Implementação de Mecanismos de Interação via Java Threads

Maria Alice Brito Rafael Bernardo Teixeira
{malice, rafaelb}@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 the language constructs and features context, in particular, concurrent programming structures, we present in this paper an interaction mechanism implementation relies on integrated concurrency model, so-called active object. This interaction assumes a kind of asynchronous RPC with use of future, in which the result of a local or remote invocation will be accessed when needed. We point out the following main topics in this implementation: 1) the concurrency integrated model was solved by the use of a Meta-Object to be created and bound for each application object in run-time; 2) the asynchronous invocation was solved by the use of proxies implemented as threads, in order to provide independent behavior in relation to their Meta-Object; 3) a kind of Message Queuing approach was implemented, in which invocation and response messages can be stored in a message box, to make possible the access to future result variable, when needed; and 4) the invocation and response messages were turned into meta messages, in accordance with Active Messages approach, in which decouples communication and synchronization by treating messages as active objects rather than passive data. We make note that Java supports shared-memory communication via threads and monitor features, but we have used these features for solve our concurrency model. The three concurrency aspects – the model, the concurrency and the interactions – hold an expressive synergy, though we have tried put focus in the interaction aspect, we could not avoid the another two aspects arose sometimes.

1. Introdução

Este trabalho está situado no contexto de linguagens de programação orientada a objeto concorrente, em particular, em controle de concorrência, com enfoque nos mecanismos de interação entre objetos.

Nossa motivação surgiu do interesse pelas questões desse contexto e pelas necessidades do modelo da Meta-Arquitetura Alana. A proposta principal desse modelo de Meta-Arquitetura, que pode ser caracterizado como um *Middleware*, é a de permitir a integração das facilidades de concorrência, transação e persistência a uma linguagem de programação orientada a objeto. Essa integração capacita a implementação de aplicações que necessitem estar controladas por transações, cujos tipos podem variar desde as mais tradicionais como as de protocolo *two-phase locking* a outras mais sofisticadas, como por exemplo, as desenvolvidas para controle de *workflow*. Além disso, as transações que compartilham os objetos de aplicação podem ser, simultaneamente, de políticas diferentes. Essa Meta-Arquitetura é constituída de três classes principais: gerente de objetos e transações (*broaker*), gerente de transação e objeto de aplicação, sendo que cada instância dessa última classe é associada a uma instância de uma classe Meta-Objeto. A implementação dessa Meta-Arquitetura foi dividida em vários subprojetos de implementação de acordo com seus módulos. Para a implementação dos mecanismos de interação aqui abordados, foram exigidos esforços de investigação de soluções em linguagens de programação, muitas vezes sem recursos para os mecanismos envolvidos.

A implementação ficou localizada no módulo Meta-Objeto da Meta-Arquitetura, em particular, nos mecanismos de interação do controle de concorrência.

O Meta-Objeto é o módulo encarregado de resolver as facilidades de concorrência e de recuperação para o objeto de aplicação, além de cooperar com a gerência da transação no nível global e ainda participar da facilidade de persistência. Ele coopera com o seu objeto seqüencial, concentrando-se, principalmente, nas tarefas de tratamento de mensagens trocadas durante a interação com outros objetos de aplicação. Assim, entre as meta mensagens, tratadas neste trabalho, foram enfocadas somente as de envio do objeto seqüencial para algum outro objeto de aplicação, as recebidas de algum outro objeto de aplicação para o seu objeto seqüencial, e as de retorno, constituindo as quatro meta mensagens envolvidas em um ciclo de interação entre

dois objetos de aplicação.

Podemos resumir o objetivo deste trabalho como uma implementação dos aspectos de interação nos mecanismos de concorrência, que são resolvidos pelo Meta-Objeto.

A modelagem dos mecanismos de controle de concorrência e recuperação, na Meta-Arquitetura Alana, foi conduzida pelo alcance de uma transparência no código dos métodos dos objetos de aplicação, reforçando o encapsulamento e evitando a anomalia de herança. Em outras palavras, as aplicações que necessitassem de facilidades de concorrência, transação, persistência e distribuição não precisariam ter, em suas classes, referências a tais aspectos.

Esse compromisso vinha sendo perseguido pela comunidade que pesquisava persistência ortogonal, passando também a transação ortogonal [5], fazendo uma concessão somente na criação dos objetos de aplicação, cujos construtores poderiam permitir parâmetros, como por exemplo, o endereço da máquina remota.

Para entender o alcance dessas expectativas, achamos apropriado incluir neste trabalho uma discussão sobre as várias abordagens em controle de concorrência adotadas nas linguagens de programação orientada a objeto, e as justificativas para as escolhas feitas para a meta arquitetura Alana, que foram levadas em conta neste projeto de implementação.

Podemos sintetizar as características da interação implementada neste projeto como: uma troca de mensagem com as opções com/sem espera e/ou com/sem resultado de retorno oferecidos pela linguagem de programação orientada a objeto. Essas opções, no código do envio de mensagem a um objeto de aplicação, podem variar desde a mais simples, no modo síncrono e sem resultado no retorno (comportando-se como uma chamada de procedimento, na programação seqüencial), à mais complexa, no modo assíncrono e com resultado no retorno.

É importante notar que durante a implementação foi perseguida a independência entre o mecanismo de interação e os outros mecanismos de controle de concorrência. Mas como existe uma sinergia entre os três aspectos envolvidos: modelo de concorrência; interação; e concorrência interna, ver seção 2, não foi possível evitar a presença desses dois outros aspectos em vários pontos da implementação da interação, como podemos observar, na seção 3.

A principal contribuição deste trabalho é a implementação em si do aspecto de interação, pertencente ao controle de concorrência, mas é interessante citarmos mais duas contribuições importantes que surgiram com esse projeto. Como para a implementação foram utilizados recursos de reflexão e de concorrência oferecidos em Java™, uma dessas contribuições é um exercício didático com o uso desses recursos, principalmente o uso de *thread*s, que não é considerado tão simples, na atividade de programação. A outra faz parte do próprio compromisso do modelo Alana, que é a de tirar da incumbência de um programador Java essas questões de reflexão e de concorrência, que ficam resolvidas no meta objeto que é transparente à programação.

Este trabalho é apresentado nas três seções seguintes: 2. Fundamentos e trabalhos relacionados, em que são discutidos os aspectos conceituais de linguagens concorrentes, que foram levados em conta na meta arquitetura Alana e na solução da interação entre objetos de aplicação (troca de mensagem assíncrona ou síncrona); 3. Modelo de computação, em que descrevemos a implementação dos mecanismos; e, finalmente, a seção 4. Conclusões.

2. Fundamentos e trabalhos relacionados

Para assegurar a exclusão mútua (sincronização de competição) no acesso a dados compartilhados, encontramos três abordagens mais antigas em linguagens de programação: semáforos [6], monitores [9, 11, 24] e troca de mensagens [10, 12] (em Ada 83, as tarefas implementam a troca de mensagens síncronas). Mais recentemente, em Ada 95, foram implementadas as trocas de mensagens assíncronas e, em Java 96, foi oferecida a facilidade de **`thread`**, que pode funcionar em combinação com a cláusula **`synchronized`** para garantir a exclusão mútua do método, ou até mesmo de um trecho de código. O uso das facilidades de Java será melhor explicado no curso da apresentação do nosso modelo de computação, na seção 3.

Para a sincronização de cooperação, que assegura que o compartilhamento será semanticamente correto, costuma-se usar semáforos. Por isso, encontramos o uso de semáforos, ainda nas outras soluções de concorrência, como, por exemplo, nos monitores. Em Java, as facilidades **`wait`** e **`notify`** em combinação com **`synchronized`** resolvem essa sincronização de cooperação, que poderemos ver também na seção 3. No entanto, fazemos notar que esses instrumentos não são os mais importantes para a sincronização no nosso modelo, porque sua sincronização de cooperação explora oportunidades de semântica dos objetos de aplicação, resolvida com o conceito de atomicidade local no controle de concorrência dos objetos de aplicação e a sincronização de competição pode ficar relaxada com alguma negociação com a sincronização de cooperação.

Nas linguagens de programação orientada a objeto, o controle de concorrência, sobre objetos compartilhados, simultaneamente, por mais de uma aplicação, fica resolvido por diversas estratégias, que, em geral, combinam três aspectos principais [22, 23, 3, 4, 17]: 1) modelo de objetos; 2) construções para interações entre objetos; e 3) a concorrência interna,

como podemos ver nas seções seguintes.

Os critérios tradicionalmente adotados na sincronização de cooperação concentram-se na invalidação de uma leitura por uma escrita. A granulosidade do que foi escrito e lido também pode variar do estado inteiro de um objeto para cada operação sobre um objeto, sendo que quanto mais fina for a granulosidade mais aumentam as chances de concorrência, quer dizer, diminuem as chances de conflito.

No controle de concorrência, em orientação a objeto, a sincronização de cooperação é dos três aspectos, o mais difícil a ser tratado. Esta questão é praticamente vinculada a interação entre objetos, porém, por causa das dificuldades envolvidas em seus mecanismos, vamos abordá-la, separadamente, provavelmente repetindo aspectos já mostrados em interação.

2.1. Modelo de objetos

Existem duas estratégias para suportar concorrência em programação orientada a objeto:

ortogonal – os *threads* de controle são independentes dos limites de encapsulamento dos objetos como por exemplo em ConcurrentSmalltalk [26] e Trellis/Owl [18];

integrada – os *threads* são de alguma forma apropriados pelos objetos, e seus efeitos além dos limites do objeto são totalmente controlados, como podemos ver nas abordagens seguintes:

pode ser gerado um novo *thread* para processar cada mensagem, tornando o modo de processamento de cada mensagem assíncrono, como, por exemplo em *Actor languages* [1];

o mecanismo de concorrência da linguagem, como por exemlo em *Eiffel* //, pode alocar um *thread* a um conjunto de objetos e todas as operações então executarem neste *thread* alocado em vez de no *thread* do *sender*. No caso mais simples, cada objeto tem seu próprio processo. Quando um processo aloca um objeto e este objeto (cliente) envia mensagem para outro objeto (servidor), duas situações podem surgir:

se a mensagem espera resultado imediato, o *thread* do objeto cliente é bloqueado até o retorno;

se nenhum resultado imediato é requerido, o *thread* do objeto cliente pode continuar e a mensagem será processada pelo *thread* do outro objeto (servidor) , e quando chegar o momento do objeto cliente precisar do resultado, o *thread* do objeto cliente ficará a espera.

o conceito de *objeto ativo* é usado, sendo considerado por vários autores [23, 3, 4, 17] como o modelo ideal de programação. Ele 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. Este modelo de programação assinala que os próprios objetos servidores têm a responsabilidade de responder aos pedidos concorrentes que são recebidos e enfileirados, provendo uma forma de *message passing* assíncrona, como, por exemplo, em *DRAGOON* [3], entre outras linguagens.

Cada uma dessas formas *integradas* de concorrência apresenta algum benefício e são muito superiores à abordagem *ortogonal*. Além disso, o conceito de *objetos ativos* favorece a combinação de simplicidade e flexibilidade, ganhando a preferência de muitos autores.

2.2. Construções para interações entre objetos

Inicialmente, os projetistas de linguagens de programação e de sistemas operacionais seguiram estritamente as abordagens, como classificadas a seguir, em [3]:

tightly coupled – uma parte da memória primária é compartilhada e todos os processadores tem acesso direto a esta memória por instrução de máquina. Esta abordagem oferece vantagem na velocidade de comunicação por causa da memória compartilhada (talvez o paradigma mais antigo para programação paralela).

loosely coupled (em sistemas distribuídos) — os processadores têm acesso somente a sua própria memória, assim, processadores podem comunicar-se pela troca de mensagens por um canal de comunicação, tal como um *link* ponto a ponto ou por uma *lan* (*local area network*). Esta abordagem oferece vantagens na facilidade da construção de seus mecanismos, especialmente, se um grande número de processadores vai participar.

Estas linhas geraram dois paradigmas de programação paralela: *variáveis compartilhadas* (para *tightly coupled*) e *message passing* (para sistemas distribuídos).

Variáveis compartilhadas Muitos sistemas operacionais para monoprocessadores são estruturados como coleções de processos, que executam em quasi-paralelo, e comunicam-se através de variáveis compartilhadas. Estes processos devem conhecer os endereços das variáveis compartilhadas e, assim, uma atribuição a tais variáveis produz um efeito imediato, tornando as execucões destes processos mais rápidas do que aquelas, em processos que utilizam *message passing*. A sincronização de acesso a variável compartilhada se tornou outro tópico de pesquisa. Numerosas linguagens de programação usam variáveis compartilhadas. A semântica deste modelo é muito simples, complicando-se, quando mais de um processo tenta escrever ou escrever e ler a mesma variável. Como as variáveis compartilhadas podem ser alteradas por qualquer processo, a segurança se torna um problema maior. Os efeitos de operações de escrita simultâneas podem deixar as variáveis indefinidas ou pode ser adotado o uso de monitores ABCL/1 [25], Orient84 [21], Java [15] que encapsulam os dados e serializam todas as operações sobre os dados.

Os dois tipos de sincronização podem ser empregados em variáveis compartilhadas e o de cooperação utiliza semáforos, *eventcounts* ou *condition*

variables., como por exemplo em Java.

A maioria das primitivas que contam com memória compartilhada utiliza replicação de dados para uma implementação mais eficiente.

Troca de mensagem A base do paradigma troca de mensagem, como uma construção para linguagem de programação, se encontra em um artigo clássico de Hoare sobre CSP [12], como já citamos, no início. Uma mensagem em CSP é enviada de um processo (o *sender*) para um outro processo (o *receiver*) e o *sender* espera até que o *receiver* tenha aceito a mensagem (*synchronous message passing*), sincronizando os processos que estejam interagindo. As mensagens passadas em *message passing* podem ser feitas no modo síncrono ou assíncrono. Com *message passing*, um intervalo de tempo transcorre desde o envio até a chegada da mensagem ao destino, e a ordem das mensagens entre um par de processos se mantém entre o envio e a chegada, sendo observada a semântica de *order-preserving*. No caso de mais de dois processos, o intervalo de tempo tem que ser levado em conta, porque não há garantia de que as mensagens cheguem na ordem correta. Por exemplo, um *receiver*, que esteja recebendo de dois *senders*, vai precisar saber qual a mensagem que foi emitida primeiramente. Estes controles tornam a abordagem com *message passing* mais segura do que a com variável compartilhada, impedindo que um processo afete a integridade de outro. Outras variações de *message passing* foram propostas, como, por exemplo, a seguir:

Assyncronous message passing, em que o *sender* continua imediatamente após o envio da mensagem Hibrid [16] e Actor languages [1], entre outras ;

Remote procedure call (*synchronous message passing*) e *rendez vous* são duas interações *two-way* entre dois processos, como, por exemplo, ABCL1 [25] que usa *RPC* com *future*;

Assyncronous message passing e *synchronous remote procedure call*, essas duas formas são oferecidas , por exemplo em POOL [2] e Concurrent Smalltalk [26]

Broadcast e *multicast* são interações entre um *sender* e muitos *receivers*.

Recursos, tais como *portas de comunicação* e *mailboxes*, como por exemplo, *cBox* em *ConcurrentSmalltalk*, também são usados para evitar endereçamento explícito de processos.

Dificuldades, como a passagem de estruturas complexas a um processo remoto e processos que não podem migrar para outros processadores, dificultando a tarefa de tornar um processo mais eficiente, devem ser levadas em conta, com o uso deste paradigma de interação.

Algumas linguagens e sistemas operacionais para mono processadores ou multiprocessadores com memória compartilhada suportam processos que se comunicam por *message passing*. Mais recentemente, a abordagem dupla, quer dizer, a aplicação do paradigma de *memória compartilhada* a *sistemas distribuídos*, tem se tornado um tópico popular na pesquisa [17]. A princípio, esta abordagem dupla pode parecer contrária ao seu objetivo inicial, em que o paradigma *message passing* combina-se melhor com as primitivas providas pelo *hardware distribuído*. Para as linguagens sequenciais, entretanto, esta abordagem dupla se torna útil, como nos paradigmas de programação: funcional, lógico, e de orientação a objeto, que não referenciam diretamente a arquitetura de hardware. Assim, muitas propostas não adotam rigorosamente *memória compartilhada* e *message passing*, mas uma mistura entre estas duas abordagens. Quer dizer, existe uma faixa que varia de *memória compartilhada* a *message passing*.

Envio/recepção de mensagens/retornos Nas construções para interações entre objetos, uma das principais questões é como são tratados o envio e a recepção de mensagens e de retornos. A escolha desta construção de interação na concorrência caracteriza o poder expressivo provido para a implementação de objetos concorrentes.

Recepção de respostas Em *message passing*, a comunicação pode ser síncrona, como por exemplo em CSPL [12], ou assíncrona, como por exemplo nas linguagens do modelo actor, ou mista, quer dizer, o programador pode fazer a opção pelo modo síncrono, ou assíncrono, ou assíncrono com uso de future (por exemplo, ABCL1 [25]). Neste tipo de comunicação (*message passing*) os clientes têm liberdade de intercalar atividades enquanto existirem pedidos pendentes. As respostas podem ser dirigidas a endereços escolhidos se a liberação do pedido pode ser explicitamente programada. A principal dificuldade com *message-passing* é a obtenção das respostas, necessitando de cooperação entre o cliente e o servidor para combinar as respostas com os pedidos, como, por exemplo, ser incluído, no pedido, o endereço de destino para a resposta. Em *remote procedure call* (RPC), o *thread* do cliente é bloqueado até que o servidor aceite o pedido, realize o serviço pedido e retorne uma resposta, assim, embora seja trivial obter uma resposta, não é possível intercalar atividades ou especificar endereços de retorno.

Uma outra alternativa para sincronizar é pela introdução de um *proxy*. A idéia principal é delegar a responsabilidade de liberação do pedido e obtenção da resposta a um *proxy*, que é um objeto independente (não precisando ser de primeira classe), deixando o cliente livre para qualquer tarefa. Esta abordagem mantém os benefícios de uma interface *RPC* e a flexibilidade da *message-passing*, com as vantagens de facilitar a combinação de pedidos e respostas. Na linguagem ABCL/1 [25] é feita uma combinação de *RPC* com *message-passing*, permitindo a um objeto possuir uma interface externa *RPC* e usar primitivas de baixo nível *message-passing* para responder por envio de uma mensagem assíncrona ao cliente ou ao seu *proxy*.

A possibilidade da representação de pedido/resposta,

como objeto de primeira classe, permite que o nome do método e o endereço de retorno sejam especificados, dinamicamente. Esta abordagem é adotada, como, por exemplo, nas linguagens Hibrid [16], POOL-T [2] e Trellis-Owl [18]. A linguagem CSP [12], que é mais antiga, permitia estas especificações estaticamente, obrigando que o servidor ficasse amarrado a estas especificações e dificultando o reuso.

Recepção de pedidos Quando um pedido chega ao servidor, este pode estar ocupado, atendendo a outros pedidos e/ou mesmo esperando por respostas de pedidos que ele próprio tenha invocado. A aceitação destes pedidos pode ser incondicional, explícita ou condicional.

Na *aceitação incondicional*, os pedidos podem ser respondidos, seguindo apenas o controle de concorrência, como, por exemplo, o da exclusão mútua.

Na *aceitação explícita*, os pedidos podem ser respondidos por meio de um comando *accept* explícito, executado no servidor, de acordo com o conteúdo (nome da operação, argumentos, etc.) ou com o estado do objeto. As linguagens, como por exemplo, Ada [3], ABCL/1 [25], POOL-T [2], adotam esta abordagem de aceitação explícita, e seus objetos são *single threads*.

Na *aceitação condicional*, os pedidos são aceitos de acordo com um predicado sobre o estado do objeto e/ou conteúdos das mensagens, como, por exemplo, em ACT++[1]. Ainda, em aceitação condicional, em [17], o autor distingue alguns mecanismos diferentes, como a seguir:

Aceitação condicional com mecanismo de sincronizador é um objeto especial associado a um grupo de objetos. Quando um método de qualquer destes objetos é invocado, uma condição neste sincronizador é avaliada e, dependendo do resultado, a execução do método é permitida ou retardada.

Aceitação condicional com mecanismo de argumentos de métodos separados, que podem ser usados para restringir a execução de um método por pré-condições sobre os argumentos declarados como "separados". A execução do método, assim, fica atrasada até que as pré-condições sejam verdadeiras e os objetos separados ficam reservados durante a chamada, para serem usados no corpo do método.

Aceitação condicional com mecanismo de notificadores de predicado de estado, que podem restringir a execução de um método, como o próprio nome diz, pela notificação vinda de um outro objeto que tenha alcançado um estado que satisfaça um predicado. Esta facilidade assume formas síncronas e assíncronas.

Aceitação condicional com mecanismo de computação reflexiva, esta abordagem faz com que na chegada de um pedido seja disparado um método do meta-objeto do servidor. O meta-objeto então trata mensagens do nível do objeto seqüencial e mailbox como objetos. Além disso, as mensagens enviadas para um objeto podem ser interceptadas, para que sua execução seja sincronizada, simulando linguagens baseadas no modelo Actor.

2.3. Concorrência interna

A concorrência de uma linguagem de programação orientada a objeto concorrente pode ser enquadrada, nas classes, a seguir, de acordo como o objeto é internamente:

Seqüencial Nesta classe, as linguagens possuem um *single thread* de controle, como por exemplo, ABCL/1[25], POOL [2], Ada tasks [3];

Quase-concorrente Nesta classe de linguagens, os objetos possuem múltiplos *threads*, mas só um pode estar ativo de cada vez. O controle pode ser explicitamente liberado para permitir intercalação de *threads*, como por exemplo, nas linguagens Hibrid [16] e em monitores;

Concorrente Nesta classe, as linguagens são consideradas completamente concorrentes, o número de *threads* internos não é restringido, porém a liberdade para a criação de novos *threads* pode variar, entre a criação livre de *thread*s, no momento da aceitação dos pedidos para objetos considerados de primeira classe; a criação de *thread*s, ignorando-se as restrições de sincronismo dos métodos normais nas linguagens, e até mesmo uma criação mais restritiva de *thread*s.

2.4. Influências entre os três aspectos: modelo de objetos, interações e concorrência interna

A concorrência interna está vinculada ao poder expressivo que é provido a objetos para o tratamento das mensagens (pedidos) e com a proteção do estado do objeto, que, por sua vez, está vinculada ao modelo do objeto.

A combinação dos três aspectos acima não é simples e podemos adiantar que para nós o modelo ideal de programação se apresenta como objetos que se comportam cooperativamente, trocando mensagens. As operações sobre um objeto podem ser executadas concorrentemente. Para este modelo de programação, na literatura, é empregado o termo *objeto ativo*, para assinalar que os próprios objetos tenham a responsabilidade de responder aos pedidos concorrentes. As mensagens deveriam ser respondidas consistentemente com o estado interno do objeto e a sua possibilidade de execução, respeitando a sincronização de competição e de cooperação. Deve ser assegurada a possibilidade de herança das classes destes objetos, inclusive para refinamento das tarefas de atendimento às mensagens. Além disso, as aplicações não devem ser

[1] *Apud* – D. G. Kafura. Concurrent Object-Oriented Real-Time Systems. Technical Report, TR88-47, Dept. of Computer Science, Virginia Tech. [Atk91]

afetadas por essas trocas.

A princípio, qualquer linguagem que combine concorrência e facilidades de orientação a objeto pode ser usada para desenvolver software de acordo com o modelo acima. No entanto, algumas estratégias não corresponderam a expectativas que deveriam proporcionar o desenvolvimento de objetos autocontidos com possibilidades de reuso. Para isso, *as classes dos objetos reusáveis deveriam fazer o mínimo de considerações em relação aos clientes de suas instâncias* e alguns requisitos sobre as facilidades de linguagens deveriam ser desenvolvidos para suportar a programação seguindo um modelo de objeto ativo. Estes requisitos foram enumerados por Michael Papathomas em [17], como podemos ver, a seguir: a) exclusão mútua; b) transparência no despacho dos pedidos; c) concorrência interna (por exemplo, *threads* concorrentes no interior do objeto); d) transparência no retorno de pedidos. Além disso, os três aspectos – 1) modelo de objetos; 2) construções para interações entre objetos; e 3) concorrência interna – devem ser escolhidos observando esses quatro requisitos acima.

O aspecto (1) modelo de objetos deve ser integrado para assegurar a sincronização de condição, a serialização e o reuso da programação de concorrência. Os outros dois aspectos (2) construções para interações e (3) concorrência interna são entrelaçados e, dessa forma os abordamos conjuntamente, a seguir.

2.5. Mecanismos de interação com objetos nos diversos modelos de concorrência

Os objetos devem contar com a possibilidade de um cliente concorrente emitir vários pedidos antes de obter uma resposta. Assim, é importante saber qual resposta corresponde a qual pedido.

Em objetos sequenciais Na abordagem *message-passing* ou *RPC*, o cliente necessita saber qual o pedido que deve combinar com a resposta. Ainda em *message-passing*, no modo síncrono, ou em *RPC*, uma dificuldade adicional é que o cliente é bloqueado até que a resposta chegue.

O uso de *proxy built-in* (*proxies* definidas pelo programador são consideradas desajeitadas para a programação) e outras aternativas semelhantes, como, variáveis futuras em ABCL/1 ou CBox em Concurrent Smalltalk poderiam ser usadas para enviar mensagens aos objetos servidores, sem provocar bloqueio no objeto cliente e combinando mais tarde com o retorno. Uma dificuldade com *proxy built-in* é que o cliente pode em algum momento precisar de: obter bloqueio, ou esperar por algum outro pedido, ou esperar por algum retorno. Todas essas possibilidades, sem um mecanismo de sincronização, após o bloqueio, podem impedir o objeto cliente de receber pedido ou retorno. Assim, o mecanismo de *proxy built-in* combinado a *single thread* ainda é considerado um suporte limitado.

Ainda em objetos seqüenciais, a combinação de pedido/retorno (RPC) com *message-passing*, usada, por exemplo, em ABCL/1, torna possível relaxar o estilo RPC e suportar *message passing* como a principal primitiva de comunicação. Mas continua sendo preciso algum esforço para casar o retorno com o pedido, bem como o de administração de filas.

Em objetos quase-concorrentes Como em monitores, este estilo faz com que a concorrência fique restrita por causa do suporte limitado ao envio das respostas. Certas linguagens de programação orientada a objeto concorrentes tornaram os monitores mais flexíveis, por exemplo, Emerald [13] e Java [15] usam monitores definidos por Hoare [11], entretanto, nem todas as operações de um objeto têm que ser declaradas como *procedures* monitoras e também vários monitores independentes podem ser usados na implementação do objeto. Trellis/Owl [18] utiliza *lock blocks* e filas de espera, permitindo esquemas mais flexíveis do que se objetos fossem identificados como monitores. Hibrid [16] utiliza um mecanismo *delegated call*, que provê uma abordagem de envio de resposta mais flexível, não restringindo a concorrência.

Em objetos quase-concorrentes Com *threads* concorrentes, o atendimento de vários pedidos de clientes concorrentes é direto pela criação de um novo *thread* para o processamento de cada pedido de cliente. Durante a execução de um *thread*, no objeto, este mesmo *thread* não poderia disparar a criação de outro *thread*, porque ele seria bloqueado. As linguagens que adotam *threads* concorrentes provêm suporte adequado ao despacho de pedidos e à concorrência interna. Porém, se a iniciação de múltiplos *threads* no objeto não for possível, como observamos acima, o tratamento da resposta ainda não fica resolvido só com este mecanismo.

2.6. Considerações

As soluções alternativas, em linguagens concorrentes, apresentadas nesta seção, foram discutidas e aquelas que permitiram a combinação com os recursos de abstração e herança do paradigma de orientação a objeto, em especial, o encapsulamento e com o cuidado de viabilizar a flexibilidade para admitir a combinação das soluções de transação foram consideradas as mais apropriadas à meta-arquitetura Alana.

A abordagem de *integração* para o modelo de objeto concorrente favorece o encapsulamento do objeto, porque o objeto se apropria dos *threads* enquanto que no modelo ortogonal os *threads* de controle são independentes dos limites do encapsulamento. Com essa opção, o conceito de objeto ativo é considerado ideal para a programação, podendo cada objeto se comportar cooperativamente, com o auxílio de um meta-objeto, trocando mensagens, que são respondidas por ele próprio, provendo uma forma de *message passing*. As transações deverão compartilhar os objetos,

e, a cada mensagem vinculada a uma transação diferente que chegue ao objeto, deve ser criado um novo *thread* pelo meta-objeto.

Este trabalho ficou restrito aos *mecanismos de interação com objetos*, que é um dos três aspectos de controle de concorrência, como apresentamos acima. Para esses mecanismos, vemos algumas soluções em linguagens de programação orientada a objeto concorrentes que combinadas podem atender às necessidades de representação de pedido/resposta e ao mesmo tempo colaborar no mecanismo *de message-passing* assíncrono, como a seguir.

A possibilidade da representação de pedido/resposta, como um objeto de primeira classe, que permite que o nome do método e o endereço de retorno sejam especificados, dinamicamente, como a solução adotada em Hibrid, Pool-T e Trellis-Owl. Encontramos em [20] a definição de "*active messages*" para a abordagem de mensagens como objetos em vez de dados passivos, usada, por exemplo, em Movie, J-machine, JOPI [14] e Obevents [8].

Podemos adicionar a esse objeto de primeira classe os papéis desempenhados por um *proxy*, tais como os de liberar o pedido e de obter a resposta, como um objeto independente, deixando o objeto cliente da mensagem livre para qualquer tarefa.

O nosso meta objeto que pode ser modelado assumindo a forma de um processo centralizado limitado e amarrado a cada objeto da aplicação, tornando o objeto da aplicação um objeto ativo, poderia também dispor dos serviços de uma caixa de mensagens, que realiza a única tarefa de receber as mensagens que chegam para o objeto de aplicação, armazenando-as numa fila e permitindo que o emissor fique desacoplado do receptor [8].

O modelo caracterizado pela aceitação incondicional seguindo apenas as regras de concorrência, em recepção de pedidos, foi considerado apropriado para a meta arquitetura Alana, por causa da capacidade de responder às mensagens de acordo com as condições de sincronização.

Essas soluções conseguiram, ao mesmo tempo, superar a dificuldade de casamento entre pedido e retorno e colaborar no mecanismo de *message-passing* assíncrono, como poderemos ver na seção seguinte, que descreve os mecanismos para a solução da interação entre objetos concorrentes, em outras palavras, a troca de mensagem assíncrona.

3. Modelo de computação

Para a implementação do aspecto de interação do controle de concorrência, os outros dois aspectos: modelo de concorrência (integrado) e concorrência interna (concorrente) estão envolvidos, conforme as expectativas abordadas nas considerações da seção 2. Este trabalho foi conduzido pelos requisitos ditados no final do anterior acompanhado de um cuidado com o encapsulamento do objeto de aplicação. O encapsulamento também é adotado na implementação dos mecanismos para o controle de concorrência.

Esses mecanismos foram localizados no Meta-Objeto da Meta-Arquitetura Alana. Essa Meta-Arquitetura constitui-se de três elementos principais, programados em classes. Esses elementos são os seguintes: 1) o gerente de objetos e de transações (GOT); 2) o gerente de transação (GT), sendo criada uma instância para cada transação que surge no ambiente de execução; e 3) o Meta-Objeto, cuja instância é criada e associada a cada seleção de um objeto de aplicação, em tempo de execução. Esses três elementos da Meta-Arquitetura cooperam e desempenham os seus papéis, nos aspectos que constituem o funcionamento de uma transação. Um conjunto de meta mensagens constitui a interface desses elementos. Essas meta mensagens são mensagens ativas de acordo com a definição em [20]. Todas as trocas de mensagens entre os elementos da Meta-Arquitetura comportam-se no modo assíncrono, para que sejam evitadas esperas inúteis. Assim, esses elementos possuem uma caixa de mensagens e o envio de mensagem por um elemento emissor significa depositar a meta mensagem na caixa do destinatário.

Neste trabalho, estamos enfocando apenas o terceiro elemento dessa Meta-Arquitetura, o Meta-Objeto. Em particular, o mecanismo de interação entre os objetos de aplicação, que podem ser executados no modo síncrono ou assíncrono. A interação é resolvida como uma troca de mensagens e os dois modos, o síncrono e o assíncrono, são tratados no mesmo mecanismo, que descrevemos nas seções a seguir.

Um objeto da aplicação, na Meta-Arquitetura Alana, adquire capacidade de concorrência, transação, persistência e distribuição, se na mensagem enviada para a sua criação (construtor) é preenchido um argumento associado a um gerente de objetos. Convém notar que as oportunidades da programação para a indicação dessas características ficaram restritas apenas ao momento da criação de um objeto. Essa decisão foi tomada porque adotamos a abordagem ortogonal de persistência, em que a indicação dessas facilidades deve ser feita para o objeto e não para a classe.

A partir da criação de um objeto de aplicação que esteja associado a um Meta-Objeto, toda vez que for enviada mensagem a esse objeto, o gerente de objetos e de transações da Meta-Arquitetura (GOT) deverá ter providenciado sua seleção (ativação). Isto significa que o GOT, entre suas tarefas de seleção, faz uma associação da parte seqüencial do objeto com o Meta-Objeto. O Meta-Objeto fica encarregado de resolver as facilidades de concorrência e de atomicidade local para o objeto de aplicação, além de cooperar com a transação no nível global e ainda participar da facilidade de persistência.

Essas facilidades, oferecidas pela meta-arquitetura

às aplicações, embora sejam resolvidas pelo meta-objeto, não serão vistas nesse trabalho, porque estamos enfocando apenas os mecanismos de interação.

A cooperação entre o Meta-Objeto e o seu objeto seqüencial, no interior de um objeto de aplicação, se refere basicamente a tarefas de tratamento de mensagens trocadas entre objetos de aplicação. Essas mensagens tratadas pelo Meta-Objeto são: a) as mensagens enviadas do objeto seqüencial para algum outro objeto de aplicação, b) as recebidas de algum outro objeto de aplicação para o objeto seqüencial, e c) os retornos dessas mensagens. Observamos também que todas essas mensagens são levadas em conta no controle de concorrência e na atomicidade local, e que, nesse processo, o objeto seqüencial pode ser consultado para que o controle saiba se duas operações são conflitantes, participando, assim, discretamente, nas tarefas de concorrência e de atomicidade local.

Além disso, cada Meta-Objeto avisa ao elemento gerente de transação sobre a participação de seu objeto de aplicação na transação, enviando as mensagens recebidas, enviadas e os retornos, que são todas registradas por esse gerente de transação. O elemento gerente de transação, por sua vez, faz pedidos a cada Meta-Objeto, relativos a consistência, gravação, confirmação de gravação e fim normal de transação, enquadradas como tarefas de colaboração com a transação no nível global.

Construção e Iniciação O Meta-Objeto e a sua Caixa de Mensagens sempre são criados pelo Gerente de Objetos e Transações (GOT). O Meta-Objeto, ao ser criado, recebe como argumentos o Objeto e a Caixa de Mensagens previamente criados. Após isso, o Meta-Objeto associa-se à Caixa de Mensagens recebida e a avisa para fazer o mesmo. Somente depois dessa associação, onde foi estabelecida a ligação entre as duas partes, o Meta-Objeto pode começar seu funcionamento.

Objeto seqüencial, teoricamente, não deveria ter conhecimento de seu Meta-Objeto. Porém, enquanto não é incluído o uso de *proxy* como referência, é realizada uma associação explícita entre o Meta-Objeto e seu öbjeto sequencial. Essas associações são realizadas no construtor do Meta-Objeto.

Na nossa implementação, consideramos apropriado que o Meta-Objeto seja executado por uma *thread*. Para isso foi necessário implementar a interface *Runnable* do Java. No processo de construção do Meta-Objeto, o último passo é a iniciação da *thread*. Essa *thread*, por sua vez, executa o método *run()* do Meta-Objeto, colocando-o em funcionamento. Esse método *run()* consiste simplesmente em retirar mensagens da Caixa e tratá-las devidamente.

A Caixa de Mensagem realiza apenas duas operações: receber mensagem e fornecer mensagem. Ela pode receber mensagens de vários elementos da Meta-Arquitetura, como outros Meta-Objetos, Gerentes de Transação e Gerentes de Objetos e Transações. Ela pode fornecer apenas para o seu Meta-Objeto associado, que está sempre tentando consumir suas mensagens, caso esteja em funcionamento. Se a Caixa estiver vazia, operações de sincronização são ativadas para fazer com que o Meta-Objeto passe ao estado de espera (*wait*).

Funcionamento e Ciclo de Mensagens Quando uma mensagem chega ao Meta-Objeto, ele dá o tratamento de acordo com o tipo, que pode ser uma das várias classes de mensagens que circulam pelos elementos da Meta-Arquitetura. Estamos, no entanto, enfocando os quatro tipos seguintes de mensagens que se desdobram da interação entre dois objetos: MsgME, MsgMR, MsgRR e MsgRE. Cada vez que uma dessas mensagens chega ao Meta-Objeto, é instanciado um Tratador de Mensagem (que inicia uma *thread*). O comportamento do tratador pode ser visto pela Figura 1 e pelo código a seguir.

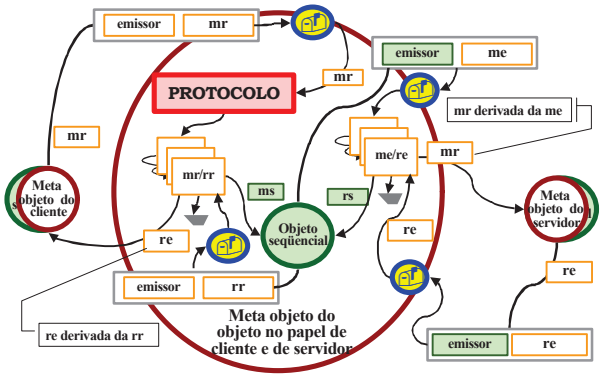

Figura 1: Ciclo de interações entre três objetos de uma aplicação e os controles exercidos pelos elementos da Meta-Arquitetura Alana.

```
/* Núcleo do Meta-Objeto. Aqui se inicia
toda a operação básica  * do Elemento.
Consite apenas em retirar mensagens da
Caixa e  * iniciar seu tratamento */
public synchronized void run() {
   Mensagem m;
   AtendeMsg am;
   while (true) {
m = caixamsg.tiraMsg();
listamsg.add(m);
am = new AtendeMsg(m, "teste");
am.start();
   }
}
/* Classe interna ao Meta-Objeto. Seu
objetivo é dar
* tratamento independente a cada
mensagem sendo tratada
* pelo Meta-Objeto. Diferentemente do
Meta-Objeto, que
* implementa a interface  Runnable, essa
classe extende a
* classe Thread. Essa classe é interna
porque precisa
* enxergar membro privados da classe
MetaObjeto e é uma
```

```
* Thread para executar em paralelo com
outras instâncias
* da mesma classe */
class AtendeMsg extends Thread {
// Mensagem que é tratada por um
instância dessa classe
   private Mensagem msg;
   /* Único construtor */
   public AtendeMsg(Mensagem msg, String
str) {
super(str);
this.msg = msg;
   }

/* Seleciona o devido tratamento da
mensagem e o
 * executa. É apenas isso que cada
instância faz. E faz
 * apenas uma vez */
public synchronized void run() {
   switch (msg.getContexto()) {
case Mensagem.CME:
trataCME();
               break;
     case Mensagem.CMR:
               trataCMR();
     break;
case Mensagem.CRR:
     trataCRR();
     break;
     case Mensagem.CRE:
     trataCRE();
     break;
   }
   notifyAll();
}
```

No interior dessa *thread*, a operação correspondente a mensagem é executada, garantindo, assim, o assincronismo da mensagem. O funcionamento de cada uma das quatro mensagens está explicado nas seções seguintes.

MsgMe (Meta Mensagem Enviada) A MsgME é criada quando o objeto de aplicação associado ao Meta-Objeto tem a necessidade de se comunicar com outro objeto da aplicação. O Objeto emissor comunica ao seu Meta-Objeto que precisa enviar uma mensagem de interação para um objeto alvo. Este mecanismo, no momento[2], é simulado pelo método `encapsula()` do Meta-Objeto, que gera a mensagem `MsgMe` e a põe em sua própria caixa. Essa simulação faz com que a Meta-Arquitetura seja transparente, parecendo uma interação direta entre o objeto remetente e o objeto alvo.

```
/*
Porta de comunicação do objeto com seu
meta-objeto. Cria uma mensagem que é
enviada para a caixa. Faz parte do
remendo e toda mensagem criada é ME.
*/
public Object encapsula(Mensagem cm,
Objeto o, String metodo, Object[]
params, String nome) {
Mensagem nova = new Mensagem(cm,
Mensagem.CME);
  nova.setPacote(o, metodo, params);
  nova.setOrigem(this);
  caixamsg.poeMsg(nova);
  nova.retornou();
  return nova.getResult();
}
```

O tratamento da MsgME consiste em criar uma MsgMR com os dados da operação a ser efetuada no objeto alvo e colocá-la na caixa do Meta-Objeto do objeto alvo.

O Tratador associado a uma MsgME gera uma MsgMR com os dados da operação a ser efetuada no objeto alvo e a coloca na Caixa do Meta-Objeto do mesmo, como podemos ver no código a seguir.

Após essas operações, a *thread* do Tratador encerra, deixando de existir. A MsgME original, porém, ainda necessita receber a resposta, o que não é feito nesse Tratador. Por isso que o Tratador apenas dá início ao tratamento.

```
/* Método de tratamento da Mensagem CME
*/
private void trataCME() {
   Mensagem nova = new Mensagem(msg,
Mensagem.CMR);
   /* Ações de marshalling */
   nova.setAlvo(msg.getAlvo());
   nova.setOrigem(msg.getOrigem());
   nova.setParam(msg.getParam());
   nova.setMetodo(msg.getMetodo());

msg.getAlvo().getMetaObjeto().getCaixaMs
g().poeMsg(nova);
}
```

MsgMR (Meta Mensagem Recebida) O tratamento desta mensagem é mais complexo do que o das outras três. Este fato ocorre porque é necessário que o método que vem *marshalled* na mensagem seja executado pelo objeto alvo. Com um mecanismo de reflexão, é providenciado o *unmarshalling* das informações vindas com a mensagem `MsgMR`. Dessa forma, a mensagem de invocação do método é preparada, para então enviá-la ao objeto alvo. Quando o método estiver executando no objeto alvo, podem surgir envios de mensagens, que causam novas interações e, assim por diante.

Enquanto o método não retornar, a *thread* do Tratador correspondente a `MsgMR` fica em estado de espera. Após o retorno, é criada uma nova mensagem, `MsgRR`, em que é colocado o resultado da operação. Essa mensagem `MsgRR` é colocada na caixa do próprio Meta-Objeto.

Como no caso anterior, após essas operações, a *thread* do Tratador deixa de existir. Porém, a `MsgMR` ainda espera ser casada com a `MsgRR`, o que não é feito nesse Tratador. O código relativo a este procedimento pode ser visto, a seguir:

```
/*
```

[2] Por enquanto estamos implementando por simulação este mecanismo. Está prevista a implementação através da referência ao objeto, que terá um método que tratará as mensagens enviadas, colocando-as na caixa de mensagens do Meta-Objeto associado ao objeto.

```
Método de tratamento da Mensagem CMR
*/
private void trataCMR() {
/* Aqui entra a chamada ao protocolo
if (not protocolo.inv (msg)) return;
*/
/* Ações para o unmarshalling do método
a ser invocado
*/
Class cl;
Method mt;
Object ret = new Object();
Object o = msg.getAlvo();
Object[] params = msg.getParam();
Class[]args = new Class[params.length];
int i;
String met = msg.getMetodo();
Mensagem nova;
cl = o.getClass();
for (i=0; i < params.length; i++) {
     Class aux = params[i].getClass();
     args[i] = aux;
   }
/* Fim das ações de unmarshalling */
try {
System.out.println("Tratando " + msg);
mt = cl.getMethod(met, args);
ret = mt.invoke(o, params);
nova = new Mensagem(msg, Mensagem.CRR,
msg + ".CRR");
nova.setOrigem(msg.getOrigem());
nova.setResult(ret);
caixamsg.poeMsg(nova);
}
catch (NoSuchMethodException e)
{System.out.println("Erro: " + e);
}
catch (IllegalAccessException e)
{System.out.println("Erro: " + e);
}
catch (InvocationTargetException e)
{System.out.println("Erro: " + e);
}
}
```

MsgRR (Meta Mensagem Retorno da Recebida) A primeira ação do Meta-Objeto sobre a meta mensagem `MsgRR` é fazer o seu casamento com a `MsgMR` correspondente. Em seguida, deve retornar ao Meta-Objeto do objeto origem uma `MsgRE`. Fazemos notar que qualquer uma dessas meta mensagens que chegam ao Meta-Objeto são incluídas em uma lista que auxilia no controle das mensagens pendentes. Assim, quando o Meta-Objeto termina de fazer o casamento e envia a `MsgRE` ao objeto origem, o contexto desse par de meta mensagens `MsgMR` e `MsgRR` não tem mais utilidade, sendo, então, removido do Meta-Objeto que as trataram. A seguir, descrevemos o código para o tratamento da `MsgRR`, que é executado na *thread* do Tratador.

```
private void trataCRR() {
     Msgem ant;
     Msgem nova;
protocolo.res (msg));
ant =(Msgem)listamsg.get
(listamsg.indexOf(msg.getMsgAnt()));
nova = new Msgem(ant.getMsgAnt());
nova.setResult(msg.getResult());
msg.getOrigem().getCaixaMsg().poeMsg(nov
a);
listamsg.remove(listamsg.indexOf(msg.get
MsgAnt()));
listamsg.remove(listamsg.indexOf(msg));
}
```

Podemos observar pelo código acima, que a remoção do contexto do par de meta mensagens `MsgMR` e `MsgRR` é traduzido na sua retirada da lista de pendências e o encerramento da *thread*. Além disso, chamamos a atenção para a diferença do tratamento da `MsgRR` em relação aos outros dois tipos de meta mensagens abordados anteriormente. O tratamento da `MsgRR` fica todo realizado no seu Tratador.

MsgRE (Meta Mensagem Retorno da Enviada) A meta mensagem `MsgRE` fecha um ciclo de interação. Em primeiro lugar, ela é associada a `MsgME` correspondente. Em segundo lugar, o Tratador da `MsgRE` confirma o casamento entre essas duas mensagens, avisando que a `MsgME` retornou, sob a forma de `MsgRE`. Neste momento, o Tratador da `MsgMR` que invocou o método que gera a `MsgME`, cujo retorno é a própria `MsgRE`, está em estado de espera, porque o método **`encapsula()`** chama o método **`retornou()`** da `MsgME` criada. Esse método permanece em estado de espera até que seja executado o método **`retorna()`** da mesma mensagem. O Tratador avisa que a `MsgME` retornou chamando o método **`retorna()`**. Em vista disso, o método **`encapsula()`** sai do estado de espera e pode retornar o resultado que veio através das mensagens, para o objeto de aplicação emissor. A partir deste momento a execução da interação em questão é considerada encerrada e as duas meta mensagens `MsgRE` e `MsgME` são retiradas da lista.

Podemos notar, pelo código a seguir, que esta implementação é semelhante a de `MsgRR`, pelas seguintes ações: há um casamento a ser resolvido; todo o tratamento da `MsgRE` é efetuado neste Tratador, além de concluir o tratamento da `MsgME`; e esse Tratador termina, no final da execução.

```
/* Método de tratamento da Mensagem CRE
*/
private void trataCRE() {
Mensagem ant;
ant = listamsg.get(
listamsg.indexOf(msg.getMsgAnt()));
ant.setResult(msg.getResult());
listamsg.remove(
listamsg.indexOf(msg.getMsgAnt()));
listamsg.remove(listamsg.indexOf(msg));
ant.retorna();
}
```

Caixa de mensagem vazia Caso a caixa de mensagens esteja vazia, o Meta-Objeto recebe um aviso para ficar em estado de espera, até que alguma mensagem nova

chegue na caixa.

O Meta-Objeto, ao receber uma mensagem de liberação de transação, procede as ações de remoção de todos os vestígios deixados pela transação que está sendo encerrada. Após essas providências, ele verifica se existe alguma mensagem na caixa, para tratá-la. Caso a caixa esteja vazia, ele verifica se não existe mais alguma outra aplicação (transação) cliente dele. Nesse caso, ele se autodesativa encerrando a *thread,* liberando todas as estruturas do seu contexto e avisando ao GOT sobre esta condição.

Considerações Apesar de enfocarmos apenas os mecanismos de interação, é necessário, para poder mostrar o seu funcionamento, implementar parte dos outros aspectos do controle de concorrência. Esses outros aspectos, a concorrência interna e o do modelo integrado, estão intimamente ligados com os mecanismos de interação.

O modelo integrado é implementado com a ajuda do Meta-Objeto, que possui um controle de concorrência interna. Nesse Meta-Objeto são utilizados *proxies*, objetos independentes que permitem que as mensagens sejam tratadas enquanto deixa o objeto cliente livre para outra tarefa. Para cada Mensagem existe um *proxy*. Esse *proxy* também permite que sejam dinamicamente especificados o nome, o método e o endereço de retorno.

O Meta-Objeto, os *proxies* e as Mensagens, juntamente com a Caixa de Mensagens e o protocolo ajudam a implementar a característica de *aceitação incondicional seguindo apenas as regras de concorrência*. Ou seja, qualquer mensagem para o objeto é colocada incondicionalmente na Caixa de Mensagens do Meta-Objeto associado. Um *proxy* (que são os Tratadores) chama o protocolo que verifica se a operação associada à mensagem recebida gera conflito de concorrência e, em caso negativo, continua o tratamento. Esse conjunto constitui o que chamamos Objeto Ativo, o qual se comunica através de *Message Passing* (Troca de Mensagens).

Das muitas meta-mensagens que circulam no ambiente de execução com a Meta-Arquitetura Alana, apenas as quatro associadas às interações entre objetos da aplicação são consideradas neste trabalho. Quando uma mensagem é enviada pelo objeto seqüencial, é iniciado um ciclo de interação, que se desdobra nas quatro meta-mensagens (`MsgME`, `MsgMR`, `MsgRR`, `MsgRE`). Para cada uma dessas meta-mensagens, é criado um Tratador. Todos esse tratadores colaboram entre si e, ao final de cada operação, deixam de existir.

Com a descrição do modelo de computação confirmamos o alcance das soluções previstas no final do 2, em que foram discutidas as alternativas existentes, para cada um dos três aspectos.

Três principais fatores contribuíram para ter gerado esses resultados positivos: a disciplina adotada para seguir os critérios definidos na seção 2; a consciência da dificuldade das combinações entre os mecanismos para os três aspectos do controle de concorrência; e o estudo meticuloso dos recursos de Java utilizados. Os recursos são relativos aos aspectos de orientação a objetos, a reflexão e *multithreading*, visando conciliá-los com as necessidades definidas no modelo de controle de concorrência adotado para a Meta-Arquitetura Alana.

Na próxima seção, explicamos alguns dos exemplos utilizados para validar a nossa implementação.

4. Conclusões

Chegamos a essas conclusões, finalmente, confirmando aquelas expectativas para os mecanismos de interação colocadas nas considerações iniciais. Essas considerações tinham por finalidade imediata atender os requisitos do modelo de meta arquitetura Alana, que em síntese é o de para favorecer o modelo integrado de objeto concorrente: objetos ativo e troca de mensagens.

Para os objetos ativos, foram implementados os Tratadores (*proxies*), que funcionam como *threads* para realizar o tratamento de mensagens independentemente do procedimento do restante do objeto seqüencial e do Meta-Objeto.

A troca de mensagens, por sua vez, utiliza o recurso da Caixa de Mensagens. A Caixa permite a aceitação incondicional de pedidos, além de ser perfeitamente utilizável pelos outros Elementos da Meta-Arquitetura Alana: o gerente de objetos e transações (GOT) e o gerente de transações (GT), que não foram enfocados no escopo deste trabalho.

A Caixa de Mensagens e os Tratadores operam em conjunto com o protocolo para garantir a aceitação incondicional seguindo apenas as regras de concorrência. O Meta-Objeto retira uma Mensagem da Caixa e cria um Tratador para ela. O Tratador, por sua vez, passa a Mensagem para o protocolo, que avalia as regras de concorrência, para finalmente, caso não haja conflitos, tratá-la adequadamente. Há vários Tratadores simultâneos. Eles foram implementados como simples *threads* Java para realizar a concorrência interna.

Para desenvolver esses mecanismos, foi necessário entender o modelo de concorrência da Meta-Arquitetura Alana, e estudar quais os recursos da linguagem Java (escolhida para ser usada), que mais se adequariam para a implementação.

Uma das dificuldades que tivemos foi a inexistência de uma linguagem com os requisitos necessários para o emprego de Meta-Objeto no estilo que a Meta-Arquitetura Alana necessita. Assim, tivemos que contornar essa lacuna com o auxílio de simulações para prover a transparência desejada na interação, pelo lado da programação da aplicação. Os resultados se mostraram satisfatórios quando conseguimos fazer com que objetos conseguissem valores armazenados em outros objetos distantes, apenas por troca de mensagens, de acordo com as especificações da Meta-

Arquitetura de maneira relativamente simples. No momento, estamos substituindo esse artifício de integração entre o Meta-Objeto e o objeto seqüencial com o uso de *proxy* de Java™.

Como resultados desse projeto, além da implementação, obtivemos duas contribuições importantes. Uma delas foi o exercício de recursos de *thread* e reflexão de Java, que não são considerados tão simples na atividade de programação de aplicações. A outra faz parte do próprio compromisso do modelo Alana, que é a tirar da incumbência de um programador Java essas questões de reflexão e de concorrência, que ficam resolvidas no Meta-Objeto que é transparente à programação.

O protocolo de controle de concorrência está em fase de implementação. Com a integração substituída pelo proxy de JavaTM e a conclusão do protocolo, teremos um protótipo da Meta-Arquitetura Alana funcionando e respaldando nossas experiências com novos modelos de transação.

Como trabalhos futuros, pretendemos adaptar nossos mecanismos ao paradigma de interação baseado em evento *publish/subscribe*, que permite o total desacoplamento de tempo, espaço e sincronização entre os *publishers* e os *subscribers*. Essa adaptação é motivada por entendermos que a Meta-Arquitetura Alana oferece as qualidades de serviço: persistência e transação consideradas para esse paradigma [7].

Referências

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

[2] P. America. POOL-T: A Parallel Object-Oriented Language. Object-Oriented Concurrent Programming – MIT Press. 1987.

[3] Colin Atkinson. *Object-Oriented Reuse, Concurrency and Distribution* - AN ADA-BASED APPROACH. Addison-Wesley Publishing Company. ACM Press. New York, New York, 1991.

[4] Steve Cook and John Daniels. *Design Object Systems - Object-oriented Modelling with Syntropy*. The Object-Oriented Series - Prentice Hall, 1994.

[5] 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.

[6] E. W. Dijkstra. Cooperating Sequential Processes. Programming Languages. (ed. Genuys), Academic Press. 1968.

[7] P. Eughster, P. Felber, R. Guerraoui and A. Kermarrec. The Many faces of Publish/Subscribe. *ACM Computing Surveys*. 35(02), June 2003.

[8] P. Eughster, R. Guerraoui and C. Damm. On Objects and Events. In Proceedings of the ACM Conference on Object-Oriented Programming, System, and Applications (OOPSLA'01). 2001.

[9] P. Brinch Hansen. The Programming Language Concurrent Pascal. IEEE Trans. of Soft. Eng. Vol. SE-1 n. 2, June 1975.

[10] P. Brinch Hansen. Distributed Processes – a Concurrent Programming Concept. Communications of ACM, 21(11). 1978.

[11] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. Communications of ACM, 17(10). 1974.

[12] C. A . R. Hoare. Communicating Sequential Processes. Communications of ACM, 21(8). 1978.

[13] Norman Hutchinson. *Emerald: An Object Based Language for Distributed Programming*. PhD Dissertation. University of Washington, Seattle, WA 98195. Technical Report 87-01-01, 1987.

[14] Jameela Al-Jaroodi and Nader Mohamed. An Object-Passing Model for Parallel Programming. In Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC´03).

[15] Doug Lea. *Concurrent Programming in Java*. Addison Wesley, 1996.

[16] Oscar Niertrasz. A Tour of *Hybrid* - A Language for Programming with Active Objects. *Advances in Object-Oriented Software Engineering*. ed. D. Mandrioli and B. Meyer, Prentice-Hall, 1992.

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

[18] C. Schaffert, T. Cooper, B. Bullis and M. Killian. An Introduction to Trellis/Owl OOPSLA'86 Conference Proceedings, SIGPLAN Notices (Special Issue). 21(11). 1986.

[19] Robert W. Sebesta. Conceitos de Linguagens de Programação. Ed Bookman 1999.

[20] D. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*. 30(02), June 1998.

[21] M. Tokoro and Y. Ishikawa. Concurrent Programming in Orient84/K: An Object-Oriented Knowledge Representation Language. *ACM SIGPLAN Notices*. 21(10), October 1986.

[22] Chris Tomlinson and Mark Scheevel. Concurrent Object-Oriented Programming Languages. *Object-Oriented Concepts, Databases, and Applications - Addison-Wesley Publishing Company*. 1988.

[23] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger - A Quartely Publication of the Special Interest Group on Programming Languages*. ACM Press. 1(1), August 1990.

[24] N. Wirth. Modula: A Language for Modular Multi-Programming. Software – Practice and Experience. v. 7 1977

[25] Y. Yonezawa, J. Briot, E. Shibayama. Object-Oriented Programming in ABCL/1. OOPSLA'86 Proceedings, SIGPLAN Notices. 21(11) 1986

[26] Y. Yokote, Tokoro. The Design and Implementation of Concurrent Smalltalk. OOPSLA'86 Proceedings, SIGPLAN Notices. 21(11) 1986.