Interceptação de Métodos Remotos em Java RMI

Marco Túlio Valente, João Paulo Santos e César Couto

[1]Departamento de Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais
Belo Horizonte - MG - Brasil

mtov@pucminas.br, jpsantos@pucmg.br, cesarfmc@pucmg.br

Resumo. *Este artigo descreve uma extensão de Java RMI com suporte a interceptação de chamadas remotas de métodos. A extensão proposta permite que programadores de aplicações distribuídas definam metaobjetos, chamados interceptadores, com métodos que são automaticamente executados quando uma chamada remota é disparada. Interceptadores podem ser usados para customizar e estender aplicações distribuídas de um modo não-invasivo. Além disso, os mesmos favorecem a implementação e modularização de requisitos não-funcionais. O sistema de interceptação proposto é dinâmico, já que interceptadores são criados, inseridos e removidos em tempo de execução. Interceptadores são também transparentes, já que não requerem modificações na lógica da aplicação distribuída cujas mensagens estão sendo capturadas. O sistema proposto é construído sobre a implementação original de Java RMI.*

Abstract. *This paper describes an extension of Java RMI with metaprogramming abstractions. The proposed extension allows users to interpose metaobjects, called interceptors, in the path of a remote method invocation. Interceptors can be used to customize and extend RMI applications in a non-invasive way. The proposed interception system is dynamic since interceptors can be plugged and unplugged at runtime and their code is generated on-the-fly, without the need of special interfaces or compilation steps. Interceptors are transparent since they do not require modification in the application logic or in the underlying RMI layer.*

1. Introdução

Middlewares para construção de aplicações distribuídas estão gradativamente incluindo suporte a mecanismos que permitem customizar, estender e monitorar seu comportamento [Wang et al., 2001, Kon et al., 2002]. Geralmente, estes mecanismos são baseados em metaobjetos [Kiczales et al., 1991], os quais permitem inspecionar e modificar as mensagens trocadas em aplicações distribuídas. Este tipo de interceptação pode ser usado, por exemplo, para tornar transparente aos desenvolvedores de tais aplicações a implementação de tarefas de maior complexidade, como protocolos de segurança, autenticação, cache, persistência ou tolerância a falhas. Dessa forma, interceptação de mensagens contribui para melhorar o grau de modularização de uma aplicação distribuída,

favorecendo a implementação de requisitos funcionais e não-funcionais em módulos distintos da aplicação. Além disso, interceptação de mensagens pode ser usada para adaptar uma aplicação a possíveis mudanças de requisitos surgidas durante seu funcionamento. Por exemplo, aplicações podem ter que interagir com novos objetos que não existiam quando as mesmas foram disponibilizadas ou ter que suportar novos requisitos não-funcionais.

Uma exceção a esta tendência de se construir *middlewares* cada vez mais abertos, adaptáveis e reconfiguráveis é Java RMI [Wollrath et al., 1996, Sun Microsystems, 2002]. RMI é um *middleware* orientado por objetos que permite invocações de métodos entre JVMs localizadas em nodos diferentes de uma rede. O sistema tem sido usado com sucesso em diversas aplicações distribuídas baseadas em Java e constitui ainda a infraestrutura básica de comunicação da arquitetura Jini [Waldo, 1999], a qual está sendo proposta para se construir sistemas distribuídos em redes dinâmicas e espontâneas.

Apesar de ser usado atualmente em diversas aplicações distribuídas, RMI ainda possui um projeto monolítico, o qual dificulta a evolução de aplicações construídas sobre o mesmo. Este artigo descreve então uma extensão de RMI, chamada RMI+, com abstrações que permitem a instrumentação e customização de aplicações construídas neste sistema. A solução proposta foi desenvolvida sobre a implementação original de Java RMI e requer modificações mínimas em sua API. Particularmente, interfaces remotas definidas por usuários e *stubs* gerados pela implementação original do sistema não são modificados. Em vez disso, os recursos de reflexividade de Java são usados para interceptar dinamicamente chamadas remotas de método, de uma maneira transparente à lógica original das aplicações. Na solução descrita no artigo, objetos *proxy*, chamados de interceptadores, são introduzidos tanto no lado do cliente, como no servidor de uma aplicação distribuída. Estes objetos podem ser instrumentados com código que é executado automaticamente antes que uma mensagem seja enviada para um objeto remoto e quando uma resposta (ou exceção) for recebida.

Este artigo encontra-se organizado como descrito a seguir. A Seção 2 apresenta uma visão geral de Java RMI. A Seção 3 descreve a arquitetura e os princípios de projeto que guiaram o desenvolvimento de RMI+. A Seção 4 apresenta a API do sistema. Na Seção 5, mostra-se um exemplo de uso de RMI+ no contexto de computação móvel. A Seção 6 descreve a implementação do sistema e a Seção 7 mostra algumas medidas de desempenho. Trabalhos relacionados são discutidos na Seção 8. Por último, a Seção 9 conclui o artigo.

2. Java RMI: Visão Geral

Java RMI é um pacote Java que permite criar objetos cujos métodos poderão ser invocados remotamente a partir de aplicações clientes, localizadas em JVMs remotas. Uma aplicação cliente em RMI tipicamente consulta o serviço de nomes do sistema – chamado RMI Registry – a fim de obter uma referência de rede, isto é, uma referência para um objeto localizado em uma outra JVM[1]. Esta referência pode ser então utilizada para invocar métodos do objeto remoto, com uma sintaxe equivalente àquela de chamadas locais.

[1]Além de serem obtidas via uma consulta ao RMI Registry, referências de rede também podem ser obtidas via resultado de uma chamada remota.

Aplicações servidoras tipicamente são responsáveis por criar objetos remotos e, eventualmente, registrá-los no serviço de nomes do sistema. Em Java RMI, a classe de um objeto remoto deve estender uma classe pré-definida, chamada `java.rmi.server.UnicastRemoteObject`. Além disso, esta classe deve implementar uma interface remota, isto é, uma interface que estende `java.rmi.Remote`. Uma interface remota define um contrato entre clientes e servidores, especificando métodos que os primeiros podem invocar nos segundos. Por fim, métodos de uma interface remota devem relacionar em sua cláusula `throws` exceções do tipo `java.rmi.RemoteException`. Em aplicações clientes, tais exceções são ativadas pela implementação de Java RMI para indicar uma falha de comunicação com a JVM servidora.

RMI Registry: A classe `java.rmi.Naming` oferece métodos estáticos para registrar (`bind`) e pesquisar (`lookup`) objetos remotos associados ao RMI Registry. Ao se registrar um objeto, deve-se informar um nome textual para o mesmo. A escolha deste nome deve ser acordada com as aplicações clientes, já que estas precisam informá-lo ao pesquisar por um objeto remoto associado ao Registry de um determinado nodo.

Arquitetura Interna: Internamente, a implementação de Java RMI se baseia em dois objetos principais, chamados de *stubs* e *skeletons*. Estes dois objetos são usados para implementar a abstração proporcionada por uma referência remota. Em uma aplicação cliente, uma referência de rede é, na realidade, uma referência para um objeto local, denominado *stub*. Este objeto implementa a mesma interface remota do objeto servidor, sendo responsável por encapsular todos os detalhes de comunicação com o mesmo. Um invocação remota é então "capturada" pelo *stub* do cliente, que converte seus parâmetros para uma representação serializada, a qual é enviada para um objeto da aplicação servidora chamado de *skeleton*. Este objeto é responsável por recuperar os dados da chamada e invocar o método servidor. Após este método ser executado, seu resultado é enviado de volta ao processo cliente, via objetos *skeleton* e *stub*.

As classes de *stubs* e *skeletons* são geradas automaticamente a partir da classe do objeto servidor. Esta geração é realizada pela ferramenta `rmic`, a qual faz parte de ambientes de desenvolvimento de aplicações em Java[2].

3. RMI+: Arquitetura e Princípios de Projeto

A Figura 1 descreve a arquitetura de interceptação de chamadas remotas proposta neste artigo. A arquitetura proposta é construída sobre a camada de comunicação fornecida pela implementação original de Java RMI, a qual contém objetos *stubs* e *skeletons*. Particularmente, *stubs* e *skeletons* podem ser considerados interceptadores, já que chamadas remotas são tratadas através dos mesmos. Entretanto, o código destes dois objetos é fixo, isto é, os usuários não têm acesso ao mesmo para adicionar funcionalidades extras. Assim, RMI+ estende RMI permitindo que os desenvolvedores de aplicações distribuídas definam seus próprios interceptadores no caminho seguido por chamadas remotas.

[2]Nas novas versões de Java, o compilador `rmic` é usado apenas para gerar o código de objetos *stubs*. *Skeletons* são criados dinamicamente e usam um código genérico para invocar métodos do servidor.

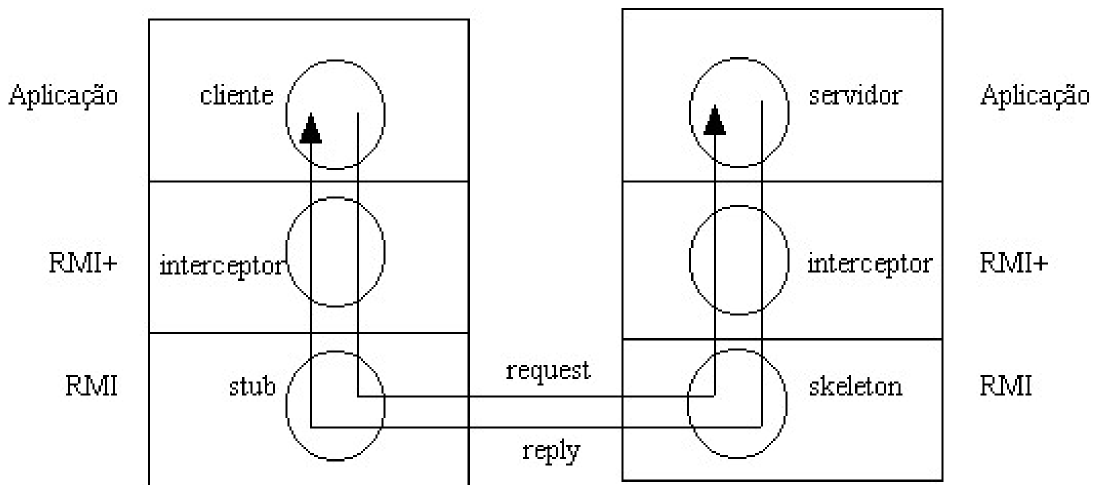

Figura 1: Arquitetura

Interceptadores são objetos de primeira classe, os quais implementam o padrão de projeto conhecido como *proxy* [Gamma et al., 1994][3] Além disso, interceptadores atendem aos seguintes requisitos:

- Interceptadores são dinâmicos e podem ser conectados e desconectados em tempo de execução. Isto significa que usuários têm total controle sobre o tempo de vida de interceptadores. Além disso, interceptadores são dinâmicos porque parte de seu código é gerada em tempo de execução. Este código intercepta e redireciona chamadas remotas para pontos de interceptação definidos pelos usuários.
- Interceptores são transparentes, já que objetos da camada de aplicação não necessitam estar cientes de sua existência. Qualquer chamada remota é sujeita a interceptação. Assim, a evolução e instrumentação de um sistema que usa interceptadores não requer mudanças no código existente. Além disso, interceptadores não requerem modificação nos *stubs* e *skeletons* gerado pelo compilador `rmic`. Não é necessário também nenhum passo ou ferramenta extra de compilação.
- Interceptores podem interceptar tanto mensagens no lado cliente como no lado servidor de uma aplicação distribuída. Interceptores do cliente são chamados antes de uma chamada remota ser enviada e depois que o resultado da mesma (ou uma exceção) for recebido. Interceptores do servidor são invocados após uma chamada remota ser recebida e antes que o resultado da mesma (ou uma exceção) seja enviado para o cliente. Assim, o mesmo método de interceptação manipula tanto as mensagens enviadas como aquelas recebidas em um determinado lado de uma aplicação distribuída. Esta característica simplifica a programação quando a lógica implementada pela interceptação abrange ambos os tipos de mensagens. Caso contrário, tal lógica deveria ser programada usando variáveis globais, tornanda-a o mais complexa e sujeita a erros.
- Interceptadores podem ser compartilhados por mais de um cliente ou servidor. Ou seja, desenvolvedores de aplicações distribuídas não necessitam escrever um código particular de interceptação para cada interface remota. Isto aumenta o

[3]Neste padrão de projeto, um *proxy* atua como um mediador de todo acesso realizado a um determinado objeto.

grau de reutilização e favorece a modularização de requisitos transversais (*cross-cutting behaviors*) [Kiczales et al., 1997].

- Interceptores podem inspecionar e modificar os parâmetros e o resultado de uma chamada remota. Eles podem também alterar o objeto remoto alvo de uma chamada a fim de redirecioná-la para um outro servidor. Interceptores do lado cliente podem modificar o fluxo normal de uma chamada remota retornando um resultado (sem chamar o objeto servidor) ou ativando uma exceção. Interceptores do lado do servidor também podem produzir um resultado ou uma exceção sem invocar o objeto servidor.

4. Interface de Programação

RMI+ é implementado como um pacote chamado `rmiplus`. As classes públicas e interfaces deste pacote são apresentadas a seguir.

Classe Naming: Esta classe suporta a mesma interface da classe `java.rmi.Naming` e, portanto, fornece métodos estáticos para registrar, pesquisar e listar objetos remotos mantidos no RMI Registry. Os métodos desta classe usam a classe original `Naming` a fim de prover um serviço de nomes a aplicações construídas usando RMI+. Entretanto, quando chamados, tais métodos instanciam a infra-estrutura de execução que suporta a definição de interceptadores.

Classe ORB: Esta classe possui métodos para "plugar" e "desplugar" interceptadores, tanto do lado cliente como do lado servidor de uma aplicação distribuída. A classe tem a seguinte interface:

```
class ORB {
  public static void attach(Remote s, Interceptor p);
  public static void detach(Remote s);
}
```

O método `attach` associa um interceptador `p` a um determinado objeto remoto `s`. Se este método for chamado no lado cliente de uma aplicação distribuída, `p` será um interceptador cliente. Se chamado no lado do servidor, `p` será um interceptador servidor. O método `detach` remove um possível interceptador que tenha sido adicionado a s. Ressalte-se que, na presente implementação, no máximo um interceptador pode ser adicionado a uma referência remota.

Interface Interceptor: Interceptadores são objetos que implementam a seguinte interface:

```
interface Interceptor {
  public Object intercept(Object obj, Method method,
                   Object[] args) throws Throwable;
}
```

O método `intercept` define o código que é executado por um interceptador. Este método possui três parâmetros. O primeiro é o próximo objeto a ser chamado no fluxo normal de uma chamada remota, isto é, o *stub* (na caso de um interceptador cliente) ou o objeto servidor (no caso de um interceptador servidor). A classe `Method` é definida pela API de reflexividade de Java. Esta classe reifica o método interceptado e fornece operações para recuperar o seu nome, seu tipo de retorno e os argumentos da chamada. Por meio do método `invoke()` pode-se inclusive executar o método reificado. O último argumento do método `intercept` é um vetor do tipo `Object` que armazena os argumentos da chamada remota que está sendo interceptada.

Assim, o código de um método `intecept` segue geralmente o seguinte padrão:

```
Object intercept(....) {
  // processa mensagem a ser enviada
  ......
  // propaga chamada
  Object result= method.invoke(obj, args);
  // processa mensagem recebida como resultado
  .......
  return result;
}
```

5. Exemplo: Sistema de Chat para Computação Móvel

A motivação original para projetar RMI+ surgiu quando da adaptação para redes sem fio de um sistema de *chat* originalmente projetado para uso em redes fixas. Basicamente, este sistema permite que vários usuários troquem entre si mensagens de texto, as quais são simultaneamente exibidas na interface de todos usuários conectados ao sistema. Assim, o sistema é constituído por aplicações clientes, instaladas nos nodos dos usuários que estão "conversando", e por uma aplicação servidora. Qualquer mensagem digitada por um dos usuários é enviada, via RMI, para a aplicação servidora, a qual se encarrega de difundi-la – via diversos métodos *callback* também implementados usando-se RMI – para todos os clientes conectados ao sistema.

Foram enfrentados as seguintes dificuldades na adaptação desta aplicação para redes sem fio:

Teste da Aplicação: Embora o sistema de *chat* estivesse sendo adaptado para o uso em redes sem fio, por questões de infra-estrutura, seria muito mais fácil testá-lo em um ambiente tradicional de rede. No entanto, redes fixas são radicalmente diferentes de redes móveis. Por exemplo, estas possuem menor largura de banda e estão sujeitas a desconexões voluntárias e involuntárias. Assim, a fim de gerar um ambiente de teste mais real, decidiu-se introduzir um interceptador no cliente que simulasse parâmetros de qualidade de serviço típicos de acessos móveis. Basicamente, este interceptador introduz um atraso em cada chamada remota e simula desconexões ativando exceções remotas, seguindo uma freqüência de desconexões típica de redes sem fio.

Mostra-se a seguir o código do interceptador adicionado no lado cliente da aplicação de *chat*:

```
class MobQoSSim implements Interceptor {
  private long delay;
  private int disconnRatio= 15;
  .....
  public Object intercept(Object obj, Method method,
                          Object[] args) throws Throwable {
    if(Random.uniform.getNextIntFromTo(0,100) < disconnRatio)
      throw new RemoteException();
    else {
      Thread.sleep(delay);
      return method.invoke(obj, args);
    }
  }
}
```

O interceptador mostrado assume que o atraso é fixo para todas as chamadas de métodos remotos e que a freqüência de desconexões segue uma distribuição uniforme. Foi também implementado uma segunda versão deste interceptador que leva em consideração o tamanho do vetor de argumentos do método chamado para computar o atraso e que simula desconexões com uma determinada duração. Neste caso, todos os métodos chamados durante o estado desconectado falham.

O código seguinte mostra como o interceptador mostrado foi "plugado" no lado cliente do sistema de chat:

```
ChatServer s= (ChatServer)
    rmiplus.Naming.lookup("srv01/chatsrv");
ORB.attach(s, new MobQoSSim(15,20));
```

Neste exemplo, assume-se que o interceptador criado introduz um atraso de 15 ms em cada chamada remota de método e que 20% das chamadas falham devido a uma desconexão.

Tratamento de Desconexões: O servidor usado por nossa aplicação de *chat* foi projetado tendo clientes fixos como alvo. Assim, já que desconexões são eventos raros em redes fixas, este servidor simplesmente descarta mensagens que não podem ser retransmitidas para nodos clientes do sistema. Desde o início da adaptação, tinha-se definido que uma política mais sofisticada era necessária para tratar estas mensagens. Um interceptador foi então adicionada ao servidor para armazenar temporariamente em um *buffer* mensagens que não puderam ser enviadas para clientes móveis (provavelmente, devido a desconexões temporárias dos mesmos). Basicamente, todas mensagens que chegam ao servidor são propagadas para os clientes do sistema via um *callback*. O interceptador proposto trata as exceções ativadas por este *callback* em caso de impossibilidade de comunicação com nodos clientes. Em vez de propagar esta exceção, o interceptador armazena a mensagem que deveria ser enviada em *buffer* e retorna o controle para o servidor. Toda vez que um cliente realiza uma nova conexão ao servidor de *chat*, o interceptador verifica no *buffer* se existe alguma mensagem para aquele cliente. Se houver, o servidor despacha as mesmas.

6. Implementação

Esta seção descreve a implementação do sistema RMI+. A implementação usa a noção de classes *proxy* dinâmicas, disponível na biblioteca de reflexividade de Java. Inicialmente, descreve-se o funcionamento básico de classes *proxy* diâmicas e então entra-se em maiores detalhes sobre a implementação de RMI+.

6.1. Classes Proxy Dinâmicas

A partir do JDK 1.3, Java fornece suporte a objetos *proxy*, os quais são instâncias de classes *proxy* dinâmicas [Arnold et al., 2000]. Estas classes têm duas propriedades principais: (i) seu código é criado em tempo de execução pela API de reflexividade de Java; (ii) elas implementam uma lista de interfaces especificadas em tempo da criação. Instâncias de classes *proxy* dinâmicas têm ainda um objeto manipulador de invocação (*invocation handler*), o qual é definido pelo cliente que solicitou a criação das mesmas. Toda a invocação de método sobre uma instância de uma classe *proxy* dinâmica é enviada automaticamente para o método `invoke` do manipulador desta instância, passando os seguintes parâmetros: a instância da classe *proxy* sobre a qual a invocação foi realizada; um objeto da classe `java.lang.reflect.Method`, o qual reifica o método que está sendo chamado; e um vetor do tipo `Object` que contém os argumentos deste método.

Classes *proxy* dinâmicas podem ser usadas para criar metaobjetos em tempo de execução, os quais seguem o padrão de projeto *proxy*, conforme mostrado na Figura 2. Sabe-se, no entanto, que tais metaobjetos são incapazes de interceptar mensagens enviadas ao atributo *self* do objeto base. Este fato certamente é uma restrição quando se usa tais metaobjetos em contextos mais gerais. Entretanto, na solução proposta neste artigo, interceptadores são usados com a finalidade exclusiva de manipular invocações remotas de métodos, as quais obviamente nunca são realizadas sobre atributos *self*.

6.2. Implementação do Sistema RMI+

A disponibilidade de classes *proxy* em Java simplificou bastante a implementação de interceptadores em RMI+. Entretanto, mesmo usando-se classes *proxy*, a implementação do sistema não é tão direta como pode-se pressupor inicialmente.

Suponha que a implementação original de RMI seja estendida com uma classe fábrica (*factory*) que cria um objeto remoto e retorna uma instância de um *proxy* dinâmico para o mesmo. A dificuldade principal surge quando tenta-se inserir este objeto no serviço de nomes de RMI. Este sistema de nomes é construído usando os próprios recursos de comunicação do padrão RMI. Assim, há duas possibilidades sobre qualquer consulta realizada sobre o mesmo (via método `lookup`):

- Se o objeto associado ao Registry for remoto, isto é, implementar uma interface que estende `java.rmi.Remote`, o *stub* deste objeto é retornado como o resultado de uma chamada ao método `lookup`. Este *stub* envia chamadas remotas para seu correspondente *skeleton*, que as repassa para o objeto remoto. Este é o caso mais comum em aplicações Java RMI.
- Se o objeto associado ao Registry for serializável, isto é, implementar a interface `java.io.Serializable`, sua representação serializada é retornada como o resultado de uma chamada ao método `lookup`. Como usual em Java, a

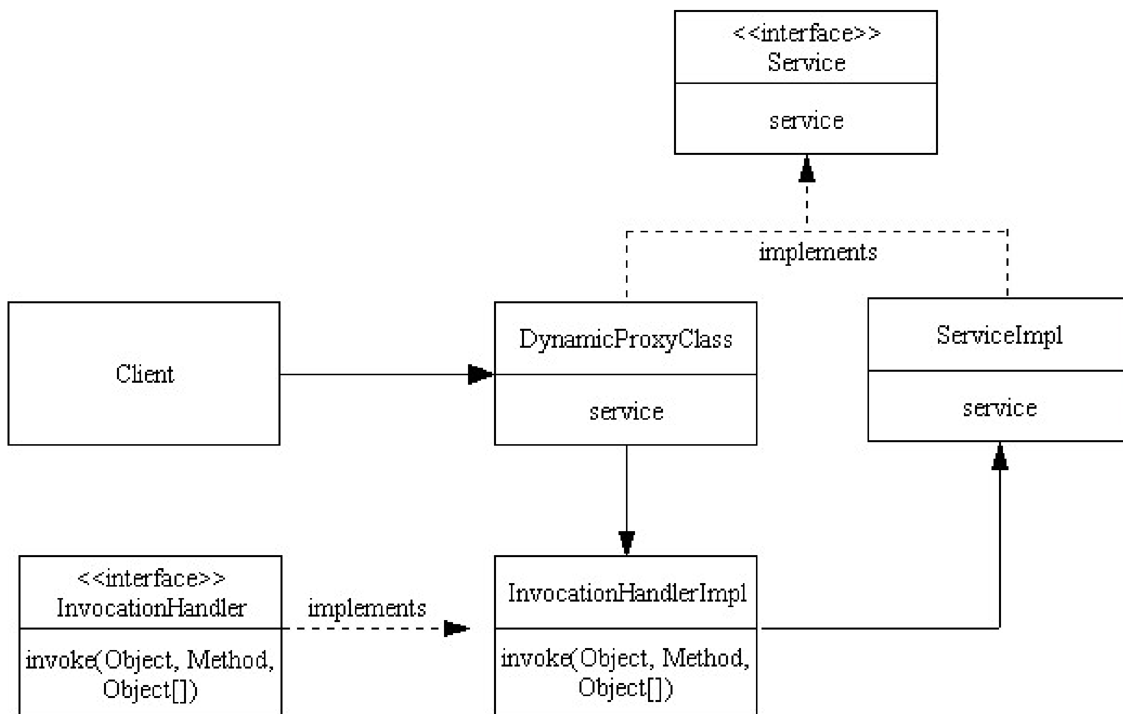

Figura 2: Padrão de projeto *proxy* usando classes *proxy* dinâmicas

serialização de um objeto p inclui todos os objetos alcançáveis a partir do mesmo. Se um objeto remoto for atingido quando se computa o fecho transitivo de um determinado objeto, o mesmo é substituído por seu *stub*, como descrito no item anterior.

Se um objeto for simultaneamente serializável e remoto, a semântica de cópia descrita na segunda alternativa acima é utilizada. Este será o caso, portanto, de um possível objeto *proxy*, criado pela fábrica mencionada acima, e associado a um objeto remoto[4]. Suponha então que p é um *proxy* associado a um objeto remoto s. Se p for inserido no Registry, o cliente que faz uma chamada ao método `lookup` para obter uma referência para o objeto servidor s, receberá uma cópia de p e dos objetos alcançáveis a partir do mesmo. O único objeto diretamente acessível a partir de p é seu manipulador de invocação ih. Neste ponto, existem duas alternativas sobre as possíveis interfaces que devem ser implementadas por este manipulador (além da interface `java.lang.reflect.InvocationHandler`):

- Se o manipulador ih for serializável, o cliente receberá uma cópia do mesmo. Além disso, como ih possui uma referência para o objeto remoto s, a mesma será substituída por uma referência para o *stub* de s. Assim, se um cliente invocar um dos métodos de s, esta chamada será direcionada para o seu *stub*, que a enviará a seu respectivo *skeleton*, já na aplicação servidora. Finalmente, o *skeleton* roteará a chamada para o objeto servidor. Assim, para os objetivos do sistema de interceptação proposto neste artigo, esta alternativa não é adequada, já que a mesma não oferece nenhuma oportunidade para interceptação no lado do servidor.

[4]Este objeto implementa `java.rmi.Remote`, já que seu objeto base também o faz. Além disso, todo objeto *proxy* dinâmico construído pela API de reflexividade de Java estende a classe `java.lang.reflect.Proxy`, a qual implementa `java.io.Serializable`.

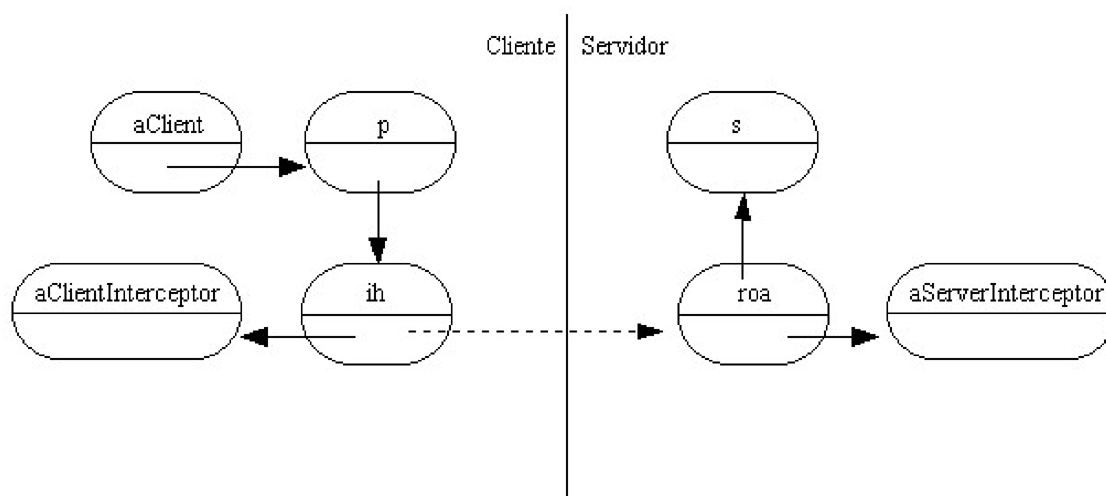

Figura 3: Adaptador para um Objeto Remoto

- Se o manipulador ih for um objeto remoto, seu *stub* é que será enviado para o processo cliente. Entretanto, o método `invoke` do manipulador de invocação possui um parâmetro da classe `Method`, a qual não é serializável ou remota. Assim, esta alternativa falharia com uma uma exceção do tipo `java.io.NotSerializableException` ativada pelo *stub* durante o processo de *marshalling* dos argumentos da chamada remota.

A solução adotada em RMI+ é uma combinação das duas alternativas de projeto apresentadas acima. Como na primeira alternativa, o manipulador de invocação é transformado em um objeto serializável. No entanto, o mesmo não referencia diretamente o objeto remoto, mas sim um objeto adaptador do mesmo, denominado roa. Este objeto é uma instância da classe `ROAImpl`, a qual implementa uma interface remota. Finalmente, como mostrado na Figura 3, o objeto adaptador referencia o objeto servidor. A classe `ROAImpl` implementa o padrão Adaptador [Gamma et al., 1994], já que a mesma converte as interfaces assumidas por chamadas remotas, originados a partir de um manipulador de invocação, para a interface assumida pelo objeto servidor.

Como a classe `Method` não é serializável, o manipulador de invocação inicialmente extrai os campos que denotam o nome e os argumentos do método chamado, os quais são reificados no objeto da classe `Method`. Estes campos são serializáveis e assim podem ser enviados como argumentos de uma chamada remota, realizada ao adaptador do objeto remoto. No lado do servidor, este objeto restaura o objeto `Method` original[5].

O manipulador de invocação (no lado do cliente) possui uma referência para um possível interceptador, o qual é chamado antes de se enviar uma mensagem ao adaptador remoto. No lado do servidor, o adaptador remoto também possui uma referência para um possível interceptador, o qual é chamado antes de enviar mensagens ao objeto servidor.

Descreve-se a seguir duas questões remanescentes tratadas pela implementação de RMI+:

[5] A mesma abordagem é usada no sistema TRMI [Gur-Ari, 2002], o qual visa simplificar a implementação de aplicações em Java RMI permitindo que toda manipulação de exceções remotas seja centralizada em um único objeto.

- Métodos remotos que retornam referências de rede como resultado. Suponha que s é uma referência de rede retornada por um método remoto. Neste caso, o adaptador do objeto remoto, em vez de s, retorna um *proxy* para s como resultado da chamada. Assim, o cliente passa a dispor da infra-estrutura necessária para inserir interceptadores neste resultado.
- Métodos remotos que possuem referências de rede como argumento. Suponha que s é uma referência de rede passada como argumento de um método remoto. De forma similar ao item anterior, o manipulador de invocação no lado cliente, em vez de s, passa como argumento um *proxy* para s.

7. Desempenho

Discute-se nesta seção o *overhead* introduzido pela incorporação de interceptadores em uma aplicação distribuída. Para isto, realizou-se um experimento, onde clientes invocam métodos remotos de um servidor com a seguinte interface:

```
public interface Remote1 extends Remote {
  public int meth1(int x, double y, String s)
               throws RemoteException;
  public int meth2(Remote2 x) throws RemoteException;
  public Remote2 meth3() throws RemoteException;
}

public interface Remote2 extends Remote {
  public void foo() throws RemoteException;
}
```

Interceptadores vazios, isto é, sem nenhum código, foram introduzidos tanto no lado cliente como no lado servidor da aplicação. Com isso, objetivou-se medir apenas o *overhead* introduzido pela infra-estrutura de interceptação. Ressalte-se ainda que referências remotas são passadas como argumentos e retornadas com resultado, respectivamente, nos métodos `meth2` e `meth3`. O objetivo nestes dois casos foi medir também o custo de criação dos objetos *proxy* que são passados no lugar de tais referências (conforme descrito no final da Seção 6).

A Tabela 1 resume os resultados obtidos no experimento. As máquinas clientes possuíam processador Atlhom, de 1 Ghz e 512 MB de RAM. Os servidores eram Pentium III, também de 1 Ghz e 512 MB de RAM. Ambos executavam Microsoft Windows 2000. Servidores e clientes estavam conectados por uma rede Ethernet. A JVM utilizada foi a que acompanha o JDK1.4 da Sun. Cada método remoto foi executado 1000 vezes e a média de tempo de todas as chamadas foi calculada.

Conforme mostram os resultados obtidos, o *overhead* advindo do uso de interceptadores é considerável. O mesmo é superior a 100% para todos os três métodos do teste realizado. No entanto, este custo é compatível com outros sistemas de meta-programação em Java. Por exemplo, interceptadores em outras arquiteturas de *middlewares*, como CORBA [Object Management Group, 2000b], também possuem custos semelhantes aos mostrados nesta tabela [Marchetti et al., 2001, Wang et al., 2001].

Operação	RMI	RMI+	Overhead
Lookup	218.91	266.91	22%
meth1	7.35	17.32	136%
meth2	17.03	48.93	187%
meth3	16.56	37.74	128%

Tabela 1: Chamadas RMI vs Chamadas RMI+ (em ms)

8. Trabalhos Relacionados

Protocolos para manipulação de metaobjetos (MOPs) já foram propostos para diversas linguagens. Guaraná [Oliva and Buzato, 1999], Kava [Welch and Stroud, 2001] e R-Java [Tomioka et al., 2001] são alguns exemplos de extensões de Java que incorporam este conceito.

Recentemente, *middlewares* orientados por objeto também estão incorporando abstrações para meta-programação. Por exemplo, CORBA suporta o conceito de interceptadores portáveis [Object Management Group, 2000a]. No entanto, interceptadores em CORBA têm interfaces surpreendentemente complexas e limitações importantes. Eles não podem, por exemplo, modificar os argumentos ou o valor de retorno de uma chamada remota de método. Em ORBs desenvolvidos em Java, interceptadores nem mesmo podem inspecionar os argumentos de uma chamada de método. Em diversas situações, interceptadores em CORBA também possuem *overheads* superiores a 100% [Marchetti et al., 2001, Wang et al., 2001].

Algumas implementações CORBA dispõem ainda de um segundo mecanismo de meta-programação, chamado de *smart stubs*. Estes *stubs* são criados pela própria aplicação e redefinem os *stubs* tradicionais de CORBA. Diferentemente de interceptadores em CORBA, *smart stubs* podem modificar os argumentos e o valor de retorno de chamadas remotas. No entanto, os mesmos oferecem possibilidade de interceptação apenas em clientes. Além disso, implementações de *smart stubs* seguem protocolos proprietários de cada ORB, já que ainda não foram padronizadas.

O sistema Aroma [Narasimhan et al., 2000] é outro exemplo de extensão de Java RMI com suporte a interceptadores. Em Aroma, interceptadores são usados para adicionar protocolos de tolerância a falhas em RMI. Interceptadores podem ser introduzidos tanto no nível de transporte, como no nível de aplicação. Basicamente, interceptadores no nível da camada de transporte permitem definir implementações customizadas de *sockets* TCP/IP. Interceptadores da camada de aplicação são similares aos interceptores propostos neste documento. Entretanto, podem ser introduzidos somente no lado do cliente de uma aplicação. Mais recentemente, o sistema SmartRMI [N. Santos, 2002] foi proposto com o objetivo de se introduzir interceptadores e *smart stubs* em Java RMI. Acredita-se, no entanto, que a interface de programação proposta em RMI+ é mais simples e intuitiva do que a existente em SmartRMI.

Middlewares reconfiguráveis e reflexivos constituem outro esforço de pesquisa objetivando a construção de sistemas abertos, reconfiguráveis e extensíveis [Kon et al., 2002]. Basicamente, os serviços oferecidos por tais *middlewares* podem ser configurados para atender aos requisitos do contexto corrente de execução de uma aplicação e reconfigurados, quando ocorrerem alterações significativas em tal

ambiente de execução.

Programação orientada por aspectos (AOP) [Kiczales et al., 1997] é uma tecnologia para separação e modularização de requisitos transversais (*croscutting concerns*). AOP fornece abstrações para modularização de tais requisitos em unidades chamadas aspectos. Uma ferramenta denominada *weaver* é então usada para "entrelaçar" classes e aspectos de um sistema. *Weaving* dinâmico, semelhante à abordagem adotada em RMI+, é discutido em [Popovici et al., 2002].

9. Conclusões

Java RMI é uma escolha natural para desenvolvimento de aplicações distribuídas onde tanto clientes como servidores são implementados em Java. Uma dos atrativos do sistema é sua forte integração como a plataforma de Java. Assim, características da linguagem, como interfaces, serialização e carregamento dinâmico de classes, contribuem para aumentar a expressividade do sistema. Mais recentemente, Java passou a incorporar a noção de objetos *proxy*, os quais são instâncias de classes geradas dinamicamente, a partir de uma lista de interfaces. Neste artigo, mostrou-se como Java RMI pode ser estendido de forma a se beneficiar deste novo conceito de Java. Basicamente, descreveu-se uma extensão de RMI com suporte a interceptadores de chamadas remotas de métodos.

O sistema de interceptação descrito no artigo oferece uma interface de programação bastante simples. Os interceptadores propostos são dinâmicos, já que parte de seu código é gerada em tempo de execução. Além disso, a interface do sistema oferece métodos para conectar e desconectar interceptadores dinamicamente. Interceptadores são também transparentes, já que se localizam entre a camada de aplicação e a camada original de comunicação provida por Java RMI, sem, no entanto, requerer modificações em ambas. Não são requeridas também modificações na JVM original de Java.

O *overhead* introduzido por interceptadores não é desprezível. No entanto, o mesmo é compatível com outros sistemas de meta-programação em Java citados na literatura. Como trabalho futuro, pretende-se investigar técnicas para reduzir este *overhead*. Pretende-se ainda permitir composição de interceptadores, isto é, a associação de múltiplos interceptadores a um mesmo objeto remoto.

Agradecimentos: Este trabalho originou-se de um projeto de pesquisa financiado pela FAPEMIG (processo CEX488/02) e pelo FIP/PUC Minas (processo 2002/38P).

Referências

Arnold, K., Gosling, J., and Holmes, D. (2000). *The Java Programming Language*. Addison Wesley, 3rd edition.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Gur-Ari, G. (2002). Empower RMI with TRMI. *JavaWorld*.

Hunt, G. and Scott, M. (1999). Intercepting and Instrumenting COM Applications. In *Fifth Conference on Object-Oriented Technologies and Systems*, pages 45–56.

Kiczales, G., J.d.R., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press, 3rd edition.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag.

Kon, F., Costa, F., Campbell, R., and Blair, G. (2002). The Case for Reflective Middleware. *Communications of the ACM*, 45(6):33–38.

Marchetti, C., Verde, L., and Baldoni, R. (2001). CORBA Request Portable Interceptors: A Performance Analysis. In *3rd International Symposium on Distributed Objects and Applications*, pages 208–217.

N. Santos, P. Marques, L. S. (2002). A Framework for Smart Proxies and Interceptors in RMI. In *15th International Conference on Parallel and Distributed Computing Systems*.

Narasimhan, N., Moser, L. E., and Melliar-Smith, P. M. (2000). Interception in the Aroma System. In *ACM 2000 Java Grande Conference*, pages 107–115.

Object Management Group (2000a). CORBA/Portable Interceptor Specification. ptc/2000-04-05.

Object Management Group (2000b). The Common Object Request Broker: Architecture and Specification. 2.4.

Oliva, A. and Buzato, L. E. (1999). The Design and Implementation of Guaraná. In *Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 203–216.

Popovici, A., Gross, T., and Alonso, G. (2002). Dynamic weaving for aspect-oriented programming. In *2nd International Conference on Aspect- Oriented Software Development*.

Sun Microsystems (2002). Java Remote Method Invocation Specification, revision 1.8.

Tomioka, E., Guimaraes, J. O., and do Prado, A. F. (2001). R-Java: A Reflection Java Extension. In *VI Simpósio Brasileiro de Linguagens de Programação*, pages 203–217.

Waldo, J. (1999). The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82.

Wang, N., Schmidt, D. C., Othman, O., and Parameswaran, K. (2001). Evaluating Meta-Programming Mechanisms for ORB Middleware. *IEEE Communications Magazine*, 39(10).

Wang, Y. M. and Lee, W.-J. (1998). COMERA: COM Extensible Remoting Architecture. In *Fourth Conference on Object-Oriented Technologies and Systems*.

Welch, I. and Stroud, R. (2001). Kava - Using Bytecode Rewriting to add Behavioural Reflection to Java. In *USENIX Conference on Object-Oriented Technology*.

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