Um Sistema de Chamada Remota de Métodos Orientado por Aspectos

**Marco Túlio de Oliveira Valente, Diana Campos Leão,
Rodrigo Palhares, Fabio Tirelo**

Departamento de Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais
{mtov,ftirelo}@pucminas.br

Resumo. *Neste artigo, apresenta-se um sistema de chamada remota de métodos, chamado AspectJRMI, que, por meio de conceitos de orientação por aspectos, disponibiliza um sistema de middleware configurável em tempo de compilação. Em AspectJRMI, programadores podem agregar funcionalidades transversais ao núcleo do sistema, o qual oferece basicamente um serviço de chamadas síncronas de métodos remotos. AspectJRMI disponibiliza aspectos para implementação dos seguintes interesses transversais: chamadas oneway, chamadas assíncronas, objetos interceptadores, combinadores de serviços e passagem de parâmetros por valor-resultado. No artigo, descreve-se a interface de programação de AspectJRMI e os primeiros resultados experimentais de uso do sistema.*

1. Introdução

Aplicações distribuídas têm sido preferencialmente construídas usando-se plataformas de *middleware* orientadas por objetos, como Java RMI [23] e CORBA [13]. No entanto, tais plataformas se transformaram nos últimos anos em sistemas complexos, inflexíveis e sobrecarregados de serviços que nem sempre são usados em todas as aplicações. Por exemplo, o número de classes do sistema JacORB – uma versão de CORBA para Java – cresceu cerca de 50% em quatro anos [27]. Devido a essa complexidade crescente, existe atualmente todo um esforço de pesquisa no sentido de se desenvolver plataformas de *middleware* que sejam abertas e configuráveis [19, 12, 16, 11]. Um dos objetivos deste esforço é permitir que programadores possam selecionar, dentre a variedade de serviços disponíveis, apenas aqueles que são de fato necessários no desenvolvimento de uma aplicação distribuída em particular. Serviços que não foram selecionados não devem causar nenhum impacto no sistema de *middleware* resultante dessa configuração, seja no tempo de execução, no consumo de memória ou no tamanho final do sistema.

Conceitos tradicionalmente utilizados no desenvolvimento de sistemas orientados por objetos, como padrões de projeto, *frameworks*, reflexão computacional e componentes, certamente podem contribuir para a implementação de sistemas de *middleware* que atendam parcialmente aos requisitos mencionados. No entanto, análises recentes mostram que determinados serviços providos por plataformas de *middleware* são transversais a diversos módulos do núcleo dessas plataformas [25, 26, 27]. Por exemplo, um estudo realizado com três implementações de CORBA, mostrou que cerca de 50% das classes

destes sistemas devem coordenar a implementação de dois interesses diferentes e 10% dessas classes coordenam três ou mais interesses [26]. Como resultado, a tarefa de remover ou acrescentar um serviço nestes sistemas não é trivial, notadamente quando se faz uso apenas de conceitos tradicionais em orientação por objetos.

Como exemplo de interesses transversais (*crosscutting concerns*) ao núcleo de um sistema de chamada remota de métodos, podem-se citar os seguintes:

- Interceptadores de chamadas remotas: objetos interceptadores possibilitam adicionar funcionalidades extras no fluxo normal de processamento de uma chamada remota de métodos. São usados, por exemplo, para implementar requisitos como *logging*, autenticação, caches, transações etc. Em [26], mostra-se que o grau de espalhamento da implementação de interceptadores em CORBA pode atingir 13% das classes deste tipo de *middleware* (isto é, 13% das classes do sistema possuem código relacionado com a implementação dessa funcionalidade). Neste trabalho, interceptadores são chamados de decoradores de referências remotas.
- Chamadas Oneway. Chamadas remotas *oneway* possuem retorno `void` e são processadas usando-se uma estratégia do tipo *best-effort*. Neste tipo de chamada, a *thread* cliente prossegue sua execução assim que a chamada é enviada para o *middleware*. Além disso, nenhuma política de tratamento e sinalização de erros de comunicação é adotada. Também em [26], descreve-se que o grau de espalhamento da implementação de chamadas *oneway* em CORBA pode chegar a quase 11% das classes do sistema.
- Chamadas Assíncronas. Assim como ocorre com chamadas *oneway*, chamadas assíncronas retornam o controle ao cliente assim que a chamada é entregue ao sistema de *middleware*, isto é, o cliente não fica bloqueado esperando o resultado da chamada. No entanto, ao contrário de chamadas *oneway*, chamadas assíncronas retornam um objeto do tipo `Future`, o qual é usado por clientes para obter o resultado da chamada quando este estiver disponível. Apesar de não se ter encontrado na literatura uma avaliação experimental do grau de espalhamento decorrente da implementação de chamadas assíncronas, acredita-se que o mesmo deve estar próximo daquele existente na implementação de chamadas *oneway*.
- Combinadores de Serviços. Combinadores de serviços foram propostos originalmente para definir procedimentos para tratamento de falhas comuns na recuperação de páginas da *Web* [3]. Neste artigo, combinadores de serviços são usados para especificar que objetos remotos devem ser acessados seqüencialmente (para incrementar tolerância a falhas), concorrentemente (para aumentar o desempenho da aplicação) ou não-deterministicamente (para prover distribuição de carga). Como exemplo de *middleware* que disponibiliza combinadores de serviços pode-se citar o sistema Aries [17].
- Passagem de Parâmetros por Valor-Resultado. Em aplicações distribuídas, passagem de parâmetros por valor-resultado é normalmente indicada como uma alternativa a passagem por referência. Em plataformas de *middleware*, a implementação de passagem por valor-resultado é um requisito transversal, já que atinge pelo menos os seguintes componentes da arquitetura: *stubs*/*skeletons* e a camada de troca de mensagens.

Neste artigo, apresenta-se um sistema de chamada remota de métodos, chamado AspectJRMI, que, por meio de conceitos de orientação por aspectos [10], disponibiliza

um sistema de *middleware* configurável em tempo de compilação. Em AspectJRMI, programadores podem agregar funcionalidades extras ao núcleo do sistema, o qual oferece basicamente um serviço de chamada síncrona de métodos remotos. Via de regra, estas funcionalidades, conforme afirmado acima, tendem a ser transversais aos módulos deste núcleo e, por este motivo, são implementadas em AspectJRMI por meio de aspectos. O objetivo é permitir que as mesmas possam ser incluídas e removidas facilmente do sistema de *middleware*. Assim, o *middleware* que apóia o desenvolvimento de uma determinada aplicação distribuída – e que foi gerado por meio de um compilador de aspectos – possui apenas o núcleo mínimo mais as funcionalidades efetivamente demandadas por essa aplicação. O núcleo de AspectJRMI é constituído pelo sistema RME [15], um *middleware* orientado por objetos bastante simples, desenvolvido originalmente para a plataforma J2ME/CLDC. Além deste núcleo, AspectJRMI disponibiliza aspectos para implementar os interesses transversais relacionados anteriormente. Estes aspectos são implementados no sistema usando-se a linguagem AspectJ [9].

O restante deste artigo encontra-se organizado conforme descrito a seguir. A Seção 2 descreve brevemente o núcleo do sistema, composto pelo sistema RME. Na Seção 3, descreve-se a API de programação de AspectJRMI, a qual é formada por um conjunto de classes e aspectos que permitem implementar de forma modular os requisitos transversais relacionados anteriormente. A Seção 4 apresenta alguns resultados experimentais de uso do sistema. A Seção 6 conclui o artigo, avaliando as principais contribuições do projeto de AspectJRMI e relacionando possibilidades de trabalhos futuros.

2. Núcleo do Sistema

O núcleo de AspectJRMI é constituído pelo sistema RME, o qual é um *middleware* orientado por objetos desenvolvido originalmente para a plataforma J2ME/CLDC. A plataforma RME é composta pelos seguintes componentes: servidor (cujos arquivos .class ocupam cerca de 68 KB), cliente (cujos arquivos .class ocupam cerca de 37 KB), um serviço de nomes e um gerador de *stubs* e *skeletons* (chamado `rmec`).

Em relação a Java RMI, o uso de RME como núcleo de AspectJRMI apresenta como principal vantagem o fato de RME adotar uma arquitetura aberta, que utiliza extensivamente diversos padrões de projeto [6, 20], tais como: Fábricas, *Singletons*, Fachadas, Decoradores, Estratégias, *Proxy*, Adaptadores, *Acceptor-Connector*, *Reactor*, *Flyweight*, dentre outros. Com isso, diversos componentes que implementam interesses não-transversais podem ser configurados em RME. Como exemplo, temos os seguintes componentes: protocolo de comunicação (TCP, UDP etc), protocolo de troca de mensagens (JRMP, IIOP, RMEP etc), protocolo de serialização (padrão de Java, SOAP etc), política de *threads* do servidor, dentre outros.

Por questões de espaço, omite-se deste artigo uma descrição mais detalhada de RME, a qual é realizada em [15]. No entanto, pode-se afirmar que programadores com alguma prática em Java RMI são capazes de utilizar o sistema sem maiores esforços.

3. AspectJRMI: Funcionalidades Extras

Além de um serviço síncrono de chamada remota de métodos, provido pelo núcleo do sistema, AspectJRMI disponibiliza de forma modular e seletiva as seguintes funcionalida-

des: combinadores de serviços, decoradores de invocação, chamadas *oneway*, chamadas assíncronas e passagem de parâmetros por valor-resultado.

3.1. Combinadores de Serviços

Suponha que C_1 e C_2 sejam duas chamadas remotas de métodos, associadas respectivamente aos serviços S_1 e S_2. AspectJRMI permite que estas chamadas sejam compostas usando-se os seguintes combinadores de serviços:

- $C_1 > C_2$ (execução alternativa): primeiro, a chamada $C1$ é executada; caso ela falhe, executa-se $C2$; caso $C2$ também falhe, a chamada composta falha. O combinador $>$ é usado para prover tolerância a falhas.
- $C_1 \ ? \ C_2$ (escolha não-determinística): uma das duas chamadas é não-deterministicamente escolhida para ser executada. O combinador $?$ é usado para prover distribuição de carga. Em caso de falha da chamada escolhida, a outra chamada é executada. Se ambas chamadas falharem, a chamada composta falha.
- $C_1 \mid C_2$ (execução concorrente): as duas chamadas são concorrente executadas. O primeiro resultado obtido é considerado como sendo o resultado da chamada composta; o segundo resultado é descartado. Se as duas chamadas falharem, a chamada composta falha. O combinador $\mid$ é usado para otimizar o tempo de resposta na execução de uma chamada remota (ao custo de invocá-la concorrentemente em dois servidores).

Combinadores de serviços são associados a referências remotas usando-se as seguintes classes[1]:

```
class SimpleRemoteRef extends RemoteRef {
  public SimpleRemoteRef(Remote endpoint);
}
class StructuredRemoteRef extends RemoteRef {
  public StructuredRemoteRef(char op,RemoteRef r1,RemoteRef r2);
}
```

A classe `RemoteRef` é uma classe abstrata que representa no sistema uma referência remota. A classe `SimpleRemoteRef` denota uma referência remota padrão, sem nenhum combinador de serviço associado. A classe `StructuredRemoteRef` é usada para denotar uma referência remota à qual se associa um combinador de serviço.

Um aspecto abstrato de nome `RemoteAspect` é usado para associar combinadores de serviços a referências remotas

```
abstract aspect RemoteAspect {
   protected abstract pointcut RemoteCalls();
   protected abstract RemoteRef getRemoteRef();
  }
```

Este aspecto é abstrato já que não define as chamadas do programa ao qual será associado (o que é feito definindo o *pointcut* abstrato `RemoteCalls`), nem a referência a ser usada para executar tais chamadas (o que é feito implementando o método abstrato `getRemoteRef`). Estes parâmetros devem ser definidos em seus subaspectos.

Exemplo: Seja um serviço remoto definido pela seguinte interface:

[1]Neste artigo, apresenta-se apenas a interface das classes e aspectos propostos em AspectJRMI. Particularmente, no caso de aspectos, omite-se o código de *advices*.

```
interface Hello extends Remote {
  String sayHello() throws RemoteException;
  String sayHello(String name) throws RemoteException;
}
```

Seja o seguinte programa cliente do serviço definido por esta interface:

```
class HelloClient {
   Hello server= ORB.InitRemoteRef();
   String s1= server.sayHello();
   String s2= server.sayHello("Bob");
   ......
}
```

Em AspectJRMI, o método `ORB.InitRemoteRef()` é usado para inicializar referências remotas que possuem combinadores de serviços associados.

Suponha que o programador queira associar o seguinte combinador de serviço à chamada de métodos remotos do tipo `Hello`: inicialmente, direcione todas as chamadas remotas para o servidor de nome `helloSrv` da estação `skank.inf.pucminas.br`. Caso estas chamadas falhem, faça uma segunda tentativa no servidor de nome `helloSrv` da estação `patofu.inf.pucminas.br`. Este combinador deve ser associado a todas as chamadas de métodos de serviços do tipo `Hello`, executadas no interior da classe `HelloClient`.

Para tanto, o programador deverá criar o seguinte subaspecto do aspecto `RemoteAspect`:

```
 1: aspect HelloClientAspect extends RemoteAspect {
 2:   private RemoteRef ref;
 3:   protected pointcut RemoteCalls(): within(HelloClient) &&
 4:                                     call(* Hello.*(..));
 5:
 6:   HelloClientAspect(){
 7:     String s1= "skank.inf.pucminas.br/helloSrv";
 8:     String s2= "patofu.inf.pucminas.br/helloSrv";
 9:     ref= new StructuredRemoteRef('>',
10:            new SimpleRemoteRef(Naming.lookup(s1)),
11:            new SimpleRemoteRef(Naming.lookup(s2)));
12:   }
13:   protected RemoteRef getRemoteRef(){
14:     return ref;
15:   }
16: }
```

Na linha 3, especifica-se que o combinador implementado neste aspecto deve ser associado a chamadas de métodos da interface `Hello`, realizadas no interior da classe `HelloClient`. Nas linhas 7 a 11, cria-se uma chamada composta, usando-se o combinador de serviço $>$ (invocação em caso de falha).

3.2. Decoradores de Referências Remotas

Decoradores de referências remotas permitem agregar processamento extra no fluxo de execução de chamadas remotas. Decoradores dispensam o uso de herança para estender

a funcionalidade da classe `RemoteRef` e permitem encadear diversas tarefas que devem ser executadas durante uma invocação remota.

Um decorador é especificado pela seguinte classe:

```
class RemoteRefDecorator extends RemoteRef {
   public RemoteRefDecorador (RemoteRef ref);
}
```

O construtor desta classe recebe como parâmetro um objeto denotando a referência remota a ser decorada. Para criar um decorador de invocação, basta estender a classe `RemoteRefDecorator`. Esta nova classe deve prover a funcionalidade extra que demandou a criação do decorador. No sistema, já se encontram implementados os seguintes decoradores: `Cache` (usado para armazenar em um cache os resultados de chamadas remotas idempotentes), `Log` (usado para registrar chamadas remotas) e `Timer` (usado para especificar um tempo máximo para conclusão de uma chamada remota, após o qual ativa-se uma exceção).

Exemplo: O código abaixo associa um decorador do tipo `Log` a cada uma das chamadas simples pertencentes à chamada composta mostrada no exemplo anterior.

```
ref = new StructuredRemoteRef('>',
        new Log (SimpleRemoteRef(Naming.lookup(s1))),
        new Log (SimpleRemoteRef(Naming.lookup(s2))));
```

3.3. Chamadas Oneway

Em chamadas remotas do tipo *oneway*, o controle retorna ao cliente assim que a chamada é recebida pela camada de *middleware*. A execução do cliente e do método remoto ocorrem de forma assíncrona. Chamadas *oneway* não retornam valor (isto é, o resultado é `void`) e não ativam exceções remotas.

O aspecto abstrato `OneWayAspect` é usado para definir chamadas do tipo *oneway*. Basicamente, este aspecto contém um *pointcut* abstrato, de nome `OneWayCalls`, o qual é usado para especificar as chamadas do programa que serão despachadas usando-se este tipo de semântica.

```
abstract aspect OneWayAspect {
   protected abstract pointcut OneWayCalls();
}
```

Exemplo: O aspecto a seguir define que chamadas do método `void sayHello()` da interface `Hello` e chamadas de todos métodos da interface `A` que retornam `void` deverão ser processadas usando-se uma semântica do tipo *oneway*.

```
aspect MyOneWayAspect extends OneWayAspect {
   protected pointcut OneWayCalls():
      call(void Hello.sayHello()) || call(void A.*(..));
}
```

3.4. Chamadas Assíncronas

Para especificar chamadas assíncronas, o programador deve usar os recursos de declarações intertipos de AspectJ. Suponha que `f` seja um método de uma interface remota `A`.

Caso seja necessário chamar este método assincronamente, basta acrescentar em `A` uma implementação *default* e vazia para *f* da seguinte forma: `Future async_f(...){}`. Isto é, usa-se o prefixo `async_` para diferenciar a função como assíncrona. A classe `Future` possui um método `getResult()` que deve ser usado para se obter o resultado da chamada remota assíncrona. Caso esta chamada ainda não tenha sido concluída, a invocação de `getResult()` permanece suspensa. Caso a chamada assíncrona tenha falhado, a invocação de `getResult()` retorna uma exceção do tipo `RemoteException`.

Exemplo: O código a seguir mostra como o método `String sayHello(String)` da interface `Hello` pode ser invocado assincronamente. Inicialmente, deve-se acrescentar uma implementação vazia deste método, prefixada com `async_`, na interface `Hello`. Para isso, pode-se usar um aspecto aninhado da seguinte forma:

```
interface Hello extends Remote {
  String sayHello() throws RemoteException;
  String sayHello(String name) throws RemoteException;
  static aspect AsyncHello {
    Future async_sayHello(String Name) {}
  }
}
```

O código a seguir chama assincronamente o método `async_sayHello` e, posteriormente, usa o método `getResult()` para sincronização e obtenção do resultado dessa chamada.

```
Future ftr= server.async_sayHello("Bob");
.............
String res=(String) ftr.getResult();
```

3.5. Passagem por Valor-Resultado

Passagem por valor-resultado é especificada declarando-se que o parâmetro formal é do tipo `Holder`. A classe `Holder` funciona como um *wrapper* para o valor a ser copiado, o qual obrigatoriamente deve implementar a interface `Marshable`. Esta interface é usada em RME para indicar objetos que podem ser transmitidos por um canal de comunicação.

A classe `Holder` possui a seguinte interface:

```
class Holder {
  public Holder (Marshable o);
  public Marshable getValue();
  public void setValue(Marshable o);
}
```

Na chamada de um método remoto que possui um parâmetro do tipo `Holder`, o cliente deve instanciar um objeto deste tipo que encapsule o valor a ser passado, usar este objeto como parâmetro real e, após a chamada, extrair o resultado deste mesmo objeto. Não é permitido a um método que tenha um parâmetro `p` do tipo `Holder` alterar o valor da referência armazenada neste parâmetro (por exemplo, fazendo `p= new Holder()`).

O programador de um sistema que faz uso de passagem por valor-resultado não necessita criar nenhum aspecto ou fazer uso de qualquer outro conceito de orientação por

aspectos. A implementação do sistema faz uso do seguinte *pointcut* para capturar todas as chamadas de métodos que incluem um parâmetro do tipo `Holder`:

```
pointcut callbyValueResult():  call(* *(..,Holder,..)) &&
                              !(within(*_Skeleton));
```

Por meio deste *pointcut*, aspectos internos de AspectJRMI tratam de implementar uma semântica do tipo valor-resultado.

Exemplo: O exemplo a seguir mostra a implementação de um método com dois parâmetros passados por valor-resultado, bem como o código que chama este método.

```
void foo(Holder h1, Holder h2) {
  Date d= (Date) h1.getValue();
  d.setDate(d.getDia()+1,12,2004);
  h2.setValue(new Date(28,2,2005));
}
.......
Holder h1= new Holder(new Date(17,2,2005));
Holder h2= new Holder(new Date());
foo(h1, h2);
(Date (h1.getValue())).print();   // imprime 18/12/2004
(Date (h2.getValue())).print();   // imprime 28/02/2005
```

AspectJRMI oferece também passagem de parâmetros por resultado, isto é, na entrada do método nada é copiado; na sua saída, o valor restante no parâmetro formal é copiado para o respectivo parâmetro de chamada. Passagem por resultado é especificada declarando-se que o parâmetro formal é do tipo `ResultHolder`.

4. Resultados Experimentais

A fim de validar preliminarmente a concepção e o projeto de AspectJRMI, foram realizados alguns experimentos com a implementação atual do sistema. O objetivo foi levantar números que pudessem servir de argumentos favoráveis à tese de que AspectJRMI permite a especialização de plataformas de *middleware* contendo apenas funcionalidades efetivamente demandadas por um aplicação distribuída. Em outras palavras, números que mostrem que em AspectJRMI usuários somente "pagam" por funcionalidades da plataforma que efetivamente são utilizadas em suas aplicações distribuídas. Por este motivo, todos os resultados medidos nos experimentos descritos a seguir se referem ao tamanho do código do *middleware* gerado e/ou da aplicação distribuída em uso.

Os programas de teste utilizados nos experimentos descritos a seguir foram implementados usando-se o JDK 1.4 e dois compiladores de aspectos diferentes: `ajc` (versão 1.5.0) e `abc` (versão 1.0.2). O primeiro compilador é padrão de AspectJ; o segundo é um compilador que inclui uma série de otimizações e oportunidades de extensões [1].

Conforme afirmado na Seção 2, o núcleo do sistema possui 37 KB para aplicações que são apenas clientes de serviços remotos e 68 KB para aplicações que são clientes e servidoras. Os aspectos e classes internos de AspectJRMI ocupam 50 KB. Existem ainda 41 KB do *run-time* de AspectJ (arquivo `aspectjrt.jar`). Estes valores são sumarizados na Tabela 1.

Componente	Cliente	Servidor
Núcleo RME	37	68
AspectJRMI	50	50
AspectJ	41	41
Total	128	159

Tabela 1: Tamanho (em KB) do núcleo e das extensões de AspectJRMI

Além do "custo fixo" descrito na Tabela 1, existe ainda o custo do processo de *weaving*, através do qual os aspectos que implementam as funcionalidades extras providas por AspectJRMI são incorporados a uma aplicação base. Para dimensionar este custo, em termos de acréscimo no código, foram desenvolvidas as seguintes aplicações distribuídas:

- P1: Aplicação cliente chama, usando uma semântica do tipo *oneway*, um método remoto passando como parâmetro uma *string* com 16 caracteres e um vetor de 64 inteiros.
- P2: Aplicação cliente chama assincronamente um método remoto que possui dois inteiros como parâmetros e que retorna uma *string*.
- P3: Aplicação cliente chama um método remoto que possui dois vetores de inteiros como parâmetros e que retorna `void`. Os vetores têm 128 e 32 inteiros, respectivamente. A chamada é realizada tendo como alvo uma referência remota a qual se associa um combinador do tipo `?` (escolha não-determinística).
- P4: Aplicação cliente chama um método remoto passando uma *string* com 32 caracteres e dois inteiros por valor-resultado.

As aplicações acima foram compiladas com os compiladores `ajc` e `abc`. Para fins de comparação, as aplicações P1 e P2 foram modificadas de forma a adotar uma semântica síncrona padrão para a chamada remota. A aplicação P3 foi modificada de forma a não utilizar um combinador de serviço. Por fim, a aplicação P4 foi modificada de forma a utilizar passagem por valor. Todas as aplicações modificadas foram então compiladas usando o compilador `javac`, padrão do ambiente JDK.

A Tabela 2 sumariza os resultados obtidos. A coluna *weaving* descreve o componente da aplicação que foi alvo do processo de *weaving*, isto é, se o compilador de aspectos instrumentalizou apenas o componente cliente da aplicação (programas P1, P2 e P3) ou os componentes cliente e servidor (programa P4). As colunas `ajc` e `abc` mostram o tamanho final, em KB, dos componentes da aplicação que foram alvos do processo de *weaving*. A coluna `javac` mostra o tamanho destes mesmos componentes na aplicação modificada para não usar aspecto, conforme descrito no parágrafo anterior. Por fim, a última coluna, `abc-javac`, apresenta a diferença de tamanho entre os códigos gerados por estes dois compiladores.

Os números apresentados na Tabela 2 permitem inicialmente afirmar que o compilador `ajc` introduz um *overhead* de espaço considerável. Certamente, `ajc` foi projetado e implementado tendo como principal objetivo servir como uma "prova de conceito" para os recursos de AspectJ. Assim, o compilador abdica de diversas otimizações importantes. Particularmente, a implementação de *advices* do tipo `around`, extensivamente empregados em AspectJRMI, pode dar origem a uma explosão no tamanho código gerado por este

	Uma chamada	Weaving	ajc	abc	javac	abc-javac
P1	Oneway	Cliente	4.73	2.46	0.84	1.62
P2	Assíncrona	Cliente	9.01	4.24	1.86	2.38
P3	Combinador ?	Cliente	5.92	3.21	0.81	2.4
P4	Valor-Resultado	Cliente/Servidor	7.77	4.03	1.82	2.21

Tabela 2: Tamanho (em KB) dos experimentos (para uma chamada remota)

compilador. Mais recentemente, com a popularização de orientação por aspectos e, particularmente de AspectJ, a otimização do processo de *weaving* se transformou em um tema de pesquisa relevante [2]. Neste sentido, considera-se que o compilador `abc` já introduz melhorias consideráveis no processo de *weaving*.

Conforme pode ser comprovado na Tabela 2, usando-se o compilador `abc` o *overhead* de espaço resultante da aplicação dos aspectos disponibilizados em AspectJRMI varia de 1.62 Kbytes (para o programa P1) a 2.4 Kbytes (para o programa P3). Ressalte-se que este *overhead* é para uma única chamada remota, a qual no entanto agrega uma funcionalidade extra (*oneway*, no caso de P1; e um combinador de serviço no caso de P4).

Na Tabela 3 mostra-se o tamanho do código para versões modificadas dos programas P1 a P4. Em vez de uma única chamada remota, estes programas realizam agora 100 chamadas remotas seqüencialmente. Como pode ser verificado nesta tabela, no caso do compilador `abc`, o *overhead* de espaço reduz-se para cerca de 100 *bytes* por chamada remota.

Acredita-se que este *overhead* seja aceitável. Implementações de CORBA para ambientes corporativos, por exemplo, chegam a ter 10 MBytes, como é o caso do sistema JacORB [7]. Já implementações de CORBA para dispositivos com recursos computacionais limitados requerem sempre mais de um megabyte. O sistema ORBit2 [14], por exemplo, possui cerca de 2 MB. Existem implementações de CORBA de tamanhos menores que os citados – como é o caso do sistema UIC-CORBA [18], o qual possui cerca de 48.5 KB na plataforma Windows CE e 100 Kb na plataforma Windows 2000. No entanto, este sistema inclui apenas um serviço básico de chamada remota de métodos. Isto é, ele deve ser comparado com o núcleo de AspectJRMI.

	100 chamadas	Weaving	ajc	abc	javac	(abc-javac)/100
P1	Oneway	Cliente	42.7	11.5	2.07	0.09
P2	Assíncrona	Cliente	51.8	14.4	3.11	0.11
P3	Combinador ?	Cliente	49.1	12.2	2.05	0.10
P4	Valor-Resultado	Cliente/Servidor	96.5	13.9	3.27	0.10

Tabela 3: Tamanho (em KB) dos experimentos (para cem chamadas remotas)

5. Trabalhos Relacionados

Diversos conceitos tradicionais em orientação por objetos já foram empregados na construção de plataformas de *middleware* com o objetivo de prover sistemas abertos, con-

figuráveis e adaptáveis. Reflexão computacional, por exemplo, é o conceito central de sistemas como OpenORB [4] e dynamicTAO [12]. No entanto, APIs reflexivas, via de regra, oferecem um baixo de nível de abstração. Sistemas como Quarterware [21] e Arcademis [16] são centrados na idéia de *frameworks*. Estes sistemas podem ser vistos como plataformas de *middleware* semi-acabadas, as quais são configuradas pelos usuários a fim de se obter um sistema final personalizado à sua necessidade. No entanto, *frameworks* suportam reconfigurações de maior granularidade e mais globais do que aquelas possíveis por meio de orientação por aspectos. TAO [19] é um *middleware* que utiliza extensivamente padrões de projeto com o objetivo de prover opções de reconfiguração, as quais, no entanto, também possuem um maior nível de granularidade.

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* [25, 26]. Mostraram ainda que uma modularização adequada de tais funcionalidades pode ser obtida usando-se orientação por aspectos. Em [27], estes mesmos autores propõem uma metodologia para decomposição horizontal de sistemas de *middleware*. Basicamente, defendem que o núcleo do sistema deve ser implementado usando-se tecnologias tradicionais de modularização e que orientação por aspectos deve ser usada no caso de requisitos transversais. Esta mesma estratégia de desenvolvimento foi adotada na implementação de AspectJRMI. Em seu trabalho mais recente, Zhang e Jacobsen descrevem o projeto de um *middleware* onde usuários podem escolher diversos componentes da arquitetura final do sistema [24]. Para isso, são propostas três fases: especificação dos requisitos da aplicação distribuída e das restrições de seu ambiente de execução; validação da arquitetura proposta e síntese da arquitetura final do sistema.

JBossAOP [8] é um versão orientada por aspectos do sistema JBoss. O sistema utiliza o conceito de anotações de Java 1.5 para implementar diversos interesses, tais como chamadas *oneway*, chamadas assíncronas, transações, persistência etc. Basicamente, anotações são usadas ao longo do código como "ganchos sintáticos", aos quais posteriormente serão adicionados aspectos. Neste caso, questiona-se se este emprego de anotações não estaria transformando o conceito em um requisito transversal, visto que anotações podem estar espalhadas em grande parte do código de uma aplicação. Preferencialmente, anotações devem ser usadas para se especificar propriedades funcionais inerentes ao elemento que está sendo anotado, como, por exemplo, que uma operação é idempotente ou que uma operação não altera o estado do objeto alvo.

Pereira et al. [17] descrevem uma linguagem de domínio específico para especificação de táticas, isto é, de procedimentos para adaptar e customizar uma aplicação distribuída a um ambiente de rede particular. Esta linguagem inclui suporte a combinadores de serviços e objetos interceptadores. Descrevem também como esta linguagem foi incorporada a um sistema de *middleware* denominado Aries. A solução proposta consiste na ampliação do papel do compilador de *stubs* do sistema Aries. A partir de interfaces remotas e de um arquivo de especificação de táticas, este compilador gera *stubs* que implementam as táticas especificadas. Em relação à solução de AspectJRMI, a implementação de táticas em Aries não dispõe do rico conjunto de operadores existentes em AspectJ para especificação de pontos de junção. Particularmente, uma vez associadas a um método de uma interface remota, táticas são sempre ativadas quando da chamada deste método. Não

é possível definir pontos de junção baseados na estrutura léxica do programa ou no seu fluxo de execução. Da mesma maneira, não é possível definir *pointcuts* e associá-los a uma mesma definição de tática

Orientação por aspectos já foi aplicada com sucesso na modularização de requisitos não-funcionais típicos de aplicações distribuídas. Soares, Laureano e Borba descrevem uma experiência de uso de AspectJ na restruturação de uma aplicação distribuída baseada em Java RMI [22]. O objetivo foi modularizar aspectos de distribuição e persistência deste sistema.

Uma primeira versão de AspectJRMI foi apresentada em [5]. Nesta versão, o sistema incluía apenas combinadores de serviços e decoradores de referências remotas.

6. Conclusões

Descreveu-se neste artigo um sistema de chamada remota de métodos composto por um núcleo (que disponibiliza um serviço básico de comunicação síncrona) e por um conjunto de aspectos que agregam as seguintes funcionalidades a este núcleo: chamadas *oneway*, chamadas assíncronas, objetos interceptadores, combinadores de serviços e passagem de parâmetros por valor-resultado. Por limitações de espaço, omitiu-se deste artigo uma descrição mais detalhada sobre a implementação das classes e aspectos internos de AspectJRMI. Espera-se abordar este tópico em um futuro artigo.

Considera-se que o emprego de uma linguagem orientada por aspectos no desenvolvimento de AspectJRMI viabilizou as seguintes contribuições em relação ao projeto de sistemas de chamada remota de métodos tradicionais:

- Modularização: o código responsável por implementar as funcionalidades extras disponibilizadas por AspectJRMI encontra-se devidamente modularizado em aspectos. Com isso, são evitados os problemas de espalhamento e entrelaçamento de código que caracterizam a implementação destes interesses em plataformas tradicionais de *middleware*.
- Reconfiguração e personalização: O serviço básico de comunicação disponibilizado por AspectJRMI, composto pelo sistema RME, ocupa apenas 37 KB na versão para clientes. Este núcleo é um sistema aberto, que permite reconfigurações dos protocolos de comunicação, troca de mensagens, serialização, dentre outros. Sobre este núcleo, AspectJRMI permite que programadores agreguem as funcionalidades descritas neste artigo apenas se as mesmas forem efetivamente demandadas em uma aplicação distribuída em particular. Assim, em vez de uma arquitetura fechada e monolítica, oferece-se um sistema de *middleware* que pode ser personalizado pelos seus usuários. Conforme mostrado na Seção 4, esta personalização não acresccenta um *overhead* considerável no tamanho final do sistema.

Este trabalho deve prosseguir em duas frentes. A primeira investigando a incorporação de novas funcionalidades transversais em AspectJRMI, como, por exemplo, a otimização de chamadas remotas no caso do objeto servidor estar localizado no mesmo espaço de endereçamento do objeto cliente (*collocation optimization*, na terminologia de CORBA), invocação dinâmica de métodos, tolerância a falhas etc. Em uma segunda

frente, pretende-se estender os experimentos da Seçao 4, de forma a comparar o desempenho em tempo de execução de AspectJRMI com outras plataformas de *middleware*, notadamente com implementações de CORBA.

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] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An Extensible AspectJ Compiler. In *4th International Conference on Aspect-Oriented Software Development*. ACM Press, 2005.

[2] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 2005.

[3] Luca Cardelli and Rowan Davies. Service Combinators for Web Computing. In *Conference on Domain-Specific Languages (DSL-97)*, pages 1–10. USENIX Association, October 1997.

[4] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 160–178. Springer-Verlag, 2001.

[5] Marco Túlio de Oliveira Valente, Rodrigo Palhares, and Fabio Tirelo. Especificação de Táticas para Invocação Remota de Métodos Usando Orientação por Aspectos. In *I Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos*, 2004.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[7] JacORB. http://www.jacorb.org. Última visita: fevereiro de 2005.

[8] JBossAOP. http://www.jboss.org/developers/projects/jboss/aop. Última visita: fevereiro de 2005.

[9] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, volume 2072, pages 327–355. Springer Verlag, 2001.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[11] Fabio Kon, Fábio Costa, Roy Campbell, and Gordon Blair. The Case for Reflective Middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[12] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer-Verlag, 2000.

[13] Object Management Group. The Common Object Request Broker: Architecture and Specification (version 2.4), October 2000.

[14] ORBit2. http://orbit-resource.sourceforge.net. Última visita: fevereiro de 2005.

[15] Fernando Magno Pereira, Marco Túlio Valente, Roberto Bigonha, and Mariza Bigonha. Chamada Remota de Métodos na Plataforma J2ME/CLDC. In *V Workshop de Comunicação sem Fio e Computação Móvel*, 2003.

[16] Fernando Magno Pereira, Marco Túlio Valente, Roberto Bigonha, and Mariza Bigonha. Arcademis: A Java-Based Framework for Middleware Development. In *XXII Simpósio Brasileiro de Redes de Computadores*, pages 539–552, May 2004.

[17] Fernando Magno Pereira, Marco Túlio Valente, Roberto Bigonha, and Mariza Bigonha. Tactics for Remote Method Invocation. *Journal of Universal Computer Science*, 10:824–842, July 2004.

[18] Manuel Román, Fabio Kon, and Roy Campbell. Reflective Middleware: From Your Desk to Your Hand. *Distributed Systems Online*, 2(5), July 2001.

[19] Douglas Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible and Maintainable ORB Middleware. *IEEE Communications*, 37(4):54 – 63, 1999.

[20] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley & Sons, 2000.

[21] Ashish Singhai, Aamod Sane, and Roy H. Campbell. Quarterware for Middleware. In *18th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1998.

[22] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *17th ACM Conference on Object-Oriented programming systems, languages, and applications*, pages 174–190. ACM Press, 2002.

[23] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232. USENIX, 1996.

[24] Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen. Towards Just-in-time Middleware Architectures. In *4th International Conference on Aspect-Oriented Software Development*. ACM Press, 2005.

[25] Charles Zhang and Hans-Arno Jacobsen. Quantifying Aspects in Middleware Platforms. In *2nd International Conference on Aspect-Oriented Software Development*, pages 130–139. ACM Press, 2003.

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

[27] 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.