismos de controle de concorrência e recuperação, que pertencem ao protocolo localizado no meta objeto e que se encontram especificados em [8]. Pretendemos apenas adiantar que esse protocolo gera uma troca intensa de mensagens com o gerente de transação e com o gerente de objetos e transações *GOT*, para poder resolver a atomicidade local e colaborar na atomicidade global, a ser verificada pelo gerente de transação.

O controle de concorrência e recuperação também é exercido sobre as mensagens propagadas pela composição dos objetos de aplicação, via o contexto de transação, agregado a mensagem [18] e com uma solução parecida com a adotada em Galileu [2], passando pelo mesmo processo em todos os meta objetos que são associados a todos os objetos de aplicação. A oportunidade de propagação do contexto de transação, obtida pela agregação desse contexto à mensagem, facilitou o controle de concorrência sobre as mensagens enviadas a objetos compostos e objetos alcançados por parâmetros. Essa propagação amplia as oportunidades de concorrência, porque cada tabela de conflito é feita especialmente para o objeto da sua classe.

Aspectos de persistência nos objetos de aplicação durante a interação Nos itens (f) e (g), verificamos a entrada do *GOT* em cena, para providenciar a seleção do objeto destinatário, que pode ainda não ter sido selecionado. Essa situação acontece com este ambiente de execução, porque decidimos, com a finalidade de economizar memória, selecionar um objeto somente quando ele for o destinatário de alguma mensagem, cujo envio esteja sendo executado. Dessa forma, o meta objeto do objeto cliente envia a mensagem *MsgMr* ao *GOT*, que além das ações de persistência, providenciará a sua chegada ao meta objeto do objeto destinatário. Há três situações possíveis em que um objeto destinatário pode se encontrar: 1) administrado pelo mesmo *GOT* do objeto emissor e já selecionado; 2) administrado pelo mesmo *GOT* do objeto emissor e ainda não selecionado; e 3) administrado por um *GOT* diferente do *GOT* que administra o emissor, podendo estar na mesma máquina ou em máquina remota, além de apresentar uma das duas situações: a de já estar selecionado ou ainda não. Se o *GOT* administrador do destinatário não coincide com o do emissor, o do emissor deve retransmitir a mensagem *MsgMr* ao *GOT* do destinatário. Caso o objeto destinatário não se encontre selecionado, o *GOT* que administra o objeto destinatário deve providenciar, na memória da máquina em que reside, a criação de um meta objeto para o objeto de aplicação destinatário e a seleção do objeto de aplicação com os atributos restaurados da memória secundária. Estando tudo isso pronto, o *GOT* que administra o objeto destinatário deposita a mensagem *MsgMr* na caixa de mensagens do meta objeto do objeto destinatário.

Rastreamento dos threads Um contexto de transação carrega consigo todas as informações úteis necessárias aos mecanismos que efetuam os controles para garantir a atomicidade local e global. Para o rastreamento do *thread* lógico, o aninhamento de composição e de transação conta com os contextos de mensagens e de transação, como a seguir. Uma **`CMR`** deve ser registrada no contexto da transação cliente, quando o *id-Transação* não é nulo. Uma transação explícita pode ser registrada numa estrutura de **`CMR`** ou no contexto da transação aninhante direta e uma transação implícita sempre será aninhada numa estrutura de **`CME`** corrente. As **`CME`**'s, por sua vez,

podem ser registradas, ou em uma **CMR**, quando as mensagens *MsgMe*'s surgem na execução de um método e a transação do contexto coincide com a que está na **CMR**, ou em um contexto de transação local, quando a transação não coincide com a que está registrada na **CMR**. Como saldo dessas oportunidades, as gerências das transações também são beneficiadas, nos mecanismos de controles do assincronismo e nos aninhamentos de transações.

5. Mensagens responsáveis pela persistência dos objetos

O modelo Alana conta com duas características na persistência: a) a ortogonalidade a tipos e declarações de variáveis, permitindo que classes de objetos de aplicação sejam independentes de aspectos de sistemas, como, por exemplo, persistência, concorrência e transação e b) a individualidade na persistência (persistência rasa), justificada pela obrigatoriedade do controle de concorrência ser exercido sobre cada objeto, mesmo aquele passado por parâmetro, ou pertencente a uma composição, ou referenciado por uma variável local.

A propriedade da ortogonalidade oferece diversas vantagens, nas facilidades de sistemas, já consideradas para a persistência [1], e que foram estendidas para a transação, em [10]. Nesse último trabalho foram lembrados dois benefícios para a ortogonalidade na transação: 1) a programação da aplicação não fica poluída com considerações alheias à sua semântica, tais como as de persistência ou de transações; e 2) qualquer classe utilizada na programação da aplicação pode ser usada para instanciar objetos seqüenciais ou persistentes ou persistentes e transacionais. Nossa abordagem apresenta a característica de transação ortogonal, como em [10], porque o controle da persistência é exercido pelos mecanismos de atomicidade local, particularmente, na parte que trata *recuperação*, que se encontra embutida no protocolo de controle de concorrência, no meta objeto de cada objeto da aplicação.

Como, em uma aplicação com Alana, conforme já visto na seção 4, no parágrafo que trata a persistência durante a interação, um objeto somente deve ser selecionado se receber mensagem, antes disso, a sua referência, que pode se encontrar em alguma variável local, ou em algum atributo de algum outro objeto, ou em algum parâmetro passado, deve conter apenas a identificação do objeto, que é um *O-Id* [4]. A referência a um objeto pelo seu *o-Id* é a característica chave da persistência rasa, porque o estado de um objeto fica constituído apenas de uma lista de *o-Id*s, gravada em memória secundária. Contando com isso, quando uma transação está sendo finalizada, o estado de cada objeto participante deve ser gravado, e isso não significa a composição inteira, apenas aqueles que participaram da transação. Essas decisões tornam todos os objetos participantes de uma aplicação, que foi programada por uma linguagem que integra a meta arquitetura Alana, *transacionais*, assim, todos terão tratamento de recuperação, quer dizer, de persistência, mesmo os voláteis. Dessa forma, em vez de *persistência ortogonal*, nossa abordagem apresenta a característica de *transação ortogonal* [10], podendo ser abreviadamente identificada como transação ortogonal e individual.

A propriedade transacional de um objeto de aplicação é conseguida pelas suas características de *objeto ativo* e de *objeto atômico*, que são incorporadas na classe do elemento meta objeto, cuja instância é associada por reflexão. O meta objeto, que é exclusivo a cada objeto da aplicação, capacita a solução de objeto ativo e de atomicidade local, resolvendo a troca de mensagens assíncronas, a sincronização de competição (exclusão mútua) dessas mensagens, a sincronização de cooperação, e a recuperação.

A ortogonalidade da transação fica resolvida com a transparência no código dos métodos, assim, as mensagens enviadas precisam ser interceptadas pelo tratamento de mensagem ainda no código gerado do método do objeto emissor. O que acontece daí em diante já foi descrito nos itens de (a) a (t), da seção 4.1, acima. Somente no momento em que está para ser pedida a criação de um objeto então é possível fazer referência, na programação, de aspectos de sistema vinculados ao objeto de aplicação, sendo criado. Nessa situação única, o programador pode optar para que o objeto de aplicação a ser criado seja ou não persistente, tendo além disso a chance de escolher um outro gerente de objetos e de transações diferente do gerente *default* (gerente do objeto emissor dessa mensagem de criação).

Para nós a solução ideal para a persistência rasa seria o alcance pela referência, diretamente, considerada um *O-Id*, a ser resolvida no código gerado pelo compilador, cujo tratamento de mensagem deveria incluir mecanismos que resolvessem a interceptação e a funcionalidade de um protocolo de meta objeto. Próxima dessa solução ideal foram feitas algumas tentativas: 1) utilizar alguma linguagem que oferecesse persistência ortogonal, como a definida para PS-Algol [ABC1983], em que os objetos são alcançáveis por uma raiz especial, considerando todos os objetos da aplicação como raízes. Essa solução resolveria a necessidade da persistência rasa e foi tentada em uma versão de Alana com Guaraná [17], que oferece persistência ortogonal e *meta object protocol*, mas como não interceptava a mensagem no envio, não foi possível resolver as necessidades de Alana; 2) utilizar a própria linguagem Java™ [20], mostrando-se complicado, porque precisou de esforço de programação para garantir a persistência rasa para os objetos de aplicação, evitando a propagação da

persistência pelo seu grafo de referência, tendo sido parcialmente implementada para a meta arquitetura Alana. O emprego do padrão *proxy* para auxiliar a solução de persistência rasa deve ser também considerado e está sendo estudado por um grupo de alunos.

Vamos descrever as mensagens de persistência, considerando que ela surja em três momentos: 1) no momento da criação de um objeto da aplicação; 2) durante o envio de uma mensagem a um objeto, quando esse objeto destinatário deve então ser selecionado, para receber e atender à mensagem, já descrito, na seção 4, dedicada às mensagens de interação; e 3) durante as ações de recuperação, cujos mecanismos se encontram embutidos no protocolo de atomicidade local, de cada meta objeto associado a cada objeto de aplicação, quando é necessário que o seu estado seja salvo (permanentemente, ou transitoriamente), na fase de gravação da finalização de uma transação. Desses três momentos, o grupo de mensagens correspondentes ao momento (2), já foi descrito no comportamento das mensagens na persistência, na seção 4.1 que trata as mensagens de interação. Assim, a descrição, em seções à frente, tratará das mensagens relacionadas à persistência, correspondentes aos dois grupos ainda não apresentados: o correspondente ao momento (1), o da criação de um objeto e o correspondente ao momento (3), o da fase de gravação na finalização de uma transação.

5.1. Mensagens de persistência relacionadas à criação de um objeto de aplicação

Um objeto de aplicação passa a existir, quando um método de um outro objeto da aplicação executa um envio de mensagem de criação a uma classe, em outras palavras, invoca um construtor de classe, como acontece, naturalmente, na programação orientada a objeto. Concluído esse envio de mensagem, o objeto de aplicação que foi criado pode passar a persistir temporariamente. Se o programador indicou a opção de persistência, preenchendo o parâmetro que identifica o *GOT default* ou um outro *GOT* residente localmente ou remotamente, o objeto de aplicação terá persistência definitiva. Embora pareça estranho, a persistência temporária é necessária por causa do aspecto de recuperação de uma transação e um objeto de aplicação com essa característica permanecerá na memória secundária enquanto durar a transação, devendo, após a sua conclusão, ser removido da memória secundária.

O ciclo percorrido na execução do envio de mensagem de criação, cujo resultado no retorno é uma referência (**`ob_Id`**) do objeto criado, é descrito nos passos, a seguir:

a) inicialmente, o envio da mensagem, no tratamento de mensagem do código do objeto emissor, é transformado em uma mensagem *MsgCrOb*, sendo depositada na caixa de mensagens do meta objeto desse objeto emissor;

b) o meta objeto do objeto emissor, quando retira de sua caixa uma mensagem do tipo *MsgCrOb*, toma as seguintes providências: i) inclui o contexto da mensagem na tabela de objetos criados por seu objeto de aplicação; ii) cria e prepara uma nova mensagem da classe *MsgCrOb*, preenchendo alguns atributos correspondentes ao objeto da aplicação que será criado, que serão usados na criação do objeto, pelo *GOT*, entre eles, por exemplo, a classe do objeto; iii) deposita a mensagem na caixa de mensagens do *GOT*;

c) o *GOT*, quando retira de sua caixa uma mensagem do tipo *MsgCrOb*, providencia o seu tratamento pelo método *MetCrOb*, cujas ações estão descritas nos itens seguintes;

d) inicialmente, no *GOT*, no método *MetCrOb*, é verificado se o *self* coincide com o *GOT* destino, que, em caso negativo, providencia a retransmissão da mensagem *MsgCrOb* para o *GOT* destino, com o auxílio da operação **`TransmiteParaOutroGerente (...)`**;

e) no *GOT*, no método *MetCrOb*, o segundo passo a ser tomado é a invocação do construtor de descritor de objeto, com o parâmetro **`m.c_CrOb`**;

f) no construtor de descritor de objeto, em primeiro lugar, são criados o meta objeto, a caixa de mensagens e o objeto seqüencial, que passam a existir no ambiente de execução;

g) no construtor de descritor de objeto, em segundo lugar, é providenciada, pela primeira vez, a gravação do estado do objeto, no **`end_FisicoPrincipal`**;

h) no construtor de descritor de objeto, após outras ações, é providenciado o retorno ao método **`MetCrOb`** do *GOT*, que continuará sua execução;

i) continuando, então, no *GOT*, no método *MetCrOb*, o terceiro passo, a ser tomado é a inclusão do descritor, já preenchido, na tabela de objetos do *GOT*;

j) no *GOT*, no método *MetCrOb*, o quarto passo a ser tomado é o preenchimento das informações, no contexto **`m.c_CrOb`**: i) **`ob_Id`** – a informação mais importante, nesse ciclo, lembrando que é por esse identificador que o objeto passará a ser referenciado, sendo o resultado da invocação do lado seqüencial; ii) **`cx_Msg`**; iii) **`swizz_Id`**;

k) no *GOT*, no método *MetCrOb*, o quinto passo a ser tomado é a criação da mensagem de retorno *RetCrOb* **`r`**, preenchendo o seu atributo de contexto **`r.c_CrOb`** com **`m.c_CrOb`** e o atributo **`r.id_EmissorMS`** com o valor do atributo de **`m.id_EmissorMS`**;

l) no *GOT*, no método *MetCrOb*, o sexto passo a ser tomado é o empacotamento dessa mensagem **`r`**, com os dois parâmetros: o de origem preenchido com **`elemento_Id`** e o de destino com o valor de

m.origem_;

m) no *GOT*, no método *MetCrOb*, o sétimo passo a ser tomado é a transmissão desse pacote ao elemento destino, o meta objeto do objeto emissor, cujos desdobramentos resultam no depósito da mensagem **r** na caixa de mensagens desse meta objeto;

n) no meta objeto do objeto emissor, quando uma mensagem do tipo *RetCrOb* é retirada de sua caixa de mensagens ela é tratada pelo método **TrataMsgNoMonitor (...)**, cujas ações são descritas nos itens seguintes;

o) no método **TrataMsgNoMonitor (...)**, inicialmente, é buscada na estrutura **t_ObjsCriados** do meta objeto, o contexto correspondente a **r.c_CrOb**, para que seus atributos sejam preenchidos com os novos valores de: meta objeto, objeto seqüencial, caixa de mensagens, swizzling;

p) no método **TrataMsgNoMonitor (...)**, em segundo lugar, é providenciado o registro do objeto recentemente criado nas estruturas do meta objeto: **t_Swizziling**, **t_AcessadosAtivos**, e **t_ObjsCriados**;

q) no método **TrataMsgNoMonitor (...)**, finalmente, o contexto **m.c_CrOb** é transmitido para o objeto seqüencial emissor da mensagem de criação;

no objeto seqüencial emissor da mensagem de criação, entre outras ações, será providenciado o retorno propriamente dito com o valor de **ob_id** como resultado.

5.2. Mensagens de persistência relacionadas à gravação de um objeto de aplicação

A gravação do estado de um objeto de aplicação, na memória secundária, é efetuada tanto, no momento de sua criação, como já foi visto acima, quanto, na fase de gravação de uma transação que tenha conseguido chegar ao seu *commit*, quer dizer, não passou por qualquer situação de conflito ou de falha. Na fase de gravação de uma transação, as mensagens *MsgGr*, *RetGr*, *MsgCfGr* e *RetCfGr* são tratadas pelos elementos da meta arquitetura, desenvolvendo um ciclo, como podemos ver, mais a frente.

Fazemos notar, neste ponto, que uma transação pode surgir de duas formas: 1) explícita, quando é executado o comando **Begin transaction**, no código de algum método, de algum objeto qualquer da aplicação; ou 2) implícita, quando uma mensagem é enviada pelo método pertencente somente ao objeto de aplicação *top level* e este envio de mensagem não pertence à lista de comandos de algum comando **Begin transaction**. O meta objeto do objeto *top level* identifica essa situação (2) pelo valor **NULL** no contexto de transação agregado a mensagem *MsgMe* que ele estaria começando a tratar. Devido a essa condição, antes do tratamento dessa mensagem, o meta objeto deve fazer iniciar uma transação e, quando a *MsgRe* retorna, ele deve fazer terminar a transação. A necessidade de ser criada uma transação implícita na situação (2) se deve ao compromisso assumido, pelo modelo Alana, em assegurar que toda a mensagem de interação entre os objetos tem agregado um contexto de transação, tornando cada mensagem transacional.

Para qualquer uma dessas duas formas de surgimento, a transação deve ser iniciada por uma mensagem *MsgIT* e encerrada por uma mensagem *MsgFT*, que devem ser emitidas pelo meta objeto do objeto de aplicação. A mensagem *MsgIT*, que providencia a criação da transação, é enviada ao *GOT* e a mensagem *MsgFT*, que inicia o processo de finalização da transação, é enviada ao gerente de transação.

Vamos pular os detalhes de como funciona o ciclo desde uma mensagem *MsgIT* até uma mensagem *MsgFT* e ir diretamente aos aspectos relacionados à gravação, que estamos abordando nesta seção. O gerente de transação, ao tratar uma mensagem *MsgFT*, verifica se a transação possui algum participante com alguma mensagem de interação pendente. Se a transação apresenta essa condição, seu gerente passa para a fase (estado) de aguarda propagação, caso contrário, passa para uma das seguintes fases (estados), dependendo da política da transação: se pessimista, para a de gravação, e se otimista, para a de consistência. Se a transação otimista for bem sucedida na fase de consistência ela passará a fase de gravação.

A gravação é realizada em duas fases: na de gravação e na de confirmação de gravação. Para qualquer tipo de transação, com política pessimista ou otimista, o gerente da transação que está sendo *committed*, solicitará a todos os meta objetos dos objetos da aplicação que são seus participantes que efetuem a gravação. Em algum momento seguinte, se todos esses pedidos de gravação retornarem em condição normal ao gerente da transação, este, então, procederá a segunda fase da gravação, solicitando a todos os meta objetos de seus objetos participantes a confirmação da gravação.

A seguir, descrevemos os passos dados durante um ciclo de gravação, a partir do momento em que o gerente de transação entrou na fase de gravação:

a) para qualquer política de transação, o gerente de transação envia a mensagem *MsgGr* a todos os seus objetos de aplicação participantes, quer dizer, deposita a mensagem *MsgGr* na caixa de mensagens do meta objeto de cada objeto de aplicação participante;

b) o meta objeto de cada objeto participante da transação, ao retirar uma mensagem do tipo *MsgGr*, deve tomar as providências, na seguinte ordem: i) atualizar os efeitos das operações no estado permanente (ainda na memória principal) e ii) solicitar, via mensagem *MsgGr*, enviada ao *GOT*, a gravação desse

estado na memória secundária;
c) no *GOT*, uma mensagem *MsgGr*, retirada de sua caixa de mensagens, é tratada pelo método *MetGr*, que providencia a gravação do estado do objeto de aplicação, que se encontra no contexto de mensagem grava objeto (*CMGr*), agregado à mensagem *MsgGr*. Essa gravação é efetuada no *end_FisicoAlternativo*, que deve ser o endereço utilizado para gravar o estado do objeto de aplicação;
d) no *GOT*, em seguida, é criada uma mensagem *RetGr* e a condição da gravação, em (c), é preenchida no atributo *cond_* dessa mensagem, sendo depois enviada ao meta objeto do objeto de aplicação;
e) em cada meta objeto, uma mensagem *RetGr*, retirada de sua caixa, é tratada pelo método *MetRGr*, cujas ações principais são providenciadas, na seguinte ordem: i) verifica a condição de gravação, para que, em caso de falha, providencie o tratamento de erro local; e ii) com qualquer condição retornada, reencaminha essa mensagem ao gerente de transação;
f) no gerente de transação, uma mensagem *RetGr*, retirada de sua caixa, é tratada pelo método *MetRGr*, cujas ações principais são providenciadas, na seguinte ordem: i) verifica a condição de gravação, para que, em caso normal, registre esse retorno, preenchendo o atributo *gravada_* do participante com **TRUE** e, em caso de falha, providencie o tratamento de erro adequado, conforme poderemos ver na seção, a frente, dedicada a atomicidade local e global, ii) verifica se todos os objetos participantes já retornaram essa mensagem *RetGr*, quer dizer, se todos os participantes estão com o valor **TRUE** no atributo *gravada_*, para então providenciar a passagem para a fase (estado) de confirmação da gravação e, ao mesmo tempo, providenciar a confirmação da gravação para todos objetos participantes, enviando aos meta objetos dos participantes a *MsgCfGr*;
g) o meta objeto de cada objeto participante da transação, ao retirar uma mensagem do tipo *MsgCfGr*, deve solicitar, via a mesma mensagem *MsgCfGr*, ao *GOT*, a confirmação da gravação do estado de seu objeto de aplicação, gravando-o no endereço principal;
h) no *GOT*, uma mensagem *MsgCfGr*, retirada de sua caixa de mensagens, é tratada pelo método *MetCfGr*, que providencia a cópia do estado armazenado em *end_FisicoAlternativo* para o *end_FisicoPrincipal*. Com esses cuidados, sempre que um objeto for ativado, como seu estado é obtido de *end_FisicoPrincipal*, garantimos que é trazido aquele estado já confirmado. Após a gravação do estado do objeto de aplicação no endereço principal, esse método cria uma mensagem *RetCfGr*, preenche a condição dessa gravação no seu atributo *cond_* e devolve esse retorno ao meta objeto;
i) em cada meta objeto, uma mensagem *RetCfGr*, retirada de sua caixa, é tratada pelo método *MetRCfGr*, cuja ação, no caso de condição normal, é apenas a de reencaminhar essa mensagem ao gerente da transação e, no caso de erro, é a de tratar o erro localmente. Como essa é a última mensagem da fase de finalização de uma transação, a fase (estado) da transação no meta objeto passa a ser a de execução normal;
j) no gerente de transação, uma mensagem *RetCfGr*, retirada de sua caixa, é tratada pelo método *MetRCfGr*, que verifica se o participante já recebeu o pedido de confirmação, preenchendo o seu atributo *pedido_Confirmar* com o valor **TRUE**;
k) ainda no gerente de transação, no método *MetRCfGr*, caso, em (j), o participante já tenha recebido o pedido de confirmação, é verificada a condição de confirmação de gravação, para que, em caso normal, seja registrado esse retorno, preenchendo o atributo *confirmada_* do participante com **TRUE** e, em caso de falha, seja providenciado o tratamento de erro adequado, conforme poderemos ver na seção, à frente, dedicada a atomicidade local e global;

ainda no gerente de transação, no método *MetRCfGr*, se em (k) foi confirmada a gravação do participante, então é verificado se todos os objetos participantes já receberam o pedido de confirmação, quer dizer, se todos estão com o valor **TRUE**, no atributo *pedido_Confirmar*, para então ser verificado também se todos os participantes retornaram a confirmação da gravação, quer dizer, estão com o valor **TRUE**, no atributo *confirmada_*, para que seja então providenciado o envio da mensagem *MsgFNT* ao *GOT*, para que esse encerre a transação.

6. Mensagens que colaboram com a atomicidade local e global

As mensagens que vão colaborar com as fases das transações são as seguintes: 1) o par *MsgIT*/*RetIT*; 2) *MsgFPrimFTP*; 3) o par *MsgCons*/*RetCons*; 4) o par *MsgGr/ RetGr*; 5) *MsgCfGr/RetCfGr*; 6) *MsgLb*, ; 7) *MsgFalha* ; 8) *MsgFT*; e 9) *MsgDAtOb*.

Outro aspecto que também já tínhamos observado, acima, é a possibilidade de uma transação surgir de duas formas: 1) de forma explícita, durante a execução de um comando **`Begin transaction`** em um método de um objeto da aplicação; 2) de forma implícita, durante a execução de uma instrução de envio de mensagem em um método do objeto considerado o principal (*top level*) da aplicação, que não se encontra aninhada por um comando **`Begin transaction`**. Qualquer que seja a forma, explícita ou implícita, pela qual uma transação tenha surgido, o meta objeto envia uma mensagem *MsgIT* ao *GOT*, que por sua vez providencia a criação de um gerente de transação, que será o responsável por controlar os passos dessa transação, relacionados a atomicidade global.

O término de uma transação é inicialmente sinalizado pela *MsgFT*, que surge também de acordo com a forma pela qual a transação foi iniciada, como a

seguir: 1) na forma explícita, em que obrigatoriamente as duas condições seguintes precisam ser combinadas: 1-a) a execução do comando **`Begin transaction`** atinge o seu delimitador de bloco **`End transaction`** e, 1-b) a transação se encontra em sua última fase, de acordo com a sua política; ou 2) na forma implícita, quando chega a mensagem *MsgRe* que combina com a mensagem *MsgMe* que originou a transação implícita.

A mensagem *MsgFT* não inicia efetivamente o processo de *commit*, porque estamos considerando que todas as mensagens possam estar no modo assíncrono, e, assim, o gerente de transação vai verificar se há ou não mensagem pendente de retorno, para, respectivamente, passar para a fase de aguarda propagação ou a fase de início de um *commit*, que pode ser a fase de consistência para as transações otimistas ou a fase de gravação para as transações pessimistas.

As mensagens, que surgem em um ciclo de *MsgIT* a *MsgFT*, circulam pelos elementos da meta arquitetura, os quais cooperam, assumindo os seguintes papéis: 1) o protocolo localizado no elemento meta objeto garante a atomicidade local, para cada transação que compartilhe o seu objeto de aplicação[1]; 2) a atomicidade global fica a cargo do gerente de cada transação, cujas fases seguem a seqüência determinada pelas transições; e 3) o controle sobre: o número de transações pessimistas ativas; as pendências entre transações; e os possíveis *deadlock*s serão tratados com a ajuda do gerente de objetos e de transações *GOT*. Os mecanismos presentes em todos esses três elementos levam em conta a política de uma transação, conduzindo-os a procedimentos e decisões diferentes, como poderemos ver pela descrição do ciclo, mais adiante.

Esse ciclo, desenvolvido a partir do envio de uma *MsgIT* pelo meta objeto ao *GOT* até a finalização efetiva de uma *MsgFT*, cuja última ação é a destruição do gerente da transação, a pedido dele próprio ao *GOT*, tem um tempo de duração mais longo do que os outros ciclos descritos, nas seções anteriores, muitas vezes coincidindo com o tempo de vida da aplicação, como podemos ver, a seguir:

a) no meta objeto, uma criação de transação na forma explícita passa pelos seguintes passos: i) o recebimento de uma *MsgIT* do objeto da aplicação, que é tratada pelo método *MetIT*; ii) no *MetIT* do meta objeto, é invocado o construtor **`CriaCTLocal (...)`** de contexto de transação local, cujos parâmetros são: a indicação de política otimista ou pessimista, conforme estava no comando **`Begin transaction`**, e o identificador da estrutura pertencente ao meta objeto, aonde esse contexto deverá ficar guardado, essa estrutura pode ser ou um contexto de transação local pertencente a um contexto de **`CMR`**, se houver aninhamento de transações no código e a transação já é aninhada, ou a própria **`CMR`**, se é uma transação simples ou se é a raiz de um aninhamento de transações; iii) no construtor **`CriaCTLocal (...)`** a mensagem *MsgIT* é reenviada ao gerente de objetos e transações *GOT*;

b) no meta objeto, uma criação de transação na forma implícita passa pelos seguintes passos: i) no recebimento de uma *MsgMe* do objeto da aplicação, no seu método **`TrataMsgNoMonitor`**, é detectada a necessidade de uma transação implícita, devido ao valor nulo encontrado em **`c_Trans.id_Transacao`**; ii) ainda, no método **`TrataMsgNoMonitor`**, da *MsgMe*, é invocado o construtor **`CriaCTLocal (...)`** de contexto de transação local, cujos parâmetros são a indicação de política otimista como *default* e o identificador da estrutura pertencente ao meta objeto, aonde esse contexto deverá ficar guardado, essa estrutura é um contexto de **`CME`** pertencente a um contexto de **`CMR`**; iii) no construtor **`CriaCTLocal (...)`** a mensagem *MsgIT* é reenviada ao gerente de objetos e transações *GOT*;

c) para ambas as situações, acima, no *GOT*, a mensagem *MsgIT* é tratada no método *MetIT*, cujas ações principais são: cria um gerente de transação; preenche as informações de aninhamento de transação no seu contexto de transação; inclui o contexto de transação na lista de transações do *GOT*, cria uma mensagem *RetIT* e preenche o seu contexto de transação com as informações atualizadas por esse próprio método; e envia essa mensagem ao gerente de transação que foi criado neste método, que se encontra no estado inicial **`InicialTO`** ou **`InicialTP2PL`**, devendo essa mensagem ser tratada pelo método *MetRIT*, desse estado inicial, nesse primeiro destino.

Cabe esclarecer, neste ponto, em primeiro lugar, que as classes **`TransaçãoOtimista`** e **`TransaçãoPessimista`** herdam a classe **`Transação`** (gerente da transação). E, em segundo lugar, que as classes para os estados iniciais das transações **`InicialTO`** e **`InicialTP2PL`** herdam a classe **`InicialTransação`** e, assim, o método *MetRIT*, que pertence a essa classe, será herdado por esses estados iniciais específicos. A coerção do estado inicial da transação é efetuada pela operação privada **`IniciaTransação`**, que é chamada no momento da criação, pelo construtor da classe **`TransaçãoOtimista`** ou **`TransaçãoPessimista`**, para os estados, respectivamente, **`InicialTO`** ou **`InicialTP2PL`**.

Cabe também esclarecer porque descrevemos nos dois itens seguintes o recebimento da mensagem *RetIT*

[1] Nesse contexto, esse objeto de aplicação também é considerado como um dos participantes da transação.

pelos dois gerentes de transação. No primeiro item, descrevemos o recebimento, no gerente da transação recentemente criada, e, no segundo item, no da transação aninhante. O método *MetRIT*, que trata a mensagem *RetIT*, efetua um novo envio dessa mesma mensagem ao gerente da transação aninhante, para que sejam registradas informações de controle das transações aninhadas.

d) no gerente da transação recentemente criada, quando a mensagem *RetIT* é retirada de sua caixa de mensagens e passada ao método *MetRIT*, para que seja tratada, inicialmente, são comparados os identificadores do gerente de transação que vem no contexto de transação da mensagem e o do *self*: se esses identificadores coincidirem, a mensagem é reencaminhada para dois destinos: i) para o meta objeto do objeto de aplicação de onde partiu a *MsgIT* e ii) para o gerente da transação aninhante dessa transação;

e) no gerente da transação aninhante da transação recentemente criada, quando a mensagem *RetIT* é retirada de sua caixa de mensagens e passada ao método *MetRIT*, para que seja tratada, inicialmente, são comparados os identificadores do gerente de transação que vem no contexto de transação da mensagem e o do *self*: se esses identificadores não coincidirem, significa que o *self* é o gerente da transação aninhante da transação recentemente criada, assim, o contexto da transação recentemente criada é incluído na lista de transações aninhadas desse gerente da transação aninhante;

Fazemos notar neste ponto que o estado da transação recentemente criada permaneceu em **`InicialTO`** ou **`InicialTP2PL`**, dependendo da política da transação.

f) no meta objeto do objeto de aplicação de onde partiu a *MsgIT*, quando a mensagem *RetIT* é retirada de sua caixa de mensagens e passada ao método *MetRIT*, para que seja tratada, nesta versão básica de Alana, três tipos de início de transação podem ter ocorrido, causando três ações diferentes nesse método *MetRIT*: i) a transação pedida foi implícita e portanto otimista (*default*): busca a mensagem enviada que detectou a ausência da transação em seu contexto, na lista de mensagens enviadas que estão esperando por uma transação implícita do meta objeto; atualiza o contexto da mensagem enviada com essa transação recentemente criada; e retoma o fluxo de execução da mensagem enviada, invocando o seu método **`TrataMsgNoMonitor (...)`**; ii) a transação pedida foi explícita e otimista: reenvia a mensagem *RetIT* à versão seqüencial; e iii) a transação pedida foi pessimista: reenvia a mensagem *RetIT* à versão seqüencial;

Por razões de clareza, colocamos separadamente neste parágrafo a descrição das mudanças de estado que podem ser efetuadas pelo método *MetRIT* do meta objeto, que pode ser encontrado nos seguintes estados: i) se em **`InicialMonitor`**, pode mudar para **`TOsEmCurso`** se transação otimista, ou para **`TPPrimFase`**, se transação pessimista *P2PL*; ii) se em **`InicialTPAninhada`**, pode mudar para **`TPPrimFase`**; iii) se em **`TOsEmCurso`**, *MetRIT* não muda o estado; iv) se em **`TPPrimFase`**, *MetRIT* não muda o estado; e v) se em **`TPSegunFase`**, *MetRIT* não muda o estado.

Alguns aspectos devem ser observados sobre o início dessas transações, como a seguir. Uma transação implícita por exemplo não pode ser aninhada, porque ela é iniciada somente se for detectada a ausência de transação no contexto de uma mensagem *MsgMe*. Uma transação explícita pode ser iniciada em um método do objeto principal da aplicação (*top level*), podendo nessa situação ser uma raiz ou uma aninhada. Uma transação explícita também pode ser iniciada em um método de qualquer outro objeto da aplicação, pela composição, devendo nessa situação ser sempre aninhada. Uma transação surgida em qualquer uma dessas três situações poderá ser aninhante de uma outra transação, se, em um método pertencente a propagação de qualquer envio de mensagem executado no interior da região de código abrangida pela transação, for executada outra iniciação de transação, que, por sua vez, também poderá ser aninhante de outra, e, assim, por diante.

Fazemos notar que, a partir desse item (f), o último acima, os caminhos pelos meta objetos, gerentes de transação e pelo próprio *GOT* serão diferentes, dependendo da política da transação, sendo mais apropriado descrevê-los em seqüências separadas, a seguir.

Comportamento de uma transação otimista após a fase inicial O gerente da transação muda do estado **`InicialTO`** para **`TOExecutando`**, quando ele recebe uma mensagem *MsgMe* ou *MsgMr*. Parece estranho ocorrer uma *MsgMr* antes de uma *MsgMe*, mas, como estamos tratando o modo assíncrono nas mensagens, imaginamos essa possibilidade.

1) Fase de execução – nessa fase, o gerente de transação é sinalizado a cada mensagem de um tipo *MsgMe*, *MsgMr*, *MsgRr* e *MsgRe* que surge no ambiente, como podemos ver na seção 4.1, dedicada às mensagens de interação, descritas através dos itens de (a) a (t).

1) Fases de finalização

a) quando chega, no meta objeto do objeto de aplicação, uma mensagem *MsgFT*, como já foi dito acima, não significa que, necessariamente, o processo de finalização deva começar, porque estamos considerando que todas as mensagens possam estar no modo assíncrono. O meta objeto, levando esse fato em conta, deve, apenas, reencaminhar essa mensagem para o gerente da transação;

b) o gerente da transação, ao apanhar uma mensagem

do tipo *MsgFT* de sua caixa de mensagens, verifica se há ou não mensagens pendentes de retorno. Caso essa pendência seja detectada, o estado da máquina de estados passa a ser **`TOAgPropag`**; e, caso contrário, o estado da máquina de estados passa a ser **`TOConsiste`**;

c) o gerente de transação, no estado **`TOAgPropag`**, ao retirar cada mensagem de interação do tipo *MsgMe*, *MsgMr*, *MsgRr* e *MsgRe*, deve registrar sua chegada em seus controles, e se a mensagem for do tipo *MsgRe*, além desse registro, deve efetuar o casamento com a *MsgMe* correspondente e varrer seus controles, para detectar a presença/ausência de pendências de mensagens enviadas e recebidas. Se essa varrição resultar na condição de ausência de pendências, a operação privada do gerente de transação **`OpConsisteTNosParticipantes`** é invocada e o estado do gerente de transação passa a ser **`TOConsiste`**;

d) na operação **`OpConsisteTNosParticipantes`**, do gerente de transação, todos os participantes da transação receberão uma mensagem de consistência. Se a transação não é aninhada, a mensagem de consistência enviada será *MsgCons* (mais completa) e se é aninhada, a mensagem de consistência enviada será *MsgSoCons* (apenas consiste);

e) no meta objeto do objeto de aplicação (o participante da transação), quando é retirada da caixa de mensagens uma mensagem do tipo *MsgSoCons*, ela é tratada pelo método **`MetSoConsiste`**, do estado **`TosEmCurso`** do meta objeto, que por sua vez invoca a operação privada do meta objeto, de nome **`OpConsisteTO`**, que retorna a mensagem *RetCons*, com a condição de conflito ausente/presente preenchida no seu atributo **`cond_`**. Essa mensagem de retorno da consistência é enviada ao gerente de transação;

f) no meta objeto do objeto de aplicação (o participante da transação), quando é retirada da caixa de mensagens uma mensagem do tipo *MsgCons*, ela é tratada pelo método **`MetConsiste`**, do estado **`TosEmCurso`** do meta objeto, que por sua vez invoca a operação privada do meta objeto, de nome **`OpConsisteTO`**, que retorna a mensagem *RetCons*, com a condição de conflito ausente/presente preenchida no seu atributo **`cond_`**. Essa mensagem de retorno da consistência é enviada ao gerente de transação;

g) no gerente de transação, quando é retirada da caixa de mensagens uma mensagem do tipo *RetCons*, ela é tratada pelo método **`MetRCons`**, do estado **`TOConsiste`** do meta objeto, que registra a condição normal, no caso de ausência de conflito, quando então o estado do gerente de transação passa para a fase seguinte normal que é a de **`TOGrava`**. Em seguida, o método varre os controles para verificar se todas os participantes já retornaram e foram bem sucedidos, e, nesse caso, quando a a transação é a raiz do aninhamento, é invocada a operação privada **`ProvidenciaGrDesdeRoot`**.

h) no gerente da transação, na operação privada **`ProvidenciaGrDesdeRoot`**, uma mensagem *MsgGr* é criada e enviada ao gerente da transação raiz;

i) no gerente da transação, quando é retirada da caixa de mensagens uma mensagem do tipo *MsgGr*, ela é tratada pelo método **`MetGr`**, do estado **`TOGrava`**, que atende a essa mensagem somente se o emissor (um gerente de transação) for o próprio ou o gerente da transação ascendente, quando então é invocada a operação privada **`OpGravaTNosParticipantes`**;

j) no gerente da transação, na operação privada **`OpGravaTNosParticipantes`**, é enviada uma mensagem *MsgGr* ao meta objeto de cada objeto de aplicação participante da transação;

O que acontece com essa mensagem *MsgGr* no meta objeto e todos os desdobramentos até o momento de ser providenciado o envio da mensagem *MsgFNT* ao *GOT* já encontra-se descrito no parágrafo entitulado *Mensagens usadas na persistência relacionadas à gravação de um objeto de aplicação*, da seção 4.2, acima.

k) no gerente de transação, uma mensagem *RetCfGr*, ao ser tratada pelo método *MetRCfGr*, do estado **`TOCfGrava`**, suas últimas providências são as verificações seguintes: i) se todos os objetos participantes já receberam o pedido de confirmação, quer dizer, se todos estão com o valor **`TRUE`**, no atributo **`pedido_Confirmar`**, para então ser verificado também ii) se todos os participantes retornaram a confirmação da gravação, quer dizer, estão com o valor **`TRUE`**, no atributo **`confirmada_`**, para que seja então providenciado o envio da mensagem *MsgFNT* ao *GOT* e a mudança do estado do gerente de transação para **`TfimNormal`**, que significa vai significar o real encerramento da transação;

no *GOT*, quando é retirada da caixa de mensagens uma mensagem do tipo *MsgFNT* (mensagem de fim normal de transação), ela é tratada pelo método **`MetFNT`**, que toma as seguintes providências: i) inclui o contexto de transação em **`f_TsFimNormal`** (fila das transações que alcançaram fim normal, quer dizer, sem conflito de concorrência e sem falha de leitura/gravação); ii) é invocada a operação privada **`CalculaNovoThreshold`**, o threshold é o timestamp mais velho de uma transação já finalizada mas que ainda pode causar conflito nas verificações entre as operações dessa transação com as outras ativas; iii) é invocada a operação privada **`LiberaTsMaisVelhasDoQueThreshold`**, a cada

finalização de transação, o valor do threshold pode subir, e as transações finalizadas com timestamps mais baixos podem então ser liberadas dos meta objetoes dos participantes, pelo envio da mensagem *MsgLb* e seus gerentes destruídos.

Comportamento de uma transação pessimista 2PL após a fase inicial O gerente da transação muda do estado **`InicialTP2PL`** para **`TP2PLPrimFase`**, quando ele recebe uma mensagem *MsgMe* ou *MsgMr*. Como já foi observado acima, no comportamento das transações otimistas, vimos que é possível ocorrer uma *MsgMr* antes de uma *MsgMe*, por causa do modo assíncrono nas m ensagens.

1) Fase de bloqueio (estado **`TP2PLPrimFase`**) – nessa fase, o gerente de transação é sinalizado a cada mensagem de um tipo *MsgMe*, *MsgMr*, *MsgRr* e *MsgRe* que surge no ambiente, como podemos ver na seção 4.1, dedicada às mensagens de interação, descritas através dos itens de (a) a (t).

a) quando chega, no meta objeto do objeto de aplicação, uma mensagem *MsgFT*, como já foi dito acima, para o comportamento das transações otimistas, não significa que, necessariamente, o processo de finalização deva começar, porque estamos considerando que todas as mensagens possam estar no modo assíncrono. O meta objeto, levando esse fato em conta, deve, apenas, reencaminhar essa mensagem para o gerente da transação;

b) o gerente da transação, ao apanhar uma mensagem do tipo *MsgFT* de sua caixa de mensagens, verifica se há ou não mensagens pendentes de retorno. Caso essa pendência seja detectada, o estado da máquina de estados passa a ser **`TP2PLAgPropagPrimFase`**; e, caso contrário, o estado da máquina de estados passa a ser **`TP2PLSegunFase`**;

c) o gerente de transação, no estado **`TP2PLAgPropagPrimFase`**, ao retirar cada mensagem de interação do tipo *MsgMe*, *MsgMr*, *MsgRr* e *MsgRe*, deve registrar sua chegada em seus controles, e se a mensagem for do tipo *MsgRe* ou *MsgRr*, além desse registro, deve efetuar o casamento com a *MsgMe* ou *MsgMr* correspondente e varrer seus controles, para detectar a presença/ausência de pendências de mensagens enviadas e recebidas. Se essa varrição resultar na condição de ausência de pendências, deve ser enviada uma mensagem *MsgFPrimFTP* aos participantes e uma mensagem *RetFT* ao meta objeto de onde se originou a *MsgFT*; e ser efetuada a coerção para que o estado do gerente de transação passe a ser **`TP2PLSegunFase`**;

A seguir dois tipos de mensagem *RetFT* ou *MsgFPrimFTP* poderão chegar a um meta objeto, dependendo da condição da mensagem *MsgIT* ter se originado ou não nele.

d) no meta objeto do objeto de aplicação (o autor do envio da mensagem *MsgIT* para o gerente da transação), quando é retirada da caixa de mensagens uma mensagem do tipo *RetFT*, ela é tratada pelo método **`MetRFT`**, do estado **`TPPrimFase`** do meta objeto, que toma as seguintes providências: i) o reinício de contexto da transação; ii) o reenvio dessa mensagem à versão do objeto de aplicação, cuja ação principal será a de fazer o código da aplicação recomeçar a partir do início do bloco interno ao comando de transação, para que então a execução propriamente dita tenha início, nessa segunda fase; iii) a coerção para que o estado do meta objeto passe a ser **`TPSegunFase`**;

e) no meta objeto do objeto de aplicação, quando é retirada da caixa de mensagens uma mensagem do tipo *MsgFPrimFTP*, ela é tratada pelo método **`MetFPrimFaseTP`**, do estado **`TPPrimFase`** do meta objeto, que toma as seguintes providências: i) o reinício de contexto da transação; e ii) a coerção para que o estado do meta objeto passe a ser **`TPSegunFase`**;

2) Fase de execução (segunda fase) – nessa fase, o gerente de transação é sinalizado a cada mensagem de um tipo *MsgMe*, *MsgMr*, *MsgRr* e *MsgRe* que surge no ambiente, como podemos ver na seção 4.1, dedicada às mensagens de interação, descritas através dos itens de (a) a (t).

f) no gerente de transação, no estado **`TP2PLSegunFase`**, quando é retirada uma mensagem do tipo *MsgFT*, ela é tratada pelo seu método **`MetFT`**, da classe **`TP2PLSegunFase`**, que verifica se na transação todas as mensagens já foram casadas com os seus retornos. No caso de todos os participantes estarem com esses pares casados, este método invoca a operação privada do gerente de transação **`ProvidenciaGrDesdeRoot`** e efetua a coerção para o estado **`TP2PLGr`**; e no caso de ainda encontrarem-se pendências, efetua a coerção para o estado **`TP2PLAgPropagSegunFase`**.

g) no gerente da transação, na operação privada **`ProvidenciaGrDesdeRoot`**, uma mensagem *MsgGr* é criada e enviada ao gerente da transação raiz;

h) no gerente da transação, quando é retirada da caixa de mensagens uma mensagem do tipo *MsgGr*, ela é tratada pelo método **`MetGr`**, do estado **`TP2PLGrava`**, que atende a essa mensagem somente se o emissor (um gerente de transação) for o próprio ou o gerente da transação ascendente, quando então é invocada a operação privada **`OpGravaTNosParticipantes`**;

i) no gerente da transação, na operação privada **`OpGravaTNosParticipantes`**, é enviada uma mensagem *MsgGr* ao meta objeto de cada objeto de aplicação participante da transação;

O que acontece com essa mensagem *MsgGr* no

meta objeto e todos os desdobramentos até o momento de ser providenciado o envio da mensagem *MsgFNT* ao *GOT* já encontra-se descrito no parágrafo entitulado *Mensagens usadas na persistência relacionadas à gravação de um objeto de aplicação*, da seção 4.2, acima.

j) no gerente de transação, uma mensagem *RetCfGr*, ao ser tratada pelo método *MetRCfGr*, do estado **`TP2PLCfGrava`**, suas últimas providências são as verificações seguintes: i) se todos os objetos participantes já receberam o pedido de confirmação, quer dizer, se todos estão com o valor **`TRUE`**, no atributo **`pedido_Confirmar`**, para então ser verificado também ii) se todos os participantes retornaram a confirmação da gravação, quer dizer, estão com o valor **`TRUE`**, no atributo **`confirmada_`**, para que seja então providenciado o envio da mensagem *MsgFNT* ao *GOT* e a mudança do estado do gerente de transação para **`TfimNormal`**, que vai significar o real encerramento da transação, no caso da transação do tipo pessimista 2PL.

Chamamos a atenção para as transações aninhadas, durante as fases de finalização, no tratamento das mensagens *RetCons*, *RetGr* e *RetCfGr*, que só podem ter as fases correspondentes concluídas, se no caminho de aninhamento das transações, todas forem concluídas e a transação da raiz da hierarquia é a finalizada, no momento. Essa medida se deve a possibilidade das transações aninhantes ainda terem chances de ser abortadas. Devemos lembrar que o nosso modelo de aninhamento é fechado e assim as transações aninhadas deverão terminar para que as transações aninhantes possam finalizar.

Para o bloqueio das transações aninhadas, os mecanismos dependem das políticas de controle de concorrência nas transações aninhadas e nas aninhantes, como a seguir. Se a transação aninhante é otimista e a transação aninhada é pessimista, ainda, outras medidas são necessárias, como, a seguir. Se a transação raiz da hierarquia em que se encontra a aninhada é a mais antiga e se as transações em curso dessa raiz pertencerem exclusivamente ao caminho da transação candidata corrente, então essa transação poderá ganhar a vez. Qualquer outra transação pertencente a um caminho diferente ou a uma hierarquia diferente é considerada conflitante (essa consideração é baseada na decisão tomada para os nossos mecanismos em que entre as mensagens associadas a transações otimistas e pessimistas, são atendidas aquelas com mais alta prioridade) e um bloqueio sobre elas poderá causar *deadlock*.

6.1. Tratamento dos Conflitos e Falhas nas transações

Uma falha pode surgir a qualquer momento, sinalizada pelo *GOT*, como, por exemplo, as falhas de comunicação entre os nós da rede, as falhas de leitura/gravação que podem variar desde o estado geral do ambiente de execução até o estado localizado de um objeto da aplicação. Um conflito pode surgir na consistência das transações otimistas ou na tentativa de bloqueio, na verificação entre as operações, nas transações pessimistas. Por motivos de simplificação das apresentações, acima, separamos a descrição dos tratamentos das condições de falhas e conflitos, que se encontram, a seguir.

Conflitos e falhas nas transações com política otimista Para as transações com políticas otimistas, os conflitos poderão surgir na fase de consistência, em cada meta objeto de um objeto de aplicação participante da transação, efetuada por um dos métodos **`MetSoConsiste`** ou **`MetConsiste`**, que invocam a operação privada **`OpConsisteTO`**, que retorna uma instância da mensagem *RetCons*, que é enviada ao gerente de transação.

No gerente de transação, essa mensagem *RetCons* é tratada pelo método **`MetRCons`**, que pode constatar dois tipos de conflito, dependendo da prioridade que a transação consistida tinha, como, a seguir: i) se o atributo **`cond_`** está com a condição *ConflitoAbortaPrópria*, então é invocada a operação privada **`ProvidenciaLiberaDesdeRoot`** e o estado do gerente de transação passa a ser **`TOAb`** (estado de transação sendo abortada), que fica a espera do retorno das liberações; ii) se o atributo **`cond_`** está com a condição *ConflitoAbortaOutra*, então essa outra transação é incluída na lista de transações a serem abortadas, em momento posterior, e a consistência da transação corrente é considerada normal. Ainda, no método **`MetRCons`**, como pode ter sido pedida a consistência de todas as transações por um caminho de aninhamento de transações, deve ser verificado se todas as transações já foram consistidas e se isso for constatado, e se a transação está, na raiz do aninhamento, então é invocada a operação privada **`ProvidenciaGrDesdeRoot`**.

Como foi dito acima, a qualquer momento, uma falha pode surgir, assim quase todos os estados tem tratamento para uma mensagem de falha. No gerente de transação, nos estados **`TOExecutando`** e **`TOAgPropag`**, quando é retirada uma mensagem do tipo *MsgFalha*, ela é tratada, pelo método **`MetFalha`**, que invoca a operação privada **`ProvidenciaLiberDesdeRoot`** e efetua a coerção para o estado **`TOAb`**.

Conflitos e falhas nas transações com política pessimista 2PL Para as transações com política pessimista *2PL*, as que estão na segunda fase estão livres de qualquer conflito. No método **`MetMr`**, do estado **`TPPrimFase`**, o bloqueio de um objeto de uma transação pessimista, na primeira fase, é tentado quando não se encontra bloqueado por uma outra transação na segunda fase, e em relação a uma outra

transação que esteja na primeira fase, obedecendo o critério da mais alta prioridade, que por *default* é o do *timestamp* mais antigo. Caso, essa outra transação, na primeira fase, já tenha bloqueado o objeto e tenha prioridade mais baixa, ela deve ser abortada. Podemos indagar porque não reinfileirar as mensagens em vez de abortar ? A explicação para isso é o fato de uma transação pessimista em sua primeira fase não ter produzido qualquer efeito no estado dos seus objetos participantes, não causando qualquer trauma ao ambiente e porque o retorno da condição de aborto poder ser resolvido semanticamente pelo comando de transação, no código do objeto da aplicação.

Devemos antes lembrar que, para a primeira fase, o meta objeto de cada objeto participante de uma transação pessimista conta com uma versão exclusiva para simular uma execução sem efeito e que, na segunda fase, cada meta objeto conta com uma versão compartilhada pelas transações que se encontram nessa fase, e que não é a versão correspondente ao estado permanente. Assim, nas duas fases, as transações podem ser abortadas sem maiores traumas. Como na transação otimista, a qualquer momento, uma falha pode surgir, assim quase todos os estados tem tratamento para uma mensagem de falha. No gerente de transação, nos estados **`TP2PLPrimFase`**, **`TP2PLAgPropagPrimFase`**, **`TP2PLSegunFase`**, quando é retirada uma mensagem do tipo *MsgFalha*, ela é tratada, pelo método **`MetFalha`**, que tem poucas diferenças, dependendo da primeira ou segunda fase: na primeira fase) invoca a operação privada **`ProvidenciaLiberaDesdeRoot`** e efetua a coerção para o estado **`TP2PLAbPrimFase`**; e na segunda fase) invoca a operação privada **`ProvidenciaLiberaDesdeRoot`** e efetua a coerção para o estado **`TP2PLAbSegunFase`**.

Considerações sobre estas mensagens que colaboram com a atomicidade local e global Observamos, nesta seção, no comportamento para as duas políticas de transação otimista e pessimista *2PL*, que o gerente de transação tem um papel restrito a administração, solicitando, através das mensagens: *MsgCons*, *MsgGr*, *MsgCfGr*, *MsgFT*, *MsgRFT*, *MsgFalha* a cada objeto participante de sua transação, que proceda, dependendo de que fase se encontre, as consistências, as gravações, as confirmações de gravação, as ações de fim normal de transação, ou as ações necessárias ao tratamento de uma falha. Essas tarefas, realizadas pelos meta objetos, enquadram-se no nível da atomicidade local, e, ao final desse longo ciclo, visto nos comportamentos acima, caso não tenha ocorrido nenhuma falha ou conflito, o gerente de transação pode dar a garantia da atomicidade global.

A atomicidade local ficou por conta do meta objeto, que, para o primeiro aspecto da atomicidade, o controle de concorrência, pode decidir, para a transação pessimista, a qual transação permitir o bloqueio do seu objeto participante, ou, para a transação otimista, na fase de consistência, se há ou não conflito entre as operações. No segundo aspecto da atomicidade, a recuperação, uma transação ao chegar ao seu final, garante a integridade e durabilidade ao estado dos seus objetos participantes, e caso tenha ocorrido algum conflito ou falha, destrói a transação, não deixando qualquer efeito sobre os estados dos seus objetos participantes.

7. Conclusões

Fazemos notar que a interface dos elementos é o conjunto de mensagens tratadas neste trabalho e que o aspecto do funcionamento, em que mostramos os ciclos percorridos por essas mensagens, foi dividido de acordo com os papéis exercidos pelas mensagens em três diferentes grupos, descritos abaixo.

Grupo 1 – Mensagens de Interação. Este ciclo é iniciado quando qualquer mensagem é enviada na execução de um método de um objeto da aplicação

Grupo 2 – Mensagens de Persistência. Este grupo possui três diferentes ciclos:

Ciclo de criação de um objeto – O ciclo se inicia na invocação de um construtor ou no ciclo de criação de um objeto durante a execução de um método de um objeto da aplicação

Gravação de um objeto da aplicação – Esse procedimento é decorrente de uma finalização de transação que pode ter sido pedida de duas formas: 1) se a execução em um método chegou ao final do bloco do comando de uma transação explícita e o gerente de transação já conferiu a chegada de todos os retornos das mensagens enviadas; ou 2) se o retorno da mensagem que gerou uma transação implícita chegou ao meta objeto (isso só acontece em objeto top level da aplicação).

Leitura de um objeto da aplicação – Esse procedimento é decorrente de um envio de mensagem a um objeto destinatário que ainda não tenha sido ativado, cuja descrição foi contemplada no ciclo de interação do grupo 1.

Grupo 3 – Mensagens que colaboram com a atomicidade local e global. Essas mensagens surgem na meta-arquitetura em um longo ciclo que começa com uma mensagem *MsgIT* originada do meta objeto para o *GOT*, cujo principal objetivo é criar um gerente de transação. Após a fase inicial de uma transação, ela entra na fase de execução, cuja maior parte está descrita no grupo 1, execução nas transações otimistas e primeira e segunda fase nas transações pessimistas. Quando o final da execução de uma transação é alcançado, é iniciada a seqüência de fases de finalização sendo diferentes para uma transação otimista e pessimista.

Chamamos a atenção para a necessidade de algumas

vezes a mensagem ser recriada com um contexto com informações mais restritas, porque não serão úteis para o elemento seguinte para o qual elas ainda deverão ser enviadas. Este é o caso da criação da mensagem externa para a *MsgMe*, quando ela é enviada ao gerente de transação, para avisar de si própria, além de comunicar a participação do objeto de aplicação.

Chamamos a atenção sobre a transação, que é a principal facilidade oferecida por essa meta arquitetura, sendo a persistência, considerada um aspecto derivado da atomicidade. Além do que, todos os esforços realizados na interação entre os objetos de aplicação tiveram a intenção de: a) evitar o conflito na execução das operações para as transações pessimistas, ou b) deixar registradas as informações necessárias para a consistência, nas transações otimistas. Essas providências (a ou b) são então aproveitadas mais tarde para a finalização da atomicidade local, nos objetos.

- Sobre a atomicidade local, lembramos que ela constitui-se de dois aspectos: o controle de concorrência, que evita conflitos entre operações; e a recuperação, que providencia a gravação em caso de êxito na transação ou impede estes efeitos, no caso de conflito ou falha. Essa atomicidade local ficou sob a responsabilidade do meta objeto.
- A atomicidade global ficou sob a responsabilidade do gerente de transação, que tem a capacidade de receber as notificações do meta objeto de cada objeto participante, referentes à atomicidade local.

Referências

[1] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An Approach to Persistent Programming. The Computer Journal. 26(4), 1983.

[2] Antonio Albano, Luca Cardelli and Renzo Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. ACM Transactions on Database Systems. 10 (2), June 1985.

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

[4] Serge Abiteboul and Paris Kanelakis. Object Identity as a Query language Primitive. In Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data. 1989.

[5] 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.

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

[7] 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.

[8] Estudo das influências de Troca de Mensagens Assíncronas nos Mecanismos que Implementam a Atomicidade Local. Cadernos do IME - Série Informática - Vol.13 -Dezembro 2002. www.ime.uerj.br.

[9] S. J. Caughey, M. C. Little, and S. K. Shrivastava. Checked Transactions in na Asynchronous Message Passing Environment. Technical Report — Department of Computing Science, University of Newcastle, Newcastle upon Tyne, NE1 7ru, UK. 1996.

[10] 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.

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

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

[13] 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.

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

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

[16] 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.

[17] Alexandre Oliva and Luiz Eduardo Buzato. The Reflexive architecture of Guaraná. Technical report IC-98-14, Instituto de Computação, Universidade Estadual de Campinas, April 1998.

[18] ODMG: A Standard for Object-Oriented DBMSs. 1997

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

[20] Sun Microsystems Inc, Java™ Object Serialization Specification, November 1998.

[21] 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.

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

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

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