Especificação de Táticas para Invocação Remota de Métodos Usando Orientação por Aspectos

Marco Túlio de Oliveira Valente Rodrigo Palhares Fabio Tirelo

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

Resumo

Aplicações distribuídas são freqüentemente construídas usando-se middleware. No entanto, os atuais sistemas de middleware usam um modelo de programação que tende a ignorar diferenças fundamentais que existem entre ambientes computacionais centralizados e distribuídos. Muitas vezes, a desconsideração dessas diferenças pode levar à implementação de aplicações distribuídas com baixo desempenho, não tolerantes a falhas, pouco escaláveis e inseguras. Táticas são procedimentos incorporados a plataformas de middleware para tratar eventos típicos de sistemas distribuídos. No entanto, táticas normalmente são requisitos transversais aos módulos de uma aplicação e, portanto, a implementação das mesmas dá origem a dois fenômenos indesejáveis: espalhamento e entrelaçcamento de código. Neste trabalho, apresenta-se um framework orientado por aspectos para implementação e modularização de táticas em aplicações distribuídas construídas usando-se Java RMI.

1. Introdução

Aplicações distribuídas são cada vez mais desenvolvidas usando-se plataformas de *middleware*, como CORBA [5] e Java RMI [11]. Esses sistemas têm como objetivo abstrair diversos detalhes de programação que são inerentes a ambientes distribuídos, tais como: primitivas e protocolos de comunicação em rede, *marshalling* e *unmarshalling* de dados, heterogeneidade dos ambientes, localização de serviços, sincronização etc. Desde a década de 90, sistemas de *middleware* sempre tiveram como principal objetivo unificar os modelos de programação distribuída e centralizada, por meio do oferecimento de abstrações que tornem o desenvolvimento de uma aplicação distribuída o mais parecido possível com o desenvolvimento de uma aplicação convencional. Assim, a palavra-chave no projeto destes sistemas sempre foi transparência. Por exemplo, sistemas de *middleware* orientados por objetos oferecem um alto grau de transparência no acesso a objetos remotos, isto é, objetos remotos são acessados com a mesma sintaxe utilizada no acesso a objetos locais.

No entanto, computação distribuída sempre foi caracterizada por diversos problemas que inexistem em um ambiente local de programação. Dentre eles, pode-se citar latência, falhas, concorrência e violações de segurança dos mais diversos tipos. Ressalte-se ainda que a diferença entre ambientes computacionais centralizados e distribuídos, em vez de diminuir, tem aumentado nos últimos anos com o surgimento de novos tipos de redes, como redes sem fio, redes par-a-par, grades, redes privadas virtuais etc.

Assim, os atuais sistemas de *middleware* têm sido criticados por adotar um modelo de programação que incentiva seus usuários a desconsiderar as diferenças fundamentais existentes entre ambientes distribuídos e centralizados [10]. Basicamente, nos atuais sistemas de *middleware*, serviços remotos são descritos por interfaces, por meio de uma notação bastante pa-

recida com aquela usada para se descrever serviços locais. Provedores de serviços implementam essas interfaces e clientes utilizam os serviços implementados como se fossem locais. Ou seja, em nenhuma das fases desse modelo de programação obriga-se um tratamento explícito de fenômenos que são característicos de ambientes distribuídos. A desconsideração desses fenômenos pode contribuir para criação de sistemas com baixo desempenho, pouco tolerantes a falhas e que não são escaláveis.

Pereira et al. [6] propõem que a especificação de um serviço distribuído por meio de uma linguagem para definição de interfaces seja acompanhada por um conjunto de táticas para lidar com eventos típicos de cenários de computação distribuída. São propostos dois tipos principais de táticas: combinadores de serviços[1] e decoradores de invocações. Combinadores de serviços são usados para especificar que servidores remotos devem ser acessados seqüencialmente (para incrementar tolerância a falhas), concorrentemente (para aumentar o desempenho da aplicação) ou não-deterministicamente (para prover balanceamento de carga). Decoradores de invocação são usados para introduzir funcionalidades extras no processamento de uma chamada remota, como *caches, buffers, logs* etc.

No entanto, táticas são requisitos transversais (*crosscutting concerns*), isto é, a implementação das mesmas usando técnicas tradicionais de programação orientada por objetos dá origem a dois problemas: espalhamento (*scattering*) e entrelaçamento (*tangling*) de código [9]. Suponha uma tática simples como a seguinte: nos módulos M_1, M_2, ..., M_n de um sistema, chamadas a métodos da interface remota A devem ser enviadas para o objeto remoto s_1 e, em caso de falha de comunicação com o mesmo, para o objeto remoto s_2. Assim, o código para implementação dessa tática estará distribuído ao longo dos módulos mencionados (espalhamento); além disso, esse código polui o código usado para implementar os requisitos funcionais da aplicação (entrelaçamento)[2].

Nesse trabalho, descreve-se um *framework* orientado por aspectos para implementação e modularização de táticas para acesso a métodos remotos em uma aplicação distribuída baseada em Java RMI. As táticas oferecidas por esse *framework* são todas relacionadas com a invocação de métodos remotos e, portanto, associam-se apenas a clientes de uma aplicação Java RMI. O *framework* proposto foi implementado em AspectJ [3].

O restante deste artigo está organizado como descrito a seguir. Na Seção 2, descreve-se o *framework* proposto no trabalho e apresentam-se as principais classes e aspectos que compõem o mesmo. Mostram-se também alguns exemplos de uso do *framework*. Na Seção 3, discute-se e avalia-se a solução proposta. A Seção 4 discute trabalhos relacionados com o *framework* apresentado. Por fim, a Seção 5 conclui este artigo.

2. *Framework* Proposto

O *framework* proposto neste trabalho inclui classes e aspectos para implementação de dois tipos de táticas: combinadores de serviços e decoradores de invocação.

[1]Combinadores de serviços foram propostos originalmente em [1] para definir procedimentos para tratamento de falhas comuns na recuperação de páginas *Web*.

[2]Poderia ser argumentado que o uso de um *proxy* [2] nesse caso seria suficiente para modularizar a implementação da referida tática em uma única classe. No entanto, pelo menos o código para criação e configuração do *proxy* nos módulos M_1, M_2, ..., M_n seria intrusivo. Além disso, conforme discutido em [7], o uso desse padrão de projeto não propicia uma modularização adequada para o tratamento de exceções remotas.

2.1. Combinadores de Serviços

Suponha que C_1 e C_2 são duas chamadas remotas de métodos, associadas respectivamente aos serviços S_1 e S_2. O *framework* proposto permite que essas 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).

Táticas do tipo combinadores de serviços são definidas pela gramática abaixo (onde terminais são escritos entre aspas):

```
<chamada> ::= <chamada_simples> | <chamada_composta>
<chamada_composta> ::= <chamada> "?" <chamada>
                     | <chamada> "|" <chamada>
                     | <chamada> ">" <chamada>
<chamada_simples> ::= <endpoint> "(" <argumentos> ")"
```

Nessa gramática, o não-terminal `endpoint` representa uma abstração usada para denotar um espaço de endereçamento remoto. No *framework* proposto, essa gramática é representada por meio do padrão de projeto Interpretador [2]. Esse padrão propõe uma representação para uma gramática de uma determinada linguagem, assim como uma forma de implementar um interpretador para a mesma. Basicamente, o padrão propõe que uma classe seja usada para representar cada regra da gramática, conforme mostrado a seguir:

```
abstract class Chamada {
  public Object run(String metodo, Class[] argClasses,
                    Object[] arg) throws RemoteException;
}
class ChamadaSimples extends Chamada {
  public ChamadaSimples(Remote endpoint) {
  public Object run(String metodo, Class[] argClasses, Object[] args)
           throws RemoteException;
}
class ChamadaComposta extends Chamada {
  public ChamadaComposta(char op, Chamada c1, Chamada c2){
  public Object run(String metodo, Class[] argClasses, Object[] args)
            throws RemoteException;
}
```

O método `run`, usado para executar uma `Chamada`, possui os seguintes parâmetros: o nome do método associado à chamada, um vetor contendo os tipos dos parâmetros do método associado à chamada (esses tipos são representados pelo tipo `Class` da biblioteca de reflexividade de Java) e os argumentos da chamada (os quais são representados como um arranjo de `Object`).

O *framework* inclui ainda um aspecto abstrato, cujo código é mostrado a seguir. Esse aspecto é responsável por obter, usando os recursos de reflexividade de AspectJ, o nome, os tipos e os argumentos de uma determinada chamada remota (linhas 5 a 8) e então executá-la (linha 9) de acordo com uma determinada tática[3]. Esse aspecto é abstrato já que não define as chamadas do programa ao qual será associado (o que é feito definindo o *pointcut* abstrato `ChamadaRemota` (linha 3)), nem a tática a ser associada a essas chamadas (o que é feito implementando o método abstrato `getChamada` (linha 2)). Esses parâmetros devem ser definidos em subaspectos do mesmo. O código do aspecto descrito é o seguinte:

```
1: abstract aspect Tactics {
2:   protected abstract Chamada getChamada();
3:   public abstract pointcut ChamadaRemota();
4:   Object around() throws RemoteException : ChamadaRemota(){
5:      CodeSignature sig= (CodeSignature) thisJoinPoint.getSignature();
6:      String nomeMetodo= sig.getName();
7:      Class[] argClasses= sig.getParameterTypes();
8:      Object[] argumentos= thisJoinPoint.getArgs();
9:      return getChamada().run(nomeMetodo, argClasses, argumentos);
    }
  }
```

Exemplo: Seja um serviço RMI definido pela seguinte interface:

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

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

```
class HelloClient {
   ......
   Hello server= TacticsHelper.init();
   String s1= server.sayHello();
   ......
   String s2= server.sayHello2("Bob");
   ......
}
```

O método `TacticsHelper.init()` deve ser chamado para inicializar toda referência remota declarada na aplicação cliente[4].

[3]Solução semelhante a essa é adotada pelo *framework* Gotech[8], proposto para modularização em aspectos de código de distribuição requerido em servidores.

[4]Na realidade, esse método apenas retorna `null`, evitando assim um erro de compilação do tipo "variável pode não ter sido inicializada". Consideramos que essa é a única solução possível para evitar esse erro, já que em AspectJ não existem pontos de junção associados à declaração de variáveis locais.

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

Para tanto, o programador usuário do *framework* deverá criar o seguinte subaspecto do aspecto `Tactics`:

```
 1: aspect HelloClientTactics extends Tactics {
 2:   private Chamada ch;
 3:   public pointcut ChamadaRemota(): within(HelloClient) &&
 4:                                    call(* Hello.*(..));
 5:
 6:   public HelloClientTactics(){
 7:     String s1= "skank.inf.pucminas.br/helloSrv";
 8:     String s2= "patofu.inf.pucminas.br/helloSrv";
 9:     ch = new ChamadaComposta('>',
10:              new ChamadaSimples(Naming.lookup(s1)),
11:              new ChamadaSimples(Naming.lookup(s2)));
12:   }
13:   public Chamada getChamada(){
14:     return ch;
15:   }
16: }
```

Na linha 3, especifica-se que a tática representada por esse aspecto deve ser associada 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).

2.2. Decoradores de Invocação

Decoradores de invocação permitem agregar processamento extra no fluxo de execução de chamadas remotas. Decoradores dispensam o uso de herança para estender a funcionalidade da classe `Chamada` e permitem encadear diversas tarefas que devem ser executadas durante uma invocação remota [2].

Um decorador é especificado pela seguinte classe:

```
class Decorador extends Chamada {
   public Decorador (Chamada ch);
   public Object run(String metodo, Class[] argClasses,
                     Object[] arg) throws RemoteException;
}
```

O construtor dessa classe recebe como parâmetro um objeto denotando a chamada a ser decorada. O método `run` simplesmente redireciona uma chamada para esse objeto.

Para criar um decorador de invocação, basta estender a classe `Decorador` e prover uma implementação para o método `run`. Essa implementação deve prover a funcionalidade extra

que demandou a criação do decorador. No *framework*, 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).

3. Discussão da Solução Proposta

Considera-se que a implementação de táticas por meio do *framework* apresentado na Seção 2 apresenta os seguintes benefícios:

Modularização: Esse benefício é conseqüência direta da implementação do *framework* em AspectJ. No sistema proposto, uma tática é implementada por meio de um único aspecto descendente do aspecto `Tactics`. Esse subaspecto define o código da tática, por meio das classes que encapsulam combinadores de serviços e decoradores de invocação, e por meio de um *pointcut* que define os pontos do código onde a tática deve ser aplicada. Assim, o código de uma tática fica encapsulado em um aspecto e, portanto, não se entrelaça com o código da aplicação.

***Obliviousness* e Separação de Responsabilidades**: Esses dois benefícios também decorrem diretamente do emprego de orientação por aspectos. Em AOP, *obliviousness* designa o fato de programas alvo não terem que aderir a nenhuma configuração ou protocolo específico a fim de serem passíveis de customização por meio de aspectos. Portanto, *obliviousness* é um requisito chave para se obter *separação de responsabilidades* (*separation of concerns*). Particularmente, no *framework* proposto, o programador pode ignorar as táticas que serão aplicadas a chamadas remotas. Essas táticas podem ser definidas por um outro desenvolvedor, com maior proficiência em programação e configuração de sistemas distribuídos. Além disso, esses dois benefícios viabilizam a implementação incremental de táticas. Uma primeira versão do sistema pode, por exemplo, ser desenvolvida sem a inclusão de nenhuma tática, a fim de validar seus requisitos funcionais. Posteriormente, táticas podem ser inseridas, a fim de implementar requisitos não-funcionais, como distribuição, tolerância a falhas, desempenho etc.

Na implementação atual do sistema, pode-se argumentar que não se obtém uma separação completa de responsabilidades, já que exceções remotas são propagadas para a aplicação cliente. No entanto, pode-se facilmente criar um combinador de serviço para tratar falhas e impedir a propagação das mesmas para o código cliente. Supondo que F seja um combinador desse tipo, uma chamada composta $C > F$ especifica que o tratamento de falhas na chamada de C é realizado em F. Assim, a interface remota usual de Java RMI poderia ser substituída por uma interface "despida" de qualquer código de distribuição. Além disso, a exceção `RemoteException` ativada pelo método `run` da classe `Chamada` poderia ser substituída por uma exceção própria e não-verificada do *framework*.

Extensibilidade: Novas táticas podem ser definidas de forma simples, bastando estender as classes `Chamada` e/ou `Decorador`. Pode-se, por exemplo, criar um combinador que implemente uma política de distribuição de carga mais sofisticada, baseada em sessões e que considere também a carga do servidor remoto.

4. Trabalhos Relacionados

Além de serem implementadas por meio do *framework* apresentado neste trabalho, considera-se que táticas podem também ser incorporadas a sistemas de *middleware* por meio de três soluções: usando-se linguagens de domínio específico para definição de táticas, usando-se metodologias para encapsular requisitos de distribuição em aspectos e usando-se objetos interceptadores. Essas três soluções são comparadas a seguir com a solução proposta neste trabalho.

Linguagens para Definição de Táticas: Pereira et al. [6] descrevem uma linguagem de domínio específico para especificação de táticas e como a mesma foi incorporada a um sistema 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, esse compilador gera *stubs* que implementam as táticas especificadas. Em relação ao *framework* proposto neste trabalho, consideramos que a implementação de táticas em Aries apresenta duas desvantagens. Primeiro, trata-se de uma solução específica para um sistema de *middleware*, enquanto que a solução proposta nesse trabalho pode ser facilmente incorporada a outros sistemas. Em segundo lugar, a solução do sistema Aries não possui o rico conjunto de operadores disponíveis em AspectJ para especificação de pontos do programa onde táticas serão colocadas em ação (pontos de junção, na terminologia de AspectJ). Particularmente, uma vez associadas a um método de uma interface remota, táticas são sempre ativadas quando da chamada desse 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 conjuntos de pontos de junção (*pointcuts*) e associá-los a uma mesma definição de tática.

Implementação de Distribuição por meio de Aspectos: Soares et al. [7] descrevem uma experiência de uso de AspectJ na restruturação de uma aplicação distribuída baseada em Java RMI. O objetivo foi modularizar em aspectos a comunicação RMI do sistema. No lado do cliente, a solução proposta inclui a implementação de um *advice* para cada método de um serviço remoto. Esse *advice* tem a função de redirecionar chamadas para a instância do objeto remoto. Assim, o mesmo poderia ser facilmente utilizado para implementar o código de uma tática. Ressalte-se, no entanto, que o *framework* proposto neste trabalho prescinde desses diversos *advices*, uma vez que o mesmo é baseado nos recursos de reflexão de AspectJ.

***Middleware* com suporte a Objetos Interceptadores:** Sistemas como CORBA [5] permitem que metaobjetos sejam interpostos no fluxo normal de execução de uma chamada remota. Assim, esses objetos podem ser usados para implementar tanto táticas baseadas em combinadores de serviços, como táticas baseadas em decoradores. No entanto, assim como ocorre no sistema Aries mencionado anteriormente, interceptadores são normalmente associados a todas chamadas de um determinado serviço remoto. O programador, nesses casos, não dispõe de meios para especificar a quais chamadas remotas um interceptador deve ser associado. Além disso, interceptadores são normalmente manipulados por meio de APIs reflexivas, baseadas em reificação de componentes internos do *middleware*, o que faz com que os mesmos ofereçam um baixo grau de abstração. De forma mais extensiva, essa abordagem é também adotada em sistemas de *middleware* reflexivos [4]. Já no *framework* proposto, reflexão está encapsulada em seus módulos, não sendo um recurso manipulado diretamente pelos usuários.

5. Conclusões

Neste artigo, apresentou-se um *framework* orientado por aspectos para implementação e modularização de táticas para invocação de métodos remotos em aplicações Java RMI. Táticas são usadas para adaptar uma aplicação para funcionamento em um ambiente distribuído. O *framework* proposto possui as seguintes propriedades: modularidade, separação de responsabilidades na implementação de táticas, não requer que protocolos especiais sejam implementados pela aplicação cliente (*obliviousness*) e extensibilidade. Essas propriedades foram obtidas, em grande medida, devido ao uso de orientação por aspectos. Como trabalhos futuros, pretende-se estender o *framework* para permitir a implementação de táticas típicas de servidores.

Agradecimentos: Este trabalho originou-se de um projeto de pesquisa financiado pela FAPEMIG (processo EDT 1906/03).

Referências

[1] 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.

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

[3] 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.

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

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

[6] 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.

[7] 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.

[8] Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis, and Marc Fleury. Aspectizing Server-Side Distribution. In *Proceedings of the Automated Software Engineering (ASE) Conference*, pages 130–141. IEEE Press, October 2003.

[9] Fabio Tirelo, Roberto da Silva Bigonha, Mariza da Silva Bigonha, and Marco Túlio de Oliveira Valente. Desenvolvimento de Software Orientado por Aspectos. XXIII Jornada de Atualização em Informática, XXIV Congresso da Sociedade Brasileira de Computação, August 2004.

[10] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag, 1997.

[11] 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.