# Preservação da Thread Lógica nas Interações

Maria Alice Brito Carlos Sari Moreira da Silva Francisco Cavedon
{malice,csari,fcavedon}@ime.uerj.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 mechanisms that hold the logical thread in the interactions of applications, which are integrated to reflective middleware. The middleware elements that contribute to these mechanisms are the proxy and the Meta-Object. When the proxy intercepts a sending message, it gets the current thread, which identification will be useful so that Meta-Object finds the messages chain where the message will be linked. We remark that applications executed on this middleware are controlled by transactions. In addition, every message sent by application objets will be always associated to one of these transactions, which must be transparent in the message API. Therefore, another contribution from this work is to allow that Meta-Object knows which transaction is associated to message, mainly its concurrency control and recovery.*

**Resumo**

*Neste artigo, descrevemos os mecanismos necessários à preservação da thread lógica nas interações de aplicações acopladas a um middleware reflexivo. Os elementos do middleware que contribuem com esses mecanismos são o proxy e o Meta-Objeto. Quando o proxy intercepta uma mensagem de envio, obtém a thread corrente, cuja identificação será útil para que o Meta-Objeto descubra a cadeia de mensagens em que a mensagem deve ser concatenada. Observamos que as aplicações executadas sobre este middleware são controladas por transações. Além disso, qualquer mensagem enviada pelos objetos de uma aplicação estará sempre associada a uma dessas transações, que, por sua vez, deverão ser transparentes na API dessa mensagem. Assim, uma outra contribuição deste trabalho é a de permitir que o Meta-Objeto tome conhecimento de qual transação está associada à mensagem, principalmente no controle de concorrência e recuperação.*

## 1. Introdução

Este trabalho situa-se nos contextos de programação seqüencial, concorrente, paralela e distribuída, pelo ponto de vista das construções em linguagens de programação integradas a um *middlware*. O *middleware*, aqui referenciado, é responsável pela funcionalidade de aspectos de sistemas, tais como a persistência, a concorrência e a distribuição. Ele é implementado por uma Meta-Arquitetura, em que os seus Meta-Objetos realizam o elo do *middleware* com a aplicação. Enfocamos, particularmente, a preservação da *thread* durante a interação entre objetos, mesmo que as mensagens sejam tratadas, no nível do *Middleware*, por *threads* distintas.

O acesso remoto e o paralelismo das invocações permitindo que uma *thread* funcione entre duas ou mais bases remotas de sistemas distribuídos é mais um dos aspectos a ser suportado em interações, que surgem com aplicações integradas ao nosso *middleware*. Os autores, em [3], sugeriram *threads* mais leves, em que uma *thread t* de uma base inicia uma computação sobre uma base remota e uma *thread t´*, considerada uma delegada, é criada sobre essa base remota para tratar a computação. Nesse momento *t* fica suspensa até que o resultado de *t´* retorne. Essas duas *threads t* e *t*' possuem a mesma identidade.

Em uma outra solução, encontrada em Hyperion [5], a comunicação utiliza eventos e esse evento é uma abstração de uma mensagem e uma RPC. Um evento possui os seguintes campos: destino, tipo e dados específicos do próprio evento, o qual deve ser tratado no nó destino por um *event handler*. Os eventos são assíncronos por natureza e, assim, existe um mecanismo que provê assincronismo no *event handler*. A *thread* do lado do emissor deve

esperar até que um resultado seja retornado do alvo, para então continuar. Nesta abordagem é adotada memória compartilhada distribuída e há um compromisso de que a escrita de programas que usam paralelismo seja igual às demais escritas, sendo o suporte à execução paralela baseado em *threads*.

Pela nossa abordagem, o elo da *thread*, através da interação entre dois objetos da aplicação, é mantido com duas providências tomadas, em cooperação, pelo *proxy* e pelo Meta-Objeto emissor. A primeira dessas providências é tomada pelo *proxy*, quando ele intercepta uma mensagem de envio, dividida em quatro ações: identificar a *thread* corrente; criar a "meta-mensagem de envio me" associada à mensagem; guardar a identificação da *thread* nessa meta-mensagem me; e depositá-la na Caixa de mensagens do Meta-Objeto emissor. A segunda providência é tomada pelo Meta-Objeto emissor, quando ele trata uma "meta-mensagem de envio me", criando uma "meta-mensagem de recepção mr", que deve ser transmitida pelo *broker* (um outro elemento do *Middleware*) para o Meta-Objeto alvo. Nessa nova meta-mensagem mr criada é guardada a identificação da meta-mensagem me.

Os efeitos produzidos pelas duas providências acima vão ser úteis, mais tarde, quando a "meta-mensagem de retorno da enviada re" chegar ao Meta-Objeto do objeto emisssor. Nesse momento, o Meta-Objeto utiliza a identificação da meta-mensagem me guardada na meta-mensagem re para encontrá-la em suas estruturas e fazer o casamento dessas duas meta-mensagens. Em seguida, o Meta-Objeto pede a meta-mensagem me, para processar o retorno de alto nível. A meta-mensagem me, então, retorna ao *proxy*, que, por sua vez, providencia o retorno ao método do nível da aplicação, fazendo com que a *thread* original seja retomada.

Este trabalho é apresentado em mais duas seções: 2. Preservando a Thread na Propagação das Mensagens e 3. Conclusões.

## 2. Preservando a Thread na Propagação das Mensagens

Estamos considerando aplicações que são executadas sobre um *Middleware* implementado pelo nosso Modelo de Meta-Arquitetura, ver [1]. Em tais ambientes de execução, um objeto é constituído dos três objetos seguintes os quais não são visíveis pela aplicação: *proxy*, Caixa de Mensagens e Meta-Objeto. O elemento que consegue a transparência na sintaxe da interação é o *proxy*.

O *proxy* juntamente com o Meta-Objeto do objeto emissor e do objeto alvo tomam uma série de ações, começando com o proxy que intercepta um envio de mensagem que parte do nível da aplicação. Entre essas ações, neste trabalho, estamos interessados apenas naquelas que possibilitam que a *thread* seja preservada durante a propagação da mensagem pela árvore de composição. A responsabilidade nesses mecanismos que descrevemos neste trabalho é o de preservar as *threads*.

Todas as interações que surgem no nível da programação da aplicação são associadas a alguma das transações que surgem no ambiente de execução, podendo ser implícitas ou explícitas. A transação associada à mensagem também não é visível pela aplicação. A transação em muitos momentos coincide com a *thread*, mas em outros momentos, quando há invocações assíncronas, mais *threads* são geradas, e quando há comandos de transação nos métodos, novas transações locais aninhadas surgem. Essas circunstâncias fazem com que uma mensagem associada a uma transação pertença a uma *thread* derivada da *thread* original da transação. Assim, em alguns momentos nos referimos a transação, sendo ela a raiz de cada árvore da cadeia de interações. Mas a preservação da *thread* a beneficia, como podemos ver, mais a frente, no momento em que a transação que está associada a mensagem é descoberta.

Fazemos notar os seguintes aspectos no comportamento do envio de uma mensagem: 1) de um método pertencente ao objeto principal da aplicação, quando há um envio de mensagem, uma transação já estará controlando essa execução, embora seja desconhecida nesse contexto; 2) essa mensagem será interceptada pelo *proxy* que referencia o objeto alvo; 3) esse *proxy* prepara a meta-mensagem me correspondente a esse envio de mensagem, cuja maior responsabilidade é a de obter a identificação da *thread* corrente e agregá-la à essa meta-mensagem me; 4) quando a meta-mensagem me está pronta, ela é depositada na Caixa de mensagens do Meta-Objeto emissor ainda; 5) quando o Meta-Objeto do emissor for trata essa meta-mensagem me, uma de suas ações é buscar pelas cadeias de interações (uma de suas estruturas internas) a estrutura em que a meta-mensagem me deverá ser concatenada, podendo ser de dois tipos: uma meta-mensagem mr

associada ao método que enviou essa mensagem; ou um contexto de uma transação local aninhada que delimite o trecho de código do método de onde essa mensagem foi enviada. A chave para essa busca é aquela identificação da *thread* que foi salva pelo *proxy*, na meta-mensagem me; 6) mais tarde, quando a meta-mensagem de retorno re chega, no caso do ciclo da interação ser concluído, o Meta-Objeto emissor acessa a meta-mensagem me que vem informada na meta-mensagem re, para notificá-la desse retorno; 7) a meta-mensagem me sai do estado de espera e retorna ao *proxy*; 8) no *proxy*, um dos dois métodos pode ter sido o original, o que trata o modo síncrono ou o que trata o modo assíncrono: 7-a) se o modo da invocação foi síncrono, o resultado da invocação é desreificado e o retorno é efetuado; e 7-b) se o modo foi assíncrono, o resultado da invocação é desreificado e preenchido na variável futura.

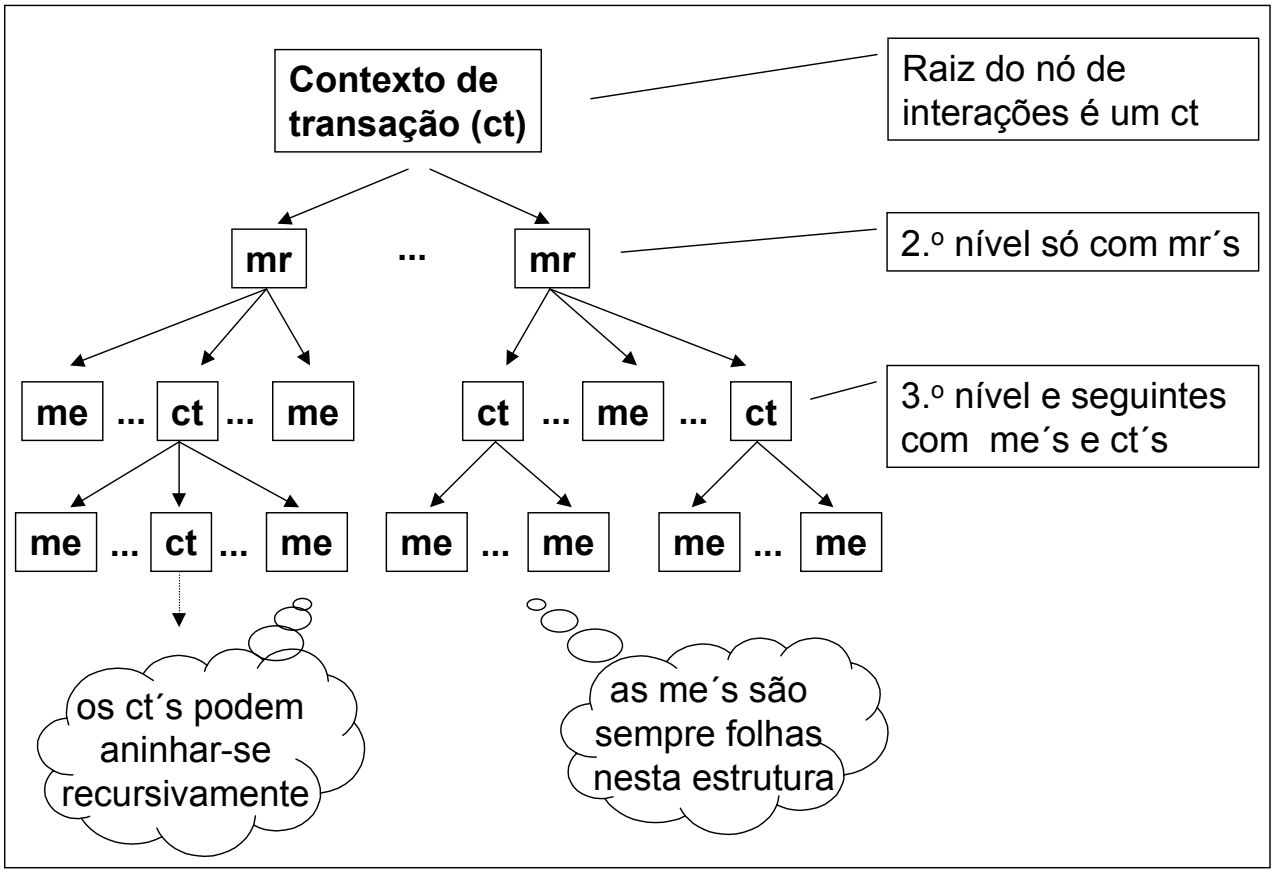


Figura 1: Diagrama de composição da estrutura do nó de interações, criada para cada transação cliente do objeto de aplicação, cujo ct é a raiz do nó.

Chamamos a atenção para o fato de que um método pode incluir ou não um ou mais comandos **`begin transaction`**, inclusive, recursivamente, aninhados. Para cada comando **`begin transaction`** é criada uma transação local aninhada, cuja ascendente de aninhamento pode ser a transação associada a meta-mensagem mr chegada ou uma outra transação local aninhada, e, assim por diante, em casos de aninhamentos recursivos em um mesmo método. Para que essa cadeia não seja perdida, a árvore da cadeia de interações deve possuir nós com estruturas de tipos correspondentes a contextos de transações locais aninhadas e a meta-mensagens mr, ver figura 1. Uma meta-mensagem me será sempre folha nesta árvore, porque ela será incapaz de aninhar alguma transação ou mensagem, não justificando a criação de uma nova *thread*.

É possível parecer estranho o surgimento de comandos do tipo **`begin transaction`** se estamos falando em transparência, inclusive de transações. Devemos observar que adicionamos esse comando, mas o programador tem a opção

de não incluí-lo no código de qualquer método de qualquer objeto pertencente a sua aplicação. Na ausência desse comando, a transação que vai controlar a aplicação é a inicial que surge no objeto *top-level* da aplicação, incluída, implicitamente, pelo *Middleware*. Nesse caso, não haverá qualquer estrutura de contexto de transação local aninhada nos nós das árvores de cadeia de interações relacionadas a essa aplicação.

Podemos imaginar a transação local aninhada como uma unidade de execução caracterizada como bloco, delimitando o código entre o **`begin transaction`** e o **`end transaction`**, que, assim, conduz à definição de uma nova instância de registro de ativação, pela *thread*. Esse bloco vai ganhar um tratador diferente, que, em outras palavras, significa uma nova *thread*.

Um Meta-Objeto pode possuir várias árvores do tipo cadeia de interações, aonde cada raiz corresponde à transação de mais alto nível associada a uma mensagem (meta-mensagem mr) chegada ao Meta-Objeto, quando esse está fazendo o seu outro papel, o de Meta-Objeto do objeto alvo dessa mensagem associada. Cada mensagem (meta-mensagem mr) que chega ao Meta-Objeto do objeto alvo ou é incluída em alguma dessas árvores já existentes no Meta-Objeto, cuja transação da raiz coincide com a sua transação associada ou, no caso de não existir árvore com transação na raiz que coincide com a essa transação associada, em uma nova árvore que deve ser criada, nesse caso.

A identificação da *thread* que o *proxy* salva na meta-mensagem me é a *thread* corrente (de Java™), obtida pela invocação da primitiva **`currentThread()`**. Essa *thread* é a mesma aonde o método da aplicação que enviou essa mensagem está executando e que foi criada antes para o tratador da meta-mensagem mr ou da transação aninhada.

No caso da mensagem ser enviada de um método que não possua comando de transação, a meta-mensagem me será incluída em um nó da árvore, cuja estrutura é do tipo meta-mensagem mr e, no caso de ser enviada de uma região de código delimitada por um bloco de um comando **`begin`** e **`end transaction`**, a meta-mensagem me será incluída em um nó da árvore, cuja estrutura é do tipo contexto de transação aninhada.

Podemos observar que essas medidas tomadas pelo *proxy* e pelo Meta-Objeto fazem com que uma *thread* lógica em um Meta-Objeto esteja sempre encadeada entre meta-mensagens mr, transações locais aninhadas e meta-mensagens me.

Vamos agora fazer um exercício sobre a etapa seguinte, quando o Meta-Objeto emissor constrói a meta-mensagem mr derivada da meta-mensagem me e providencia a sua transmissão para o objeto alvo, que significa depositá-la na Caixa de mensagens do Meta-Objeto do objeto alvo, com a cooperação do *broker* do *middleware*. Chamamos a atenção aqui para as seguintes informações carregadas pela meta-mensagem mr que ajudam a preservar o elo da transação através do ambiente de execução, quer seja local ou remoto. Uma dessas informações é o contexto da transação que está controlando essa mensagem enviada e a outra é a referência da meta-mensagem me, para que, no retorno, a meta-mensagem re possa lhe encontrar.

Na chegada ao objeto alvo, quer dizer, quando o seu Meta-Objeto for tratar a meta-mensagem mr, nós já sabemos, será procurada a árvore cuja raiz possua um contexto de transação igual ao carregado pela meta-mensagem mr, e, no caso de ainda não existir, será criada uma nova árvore. A meta-mensagem mr será então incluída nesta árvore como já vimos acima. Essa providência garante o elo da transação e da *thread* lógica entre dois objetos.

Após essa providência, o Meta-Objeto do objeto alvo submete a mensagem recém chegada ao controle de concorrência. Quando a mensagem é liberada pelo controle de concorrência, o Meta-Objeto providencia sua desreificação e invoca o seu método no objeto seqüencial. A partir daí, esse novo método será executado, com chances de novas mensagens serem enviadas, propagando-se os comportamentos descritos acima, repetidamente, através da composição dos objetos da aplicação.

Quando um método termina, ele retorna à unidade chamadora, que foi o tratador da meta-mensagem mr, a qual gera uma meta-mensagem rr, correspondente a mr, o resultado da invocação é reificado e preenchido na meta-mensagem rr. Essa meta-mensagem rr é colocada na Caixa de Mensagens do Meta-Objeto do próprio objeto alvo ainda.

Quando o Meta-Objeto for tratar a meta-mensagem rr, ele cria o seu tratador, cuja sua primeira ação, é fazer o casamento dessa meta-mensagem rr com a mr correspondente. Em

seguida, deve criar uma meta-mensagem re derivada da meta-mensagem rr e pedir para que o broker a transmita para o Meta-Objeto do objeto emissor, que significa colocá-la na sua Caixa de Mensagens. Neste ponto, a *thread* do tratador da meta-mensagem rr é encerrada. O contexto desse par de meta mensagens mr e rr já cumpriu o seu papel no que diz respeito ao ciclo da interação, porém ainda vai passar por situações de controle relacionadas ao controle de concorrência (atomicidade local), devendo então ser mantido no Meta-Objeto que tratou essas meta-mensagens.

Quando o Meta-Objeto do emissor for tratar a meta-mensagem re, ele cria um tratador para essa meta-mensagem, cuja primeira ação é combinar essa meta-mensagem re com a a meta-mensagem me correspondente. Em segundo lugar, dependendo do modo síncrono/assíncrono, que é detectado pelos parâmetros que foram reificados na meta-mensagem me, ações diferentes são tomadas, como podemos ver, a seguir.

No modo síncrono, é enviada uma mensagem **`avisaRetorno()`** à instância de me, cuja finalidade é notificá-la de que o retorno chegou, fazendo com que o método **`aguardaRetorno()`** saia do estado de espera e retorne ao método chamador do *proxy*, que desreifica o seu resultado e retorna ao objeto seqüencial do emissor.

No modo assíncrono, é enviada uma mensagem **`setFuture(reifiedResult)`** à estrutura da variável futura, cuja referência ficara guardada na estrutura da meta-mensagem me. Esse método desreifica o resultado, preenche-o na estrutura, troca o valor do *flag* para *cheio* e notifica essa condição à própria estrutura. Em algum momento, o método do objeto emissor pode estar interessado na variável futura, enviando a mensagem **`getFuture()`** à sua estrutura. Esse método fica em espera até que o *flag* da estrutura alcance o valor *cheio*, situação que já pode ter acontecido antes mesmo desse interesse ser manifestado.

Ao mesmo tempo, podemos observar a possibilidade de rastreamento de uma transação por cada árvore encontrada nos Meta-Objetos acessados no caminho da propagação de uma mensagem. Todos esses Meta-Objetos encontram-se também registrados como participantes da transação nos controles de sua gerência, em um objeto criado para transação de uma classe chamada Gerente de Transação. Esse gerente de transação toma conhecimento e registra cada participante, porque o Meta-Objeto, quando trata cada meta-mensagem do ciclo de interação (me, mer, rr e re), sempre a envia para o gerente de transação com a intenção desse gerente de transação participar do controle da transação. Assim, podemos percorrer os objetos participantes de uma transação por dois caminhos, ou pela lista de participantes na gerência da transação, ou pela árvore de cadeia de interações de cada Meta-Objeto, cujos nós tem o elo para o próximo objeto de aplicação participante. A lista de objetos participantes da gerência de transação vai colaborar com a execução das fases de uma transação, como por exemplo, aguardar a propagação de todas as mensagens para concluir que uma fase da transação foi concluída e poder passar à fase seguinte, como, por exemplo, a de bloqueio para a de execução, em uma transação two-phase locking ou a de execução para a de consistência em uma transação otimista, entre outras fases.

## 3. Conclusões

A árvore no Meta-Objeto faz o elo entre as chamadoras e as chamadas, substituindo a pilha de registros de ativação da computação seqüencial. Nessa cadeia, algumas *threads* a mais que as transações vão surgir, por causa da possibilidade de invocações assíncronas e surgimento de transações aninhadas. Mas todas as cadeias serão preservadas, sendo os retornos de rr e de re combinados as suas mensagens de mr e de me, respectivamente. As cadeias de transações aninhadas também retomarão as cadeias de suas estruturas ascendentes, sendo uma outra transação aninhada ou a mr do método de onde ela tenha surgido. Todos esses elos são resolvidos, localmente, no Meta-Objeto, pelas estruturas das árvores e entre Meta-Objetos, pelas identificações das me's que seguem nas mr's, permitindo que nos retornos das rr's, as re's sejam transmitidas para os Meta-Objetos emissores dessas mensagens, corretamente.

## Referências

[1] Brito, M., Menezes, A., Hess, D., Almeida, É. and Freitas, M. (2003) "Semântica das Mensagens que Constituem as Interfaces dos Três Principais Elementos da Meta Arquitetura Alana", Cadernos do IME – Série Informática, Vol. 14, julho.

[2] Eugster, P. and Baehni, S. (2002) "Abstracting Remote Object Interaction in a Peer-2-Peer Environment", In Proceedings of the ACM 2002 Java Grande Conference, November.

[3] Hicks, M., Jaganhattan, S., Kelsey, R., Moore, J. and Ungureanu, C. (1999) "Transparent Communication for Distributed Objects in Java", In: Proceedings of the ACM 1999 Java Grande Conference, June.

[4] Izatt, M., Chan, P. and Brecht, T. (1999) "Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications", In: Proceedings of the ACM 1999 Java Grande Conference, June.

[5] MacBeth, M., McGuigan, K. and Hatcher, P. (1998) "Executing Java Threads in Parallel in a Distributed-Memory Environment", In: Proceedings CASCON'98. Published by IBM Canada and the National Research Council of Canada.

[6] Skilicorn, D. and Talia, D. (1998) "Models and Languages for Parallel Computation", ACM Computing Surveys, Vol. 30, No. 2, June.