Otimizações de Co-Localização em AspectJRMI

Marco Túlio de Oliveira Valente, Rodrigo Palhares, Diana Campos Leão

Departamento de Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais
mtov@pucminas.br

Abstract. *In object oriented middleware, collocation optimizations support direct remote method invocations when client and server objects are placed in the same address space. In such cases, the cost of remote calls is reduced since we remove the overhead of using the middleware infrastructure. In this paper, we describe the implementation of collocation optimizations in AspectJRMI, an aspect oriented communication middleware designed to support compile time reconfigurations.*
Keywords: aspect-oriented middleware; collocation optimizations.

Resumo. *Em plataformas de middleware orientadas por objetos, otimizações de co-localização viabilizam invocar de forma direta métodos remotos quando objetos servidores e clientes estão localizados no mesmo espaço de endereçamento. Com isso, reduzem-se os custos de tais invocações, pois as mesmas ocorrem sem o overhead inerente à utilização da infra-estrutura de comunicação implementada pelo middleware. Descreve-se neste artigo a incorporação de otimizações de co-localização em AspectJRMI, um middleware orientado por aspectos que permite configurações de seus serviços em tempo de compilação.*
Palavras-chave: middleware orientado por aspectos; otimizações de co-localização.

1 Introdução

A principal abstração provida por sistemas de *middleware* orientados por objetos é a idéia de chamada remota de métodos, por meio da qual métodos de objetos remotos são chamados de forma transparente, isto é, usando a mesma sintaxe de chamadas locais. Para implementar esta abstração, sistemas de *middleware* encapsulam diversos detalhes inerentes a programação em ambientes de rede, tais como primitivas e protocolos de comunicação, *marshalling* e *unmarshalling* de dados, heterogeneidade de ambientes, sincronização etc.

No entanto, em determinados cenários, pode ocorrer de o objeto remoto ser implantado (*deployed*) ou então migrar para o processo cliente da invocação (ou vice-versa). Em tais cenários, seria desejável que mensagens para o objeto servidor fossem despachadas sem utilizar a infra-estrutura do *middleware* e, portanto, sem incorrer nos custos de comunicação via uma pilha de protocolos de rede. Em sistemas de *middleware*, este tipo de otimização é chamado de *otimização de co-localização* [8]. Tais otimizações

são disponibilizadas, por exemplo, em diversas implementações de CORBA, incluindo os sistemas TAO [7] e ORBacus [5]. Outros sistemas de *middleware*, como ICE [3] e COM/DCOM [1], também possuem suporte a otimizações de co-localização.

Em geral, implementações de co-localização apresentam um comportamento transversal [10]. Sempre que uma referência remota é obtida em um *middleware* orientado por objetos, deve-se verificar se a mesma está associada a um objeto local, isto é, a um objeto localizado no mesmo espaço de endereçamento do objeto que obteve a referência. Esta verificação requer modificações nos seguintes componentes de uma plataforma de *middleware*: *stub* (para verificar se o retorno de um método remoto é uma referência para um objeto local), *skeleton* (para verificar se argumentos de um método remoto são referências para objetos locais) e no próprio ORB (para armazenar, por exemplo, uma tabela de referências para objetos locais, a qual é consultada pelas verificações descritas anteriormente). Além disso, a implementação de co-localização normalmente utiliza novos componentes, como *stubs* e *skeletons* especiais, os quais são responsáveis por incorporar a semântica de co-localização a uma chamada remota. Estes *stubs* e *skeletons* especiais não utilizam protocolos de comunicação em rede [8].

Assim, o suporte a otimizações de co-localização em plataformas tradicionais de *middleware* orientados por objetos não apresenta níveis adequados de modularização [10], o que torna mais complexa sua implementação e manutenção. Além disso, as plataformas atuais de *middleware* com suporte a otimizações de co-localização sempre carregam código para implementar esta funcionalidade, independentemente de a mesma ser requerida ou não em um determinado sistema distribuído. Em outras palavras, o usuário de um sistema de *middleware* não pode desativar o suporte a otimizações de co-localização, de forma a obter um sistema que possua um código executável de menor tamanho, que utilize menos espaço de memória e que possua um melhor desempenho.

Neste artigo, descreve-se uma implementação orientada por aspectos de otimizações de co-localização. Esta implementação foi realizada sobre a infra-estrutura disponibilizada pelo sistema AspectJRMI [2, 6], o qual é um sistema de *middleware* configurável em tempo de compilação. Em AspectJRMI, programadores podem agregar funcionalidades extras ao núcleo do sistema, o qual oferece um serviço de chamada síncrona de métodos remotos. Via de regra, estas funcionalidades tendem a ser transversais aos componentes deste núcleo e, por este motivo, são implementadas por meio de aspectos. A versão atual de AspectJRMI possui aspectos para as seguintes funcionalidades: chamadas *oneway*, chamadas assíncronas, passagem de parâmetros por valor-resultado, combinadores de serviços e objetos interceptadores.

O restante deste artigo está organizado como descrito a seguir. Na Seção 2, descrevem-se a interface de programação e os principais aspectos usados para incorporar otimizações de co-localização em AspectJRMI. A Seção 3 apresenta os resultados de um experimento conduzido com o intuito de mostrar os ganhos de desempenho proporcionados por otimizações de co-localização. A Seção 4 descreve trabalhos relacionados e a

Seção 5 avalia o sistema proposto e apresenta as conclusões do artigo.

2 Otimizações de Co-Localização em AspectJRMI

2.1 Interface de Programação

Objetos com métodos remotos sujeitos a otimizações de co-localização devem implementar uma interface derivada de uma interface de nome `Collocated` (além da interface `Remote`, conforme usual em aplicações baseadas em AspectJRMI).

O seguinte trecho de código mostra a definição, instanciação, ativação e registro no serviço de nomes de um objeto remoto, cuja classe `A_Impl` implementa uma interface remota `A`, sujeita a otimizações de co-localização (já que estende `Collocated`):

```
 1: interface A extends Remote, Collocated {
 2:   public int foo(int a, float b) throws ArcademisException;
 3: }
 4: class A_Impl extends ArcademisRemoteObject implements A {
 5:   .....
 6:   public int foo(int a, float b) throws ArcademisException {
 7:     .....
 8:   }
 9:   public static void main(String[] args) {
10:     .....
11:     A a = new A_Impl();              // cria objeto remoto
12:     ArcademisNaming.bind("objx", a);// registra objeto remoto
13:     a.activate();                    // ativa objeto remoto
14:     .....
15:   }
16: }
```

O próximo trecho de código mostra uma chamada ao serviço de nomes para obter uma referência remota para o objeto previamente registrado com o nome `objx` e, em seguida, realiza uma chamada remota usando a referência obtida:

```
17: A a = (A) ArcademisNaming.lookup("objx");  // servidor nomes
18: int x= a.foo(10,1.1);                      // chamada remota
```

Em AspectJRMI, caso o objeto que realizou a chamada na linha 18 e o objeto servidor instanciado na linha 11 estejam alocados no mesmo espaço de endereçamento, a referida chamada será otimizada de forma transparente aos usuários do sistema. Em outras palavras, a chamada da linha 18 será realizada de forma direta, sem utilizar nenhum componente interno da infra-estrutura de comunicação provida por AspectJRMI.

2.2 Implementação

A implementação de otimizações de co-localização em AspectJRMI usa aspectos para realizar as seguintes tarefas:

Interceptar instanciações de objetos sujeitos a otimizações. Para realizar esta interceptação, o seguinte *pointcut* é usado:

```
pointcut newCollocatedObject(ArcademisRemoteObject r):
  execution(Collocated+.new(..)) && this(r);
```

Este *pointcut* inclui a execução de construtores de objetos remotos sujeitos a otimizações (isto é, que implementem `Collocated`). Um *advice* do tipo *after* associado ao mesmo armazena referências para estes objetos em uma tabela *hash*, fornecendo como chave um identificador único na rede para o objeto instanciado. Este identificador único é gerado a partir da combinação de três parâmetros: um valor que identifica a máquina virtual na qual o objeto foi instanciado, um valor que identifica o instante de tempo em que o objeto foi instanciado e um contador que armazena o número de vezes que foi solicitada a criação de identificadores únicos. As referências para objetos remotos armazenadas nessa tabela são *fracas*, isto é, as mesmas não são consideradas para fins de coleta de lixo. Assim, tais referências não impedem o coletor de lixo de remover um objeto remoto, caso o mesmo somente seja referenciado a partir da tabela *hash* manipulada por este *advice*.

Interceptar obtenções de referências remotas. Em sistemas de *middleware* orientados por objetos, existem apenas duas maneiras de se obter uma referência remota (isto é, uma referência para um objeto localizado em um espaço de endereçamento remoto): como resultado de uma chamada a um método remoto (incluídas neste caso consultas ao serviço de nomes) ou como argumento de uma chamada remota de métodos. Assim, a implementação de co-localização em AspectJRMI deve: (1) interceptar toda obtenção de uma referência remota *rr*; (2) obter o identificador *id* do objeto remoto denotado por *rr* (o que é imediato, pois em sistemas de *middleware* derivados de Arcademis referências remotas sempre incluem o identificador do objeto referenciado pelas mesmas); (3) consultar a tabela *hash* de objetos remotos instanciados no espaço de endereçamento corrente, fornecendo *id* como chave; (4) caso a consulta seja bem sucedida, retornar para a aplicação cliente não mais a referência *rr*, mas sim a referência *rl* (uma referência padrão de Java) associada a *id* nesta tabela.

A fim de interceptar métodos que retornam referências remotas para objetos sujeitos a otimizações de co-localização, usa-se o seguinte *pointcut*:

```
pointcut RemoteReferenceAsResult(): call(Remote+ *(..));
```

A fim de interceptar métodos remotos que recebem referências remotas como argumentos, usa-se o seguinte *pointcut*:

```
pointcut RemoteReferenceAsArgument():
  call(public Object Stream.readObject()) && within(*_Skeleton);
```

Veja que este *pointcut* intercepta chamadas ao método `readObject` realizadas no escopo léxico de classes que representam *skeletons* em AspectJRMI. O método

`readObject` é usado para realizar o *unmarshalling* de parâmetros que são referências para objetos. Um *advice* do tipo *around* é usado então para obter o resultado de tais chamadas. Caso o mesmo seja uma referência do tipo `Collocated` e o objeto referenciado seja local (isto é, esteja presente na tabela *hash* mencionada anteriormente), entra em ação o código destinado a substituir tal referência por uma referência padrão de Java.

2.3 Semântica de Chamadas Remotas

A implementação de co-localização descrita anteriormente é classificada como *direta* [8], visto que a mesma substitui uma referência de rede (que aponta para um *stub*) por uma referência *direta* para o objeto servidor. Esta referência direta é uma referência padrão de Java. A principal vantagem deste tipo de co-localização é o ganho de desempenho, pois a mesma elimina completamente a infra-estrutura do *middleware*, transformando para todos fins uma chamada remota em uma chamada local.

No entanto, em determinadas situações, chamadas remotas possuem uma semântica distinta de chamadas locais. A principal diferença é que objetos serializáveis em AspectJRMI são passados por valor quando usados como parâmetros de chamadas remotas. No entanto, se tais chamadas forem alvo de uma otimização, o que será transmitido para o método chamado passa a ser uma referência para os argumentos que denotam objetos serializáveis, conforme usual em Java (e não mais uma cópia dos mesmos gerada durante o processo de *marshalling*, conforme usual em AspectJRMI).

Nesta extensão, a fim de manter a semântica usual de Java em chamadas remotas alvo de otimizações de co-localização, foi desenvolvida uma extensão à implementação descrita anteriormente. Do ponto de vista da interface de programação, esta extensão apenas requer que objetos com métodos sujeitos a otimização implementem uma interface que estenda a interface `CollocatedProxy` (em vez de `Collocated`).

A implementação dos *pointcuts* mostrados na Seção 2.2 foi alterada de forma a referenciar esta nova interface. Os *advices* associados a estes *pointcuts* em vez de retornarem uma referência direta para o objeto servidor, retornam uma referência para um *proxy* do mesmo. Este *proxy* é gerado dinamicamente, usando o conceito de classes *proxy* dinâmicas da API de reflexão de Java. O tratador de invocação associado a estes *proxies* simplesmente percorre o vetor de argumentos das chamadas remotas. Caso um destes argumentos implemente a interface `Marshalable`, substitui-se o objeto referenciado por uma cópia do mesmo[1]. Em seguida, o tratador de invocação repassa a chamada para o objeto servidor.

3 Resultados Experimentais

Esta seção apresenta os resultados de um experimento realizado com o objetivo de medir os ganhos de desempenho gerados por otimizações de co-localização em aplicações

[1]Em AspectJRMI, objetos serializáveis que são passados por valor devem implementar uma interface de nome `Marshalable`.

distribuídas. Os programas do experimento foram compilados e executados usando o ambiente JDK (versão 1.5.0) e o compilador de aspectos `ajc` (versão 1.1). O experimento foi realizado em uma máquina com processador Athlon XP2600, com 1 GB de RAM. Objetos clientes e servidores foram sempre instanciados em uma única JVM executada nesta máquina. Os tempos medidos sem otimização não incluem o tempo de transmissão de dados em uma rede. Ressalte-se, no entanto, que tais tempos incluem todo o *overhead* da infra-estrutura de comunicação de AspectJRMI (*marshalling*, *unmarshalling*, protocolo do *middleware*, TCP/IP etc).

No experimento realizado, a aplicação servidora foi desenvolvida de forma a instanciar um objeto remoto com o seguinte método:

```
void foo (long iteracoes) throws ArcademisException {
   double a=Math.random(), b=Math.random();
   for(long i=0; i<iteracoes; i++) {
      a=a*(b*(a*(b*(a*(b*(a*(a*(b*a)))))));
      a=Math.random(); b=Math.random();
   }
}
```

A partir de um objeto cliente instanciado na mesma JVM do objeto servidor, o método `foo` foi então invocado diversas vezes, variando o parâmetro que representa o número de iterações. Na primeira seqüência de invocações, otimizações de co-localização não foram ativadas, isto é, os aspectos responsáveis por tal funcionalidade não foram compilados. Na segunda seqüência de invocações, otimizações de co-localização diretas foram então ativadas. A Figura 1 apresenta o número de chamadas por milisegundo em ambos os casos.

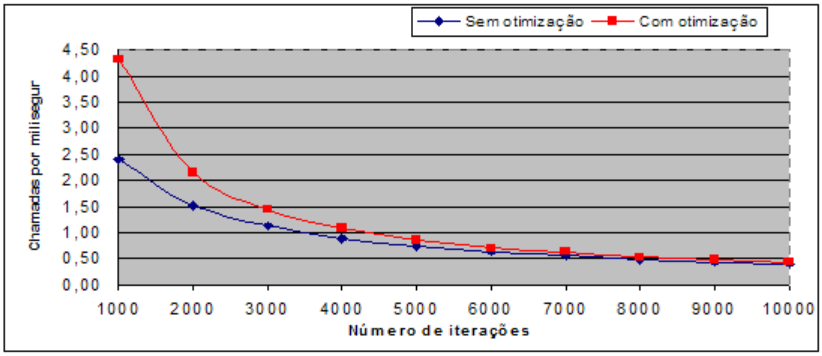

Figura 1. Chamadas/mseg no experimento realizado, com e sem otimizações de co-localização ativadas.

Como pode ser verificado na Figura 1, para um número menor de iterações o ganho obtido com otimizações de co-localização é significativo. Para 1000 iterações, chega-se a executar quase o dobro de chamadas por milisegundo quando se ativa a co-localização (4.32 contra 2.39 chamadas/mseg). Esta diferença vai decrescendo à medida

que se aumenta o número de iterações. Para dez mil iterações, chega-se praticamente a um empate (0.43 contra 0.40 chamadas/mseg). Este resultado é compreensível, já que para um número muito grande de iterações, o *overhead* de comunicação gerado pelo *middleware* passa a ser insignificante frente ao tempo de execução do método remoto.

4 Trabalhos Relacionados

Otimizações de co-localização já foram propostas para diversos sistemas de *middleware*, incluindo CORBA [4], ICE [3] e COM/DCOM [1]. Implementações de CORBA oferecem otimizações tanto na forma direta como utilizando *proxies*. Otimizações via *proxy* em CORBA são conhecidas como otimizações padrão [8].

Zhang e Jacobsen, por meio de análises de implementações de CORBA, quantificaram o grau de espalhamento e entrelaçamento de código existente na implementação de diversas funcionalidades comuns em plataformas de *middleware*, incluindo otimizações de co-localização [9]. Em [10], estes mesmos autores propõem uma metodologia para decomposição horizontal de sistemas de *middleware*, a qual serviu de inspiração para o projeto de AspectJRMI. A fim de validar o conceito de decomposição horizontal, os autores realizaram uma refatoração do sistema ORBacus [5], na qual otimizações de co-localização foram implementadas por meio de aspectos. Como esperado, a refatoração proposta preserva a interface de programação de CORBA. Por outro lado, o projeto de AspectJRMI não foi restrito por uma interface de programação pré-definida. De forma similar a Java RMI, AspectJRMI não requer uma linguagem própria de definição de interfaces, nem extensões na linguagem Java ou em sua máquina virtual. Em vez disso, o sistema procura tirar proveito dos conceitos de programação orientada por aspectos fornecidos por AspectJ para disponibilizar um sistema de *middleware* onde o usuário pode selecionar os serviços que deseja efetivamente incorporar em sua aplicação.

5 Conclusões

Acredita-se que a solução proposta para otimizações de co-localização em AspectJRMI apresenta os seguintes benefícios:

- **Modularização:** Assim como ocorre com os demais serviços providos por AspectJRMI, o código responsável por implementar otimizações de co-localização encontra-se devidamente modularizado em aspectos.
- **Reconfiguração e Customização:** Coerente com a filosofia de projeto de AspectJRMI, cabe ao usuário do sistema determinar se deseja ou não ativar otimizações de co-localização. Caso queira desativar tais otimizações, basta não combinar os aspectos responsáveis por sua implementação.
- **Obliviousness:** Na solução proposta, cabe ao programador indicar os objetos remotos para os quais deseja ativar o monitoramento de co-localização, declarando para isso que suas classes implementam uma interface derivada de `Collocated` (ou de `CollocatedProxy`). Esta exigência permite delimitar melhor os *pointcuts* responsáveis por realizar este monitoramento. Por outro lado, a mesma diminui o grau de *obliviousness* provido pela solução. Caso seja importante uma

implementação de co-localização que seja totalmente transparente ao código alvo, recomenda-se eliminar a interface `Collocated` e, nos `pointcuts` descritos na Seção 2.2, utilizar `Remote` no lugar de `Collocated`.

- **Desempenho:** Conforme mostrado na Seção 3, os ganhos de desempenho gerados por otimizações de co-localização do tipo direto são significativos – em certos casos próximos a 100%. No entanto, estes ganhos ocorrem principalmente em métodos onde o custo de comunicação é representativo, quando comparado ao custo de execução dos mesmos.

Como trabalho futuro, pretende-se avaliar também os ganhos de desempenho obtidos com otimizações que preservam a semântica de chamadas remotas, descritas na Seção 2.3. Pretende-se também investigar a incorporação de novas funcionalidades transversais em AspectJRMI, como, por exemplo, invocação dinâmica de métodos, persistência etc.

Agradecimentos: Este trabalho originou-se de um projeto de pesquisa financiado pela FAPEMIG (processo EDT 1906/03 - Programa de Infra-estrutura para Jovens Doutores).

Referências

[1] Don Box. *Essential COM*. Addison Wesley, 1997.

[2] Marco Tulio de Oliveira Valente, Fabio Tirelo, Diana Campos Leao, and Rodrigo Palhares. An aspect-oriented communication middleware system. In *International Symposium on Distributed Objects and Applications*, LNCS. Springer-Verlag, October 2005.

[3] Michi Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66–75, 2004.

[4] Object Management Group. The common object request broker: Architecture and specification revision 3.0.2, December 2002.

[5] Orbacus. http://www.orbacus.com.

[6] Fernando Magno Pereira, Marco Túlio Valente, Roberto Bigonha, and Mariza Bigonha. Arcademis: A framework for object oriented communication middleware development. *Software Practice and Experience*, 2005. To appear.

[7] Douglas Schmidt and Chris Cleeland. Applying patterns to develop extensible and maintainable ORB middleware. *IEEE Communications*, 37(4):54 – 63, 1999.

[8] Douglas Schmidt, N. Wang, and S. Vinoski. Object interconnections collocation optimizations for CORBA. *SIGS C++ Report*, 10(9), 1999.

[9] Charles Zhang and Hans-Arno Jacobsen. Refactoring middleware with aspects. *IEEE Transactions Parallel and Distributed Systems*, 14(11):1058–1073, 2003.

[10] Charles Zhang and Hans-Arno Jacobsen. Resolving feature convolution in middleware systems. In *19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 188–205. ACM Press, 2004.