Wendell Figueiredo Taveira

Chamada Assíncrona de Métodos Remotos em Java

Dissertação de Mestrado apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

25 de Agosto de 2003

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Chamada Assíncrona de Métodos Remotos em Java

WENDELL FIGUEIREDO TAVEIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Profa MARIZA ANDRADE DA SILVA BIGONHA - Orientadora
Departamento de Ciência da Computação - ICEx - UFMG

Prof. MARCO TÚLIO DE OLIVEIRA VALENTE - Co-orientador
Departamento de Ciência da Computação - PUC - MG

Prof. JOSÉ CARLOS MALDONADO
Departamento de Computação e Estatística - USP

Prof. OSVALDO SÉRGIO FARHAT DE CARVALHO
Departamento de Ciência da Computação - ICEx - UFMG

Prof. ROBERTO DA SILVA BIGONHA
Departamento de Ciência da Computação - ICEx - UFMG

Belo Horizonte, 31 de julho de 2003.

Agradecimentos

Agradeço a Deus por me permitir viver.

Agradeço à minha família por todo o amor.

Tenho muitos ainda a quem agradecer.

À Profa. Mariza Bigonha pela excelente orientação, pelo incentivo e pelo apoio em tantos momentos onde desistir de tudo era mais fácil.

Ao Prof. Marco Túlio por ter me ensinado novos significados para palavras como competência e responsabilidade. Muito obrigado pela orientação.

Ao Prof. Roberto Bigonha pelos comentários e sugestões sempre tão apropriados e inteligentes.

A todo pessoal do Laboratório de Linguagens, em especial, ao Fernando Magno sempre generoso e sábio. Obrigado pela companhia tão alegre durante este tempo.

Ao Colegiado do curso por toda compreensão a mim dispensada. Em particular, agradeço à Renata Viana, à Emília Soares e ao Prof. Henrique Pacca.

Ao Prof. Antônio Alfredo Loureiro que já era um modelo de professor para mim e que se tornou também um modelo de ser humano. Agradeço por toda força e abertura de alma. Foi muito importante para mim que alguém continuasse a acreditar em minhas capacidades.

À Letícia, Rosilane e Tulinha pelo carinho e atenção.

Ao amigo Ademilson, presença fraterna pelos caminhos da vida, não poucas vezes tortuosos e difíceis.

Finalmente, ao CNPq e à FAPEMIG (proc. CEX 488/2002) pelo apoio financeiro.

Resumo

Java RMI é o modelo computacional utilizado para construção de sistemas distribuídos utilizando a linguagem Java. Entretanto, Java RMI não é capaz de realizar chamadas assíncronas, o que pode diminuir a eficiência do sistema em alguns casos. Neste trabalho, apresentamos FlexRMI, um sistema totalmente compatível com a linguagem Java que torna possível invocar métodos remotos assincronamente de maneira transparente. A ferramenta foi implementada utilizando os recursos de reflexão computacional e de *proxy* dinâmico fornecidos por Java, tornando a solução simples e elegante.

Abstract

Java RMI is the standard computational model used to develop distributed systems in the Java language. Although widely used in the construction of distributed systems, the use of Java RMI is limited because this middleware does not allow asynchronous method invocation. In this masters thesis it is presented FlexRMI, a Java based system that supports asynchronous invocation of remote methods. The system is completely implemented in Java, making use of the reflective and dynamic proxy facilities of this language. The implementation is also compatible with standard Java RMI distributed systems.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Sistemas distribuídos têm se firmado cada vez mais como o padrão de plataforma de desenvolvimento de *software*. De modo geral, um sistema distribuído é composto por componentes localizados em uma rede de computadores e que se comunicam via troca de mensagens com o objetivo de compartilhar recursos [27]. A Internet caracteriza bem este novo tipo de ambiente, onde processadores heterogêneos, provavelmente situados em locais distintos, formam uma rede de proporção global [12, 9].

Um computador que executa componentes que formam parte de um sistema distribuído é chamado de hospedeiro[1], sítio[2] ou nodo[3]. Geralmente, o termo **sítio** é utilizado para indicar a localização de máquinas e **hospedeiro**, para referenciar um sistema específico em um sítio. Um hospedeiro pode ser visto como um conjunto de componentes operacionais, tais como *hardware* e sistema operacional de rede, como mostrado na Figura 1.1.

Para um determinado processador em um sistema distribuído, os demais hospedeiros e seus componentes operacionais, denominados de recursos, são ditos **remotos**, em contraste com seus próprios recursos que são ditos **locais**. Geralmente, um hospedeiro, denominado **servidor**, possui um recurso que outro hospedeiro, o **cliente**, deseja utilizar. É, portanto, função de um sistema distribuído proporcionar um ambiente adequado no qual os recursos possam ser compartilhados [23].

[1]Do inglês, *host*.
[2]Do inglês, *site*.
[3]Do inglês, *node*.

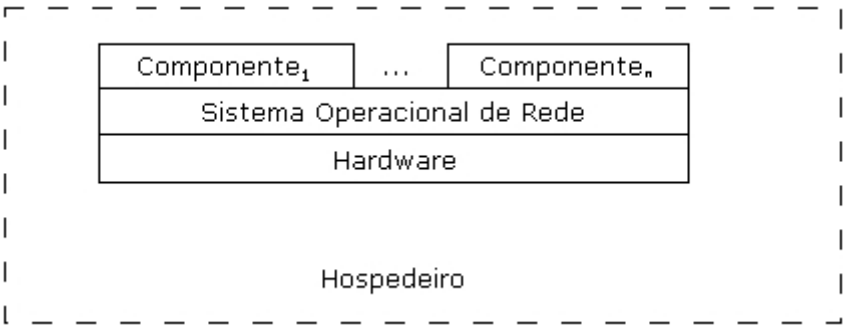

Figura 1.1: Hospedeiro em um Sistema Distribuído [9].

A Figura 1.2 ilustra um sistema distribuído.

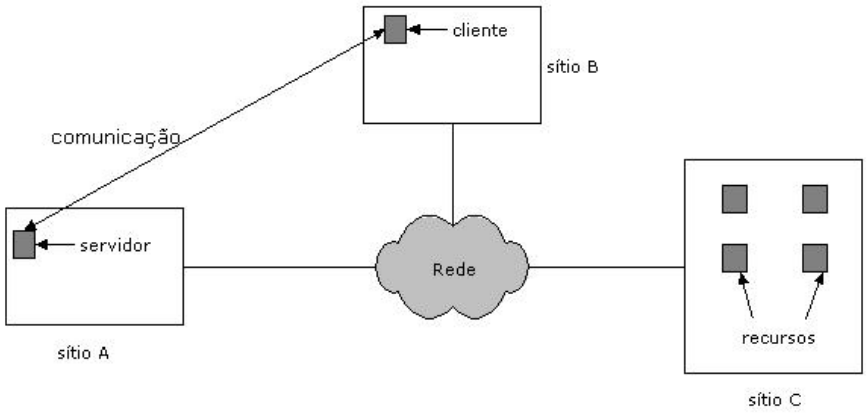

Figura 1.2: Um Sistema Distribuído [23].

Ao se programar um sistema distribuído, é importante levar em consideração os requisitos não-funcionais que são tipicamente adotados nas arquiteturas de sistemas distribuídos [9]:

- escalabilidade: denota a habilidade de acomodar o crescimento da carga computacional. Arquiteturas de sistemas distribuídos suportam escalabilidade utilizando mais do que um hospedeiro.

- *openness:* sistemas abertos podem ser facilemente estendidos e modificados.

Componentes de sistemas distribuídos realizam *openness* por meio da comunicação usando interfaces bem definidas.

- heterogeneidade: a heterogeneidade de um componente surge do uso de diferentes tecnologias para a implementação de serviços, para gerenciamento de dados e para execução de componentes em diversas plataformas. Componentes heterogêneos podem ser muito bem acomodados por sistemas distribuídos.

- Acesso e compartilhamento de recursos: frequentemente, recursos como *hardware*, *software* e dados precisam ser compartilhados por mais do que um usuário. Nestes casos, questões de segurança no acesso aos recursos precisam ser levadas em consideração.

- tolerância a falhas: operações que continuam a executar mesmo na presença de falhas são ditas tolerantes a falhas. Sistemas distribuídos realizam tolerância a falhas por meio de replicação.

Como os recursos compartilhados estarão provavelmente em máquinas com diferentes *hardware* e *software*, a interação direta entre dois ou mais hospedeiros se torna uma tarefa de relativa complexidade. Para resolver tais problemas, é comum adicionar uma camada entre os sistemas operacionais de rede e as aplicações, denominada de *middleware*, como pode ser visto na Figura 1.3, cujo objetivo é aumentar o nível de abstração no desenvolvimento de aplicações distribuídas.

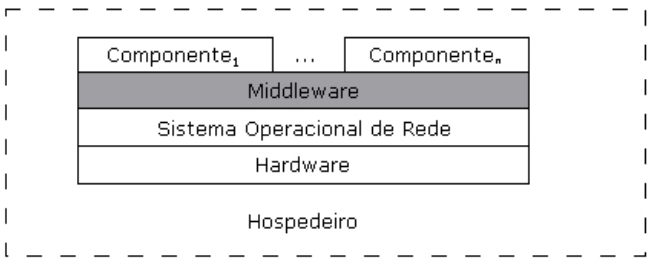

Figura 1.3: *Middleware* em um Sistema Distribuído [9].

Sistemas de *middleware*, tais como CORBA [20], Java RMI (*Remote Method Invocation*) [2, 24] e COM/DCOM [6, 29], tradicionalmente fornecem apenas um

tipo de comunicação entre os nodos, a saber, a comunicação síncrona. Este tipo de comunicação é feita por meio de chamadas de métodos de objetos remotos, sendo que o cliente fica bloqueado à espera do resultado de uma chamada remota. Chamadas síncronas são adequadas principalmente para redes locais as quais possuem as seguintes características:

- previsibilidade;
- latência com limite superior definido;
- largura de banda confiável e constante.

No entanto, mecanismos de comunicação exclusivamente síncronos não são adequados a novos ambientes de redes, como a própria Internet, redes móveis ou redes sem fios [5, 8]. Estas novas redes são caracterizadas por flutuações constantes na largura de banda disponível e por uma elevada latência. Com isso, torna-se difícil estipular um limite superior para envio de uma mensagem e recepção de seu resultado o qual seria utilizado para sinalizar falhas de comunicação com o objeto chamado. Observe que a definição de um limite superior típico de redes locais, da ordem de milisegundos, pode dar origem a sistemas pouco tolerantes a falhas. Por outro lado, caso assuma-se o pior caso e seja definido um limite superior da ordem de segundos, o resultado será uma degradação no tempo de resposta das aplicações. Logo, em ambientes como esses, é interessante sobrepor *computações locais* com *computações remotas* para fins de otimizações, por meio de chamadas assíncronas [31]. Em uma chamada assíncrona, o cliente que invocou um método remoto não fica bloqueado à espera do resultado. Ao contrário, após realizar a chamada, ele continua a execução normal e, em algum ponto mais adiante da execução, pode requisitar pelo valor de retorno.

Existe uma tendência atual de incorporar o mecanismo de comunicação assíncrona a *middlewares* orientados por objetos. Prova disto, é que a última versão de CORBA passou a incorporar chamadas assíncronas de métodos [32]. A implementação oficial de Java RMI, entretanto, ainda não incorporou esta funcionalidade. O modelo computacional de Java RMI permite a um programador invocar métodos em objetos remotos, como se tais invocações fossem locais, tornando-se

fácil de usar. Java RMI pode ser utilizado no desenvolvimento de sistemas puramente Java, além de consistir na base de novas tecnologias, tal como Jini [33], um sistema para construção de aplicações distribuídas em redes sujeitas a constantes reconfigurações, como em redes móveis, por exemplo.

1.1 Objetivos

O objetivo deste trabalho é investigar, especificar e desenvolver um sistema capaz de introduzir o mecanismo de comunicação assíncrona em Java RMI. A idéia é implementar uma extensão de Java RMI com suporte a chamadas assíncronas, denominada FlexRMI (*Flexible* RMI) [28]. Procuramos também manter a compatibilidade daquele sistema com este último. Sendo assim, FlexRMI corresponde a um modelo híbrido no qual é possível invocar métodos remotos síncrona ou assincronamente. Nada impede, por exemplo, que um método seja chamado sincronamente em algum ponto do programa e assincronamente em outro. Caberá ao programador a decisão de escolher como a chamada será realizada.

1.1.1 Requisitos da Implementação de FlexRMI

A ferramenta FlexRMI deverá atender os seguintes requisitos:

1. Ausência de alterações na linguagem Java: FlexRMI não deverá alterar a sintaxe nem a semântica de Java. Os programas escritos deverão ser compatíveis com o compilador disponível no J2SDK.

2. Ausência de alterações em RMI: FlexRMI deverá utilizar os mesmos mecanismos de Java RMI para comunicação via rede entre os componentes cliente/servidor.

3. Resultados via *polling*: FlexRMI deverá implementar o mecanismo de comunicação assíncrona via *polling*, descrito na Seção 2.4.3.

4. Resultados via *callback*: FlexRMI deverá implementar o mecanismo de comunicação assíncrona via *callback*, descrito na Seção 2.4.4.

5. Integração com sistemas RMI síncronos: programas escritos em Java RMI deverão executar normalmente em FlexRMI. Deverá ser possível ainda, realizar chamadas assíncronas a métodos de sistemas RMI síncronos.

1.2 Principais Contribuições

Este trabalho de dissertação de mestrado apresenta cinco contribuições principais:

1. Uma avaliação das diferentes tecnologias utilizadas no desenvolvimento de sistemas distribuídos. São descritos vários aspectos de cada uma das tecnologias, como arquitetura do sistema, modelos de referência, protocolos de comunicação e questões de segurança. Além disso, uma análise crítica e comparativa elucida os pontos positivos e negativos, deixando claro onde e quando utilizar cada tecnologia.

2. O projeto e implementação de uma extensão de Java RMI com suporte a chamadas assíncronas de métodos a qual, além de representar uma alternativa para os desenvolvedores de sistemas distribuídos em Java, apresenta detalhes interessantes em sua implementação, utilizando conceitos avançados da linguagem Java.

3. A introdução de um novo modelo de programação, uma vez que o sistema implementado é uma alternativa para a programação artesanal, na qual os programadores são obrigados a trabalhar em baixo nível e tratar peculiaridades dos objetos.

4. Uma demonstração da viabilidade de se estender o potencial de Java RMI com a introdução de assincronismo. Esta demonstração foi realizada pela implementação de um pequeno experimento e avaliação do tempo de execução do mesmo.

5. Implementação do modelo de *polling* e *callback* para comunicação assíncrona, de forma a oferecer aos usuários de FlexRMI um ambiente simples e que

ao mesmo tempo permita incorporar o assincronismo a suas aplicações distribuídas. Em particular, o modelo de *callback* não foi implementado em nenhum sistema estudado, exceto em uma versão mais recente de CORBA.

1.3 Organização da Dissertação

Este trabalho encontra-se organizado conforme descrito a seguir:

- O Capítulo 2 apresenta um estudo dos diferentes *middlewares* existentes para a construção de sistemas distribuídos. É apresentado ainda um estudo sobre os modelos síncronos e assíncronos de invocação de métodos, além de um estudo da funcionalidade de reflexão computacional em Java a qual foi utilizada na implementação de FlexRMI. Finalizando o capítulo, é apresentada uma revisão bibliográfica sobre os trabalhos que tratam de chamadas assíncronas de métodos. Além de apresentar cada trabalho, é feita uma avaliação crítica das soluções existentes.

- O Capítulo 3 traz uma descrição de como utilizar FlexRMI. São mostrados exemplos que ilustram detalhadamente como um programador deve proceder para implementar sistemas que empregam as funcionalidades de FlexRMI.

- O Capítulo 4 descreve várias decisões tomadas para implementar FlexRMI.

- O Capítulo 5 mostra um exemplo de aplicação distribuída em FlexRMI, bem como uma análise do desempenho da mesma.

- O Capítulo 6 apresenta as conclusões desta dissertação e perspectivas de trabalhos futuros.

Capítulo 2

Revisão da Literatura

2.1 *Middlewares* para Construção de Sistemas Distribuídos

2.1.1 Java RMI

Java RMI [24, 30] é um sistema de objetos distribuídos para a plataforma Java. Especificamente, Java RMI permite aos desenvolvedores manipular objetos remotos muito similarmente ao modo como objetos locais são manipulados, escondendo todos os detalhes de implementação. Em linhas gerais, para se programar em Java RMI só é preciso importar um pacote, procurar pelo objeto remoto em um registro e garantir a captura de `RemoteException` quando um método de um objeto remoto for invocado.

No modelo de objetos distribuídos de Java, um **objeto remoto** é um objeto cujos métodos podem ser chamados de uma outra Máquina Virtual Java (JVM) e, até mesmo, de um outro nodo da rede. Objetos remotos são descritos por uma ou mais interfaces, correspondendo a interfaces Java que declaram os métodos do objeto remoto. A interface serve como um contrato para assegurar a consistência de tipos entre os objetos cliente e servidor.

As interfaces e classes responsáveis pela especificação do comportamento remoto do sistema RMI são definidas nos pacotes `java.rmi` e `java.rmi.server`. A Figura 2.1 mostra o relacionamento entre estas interfaces e classes.

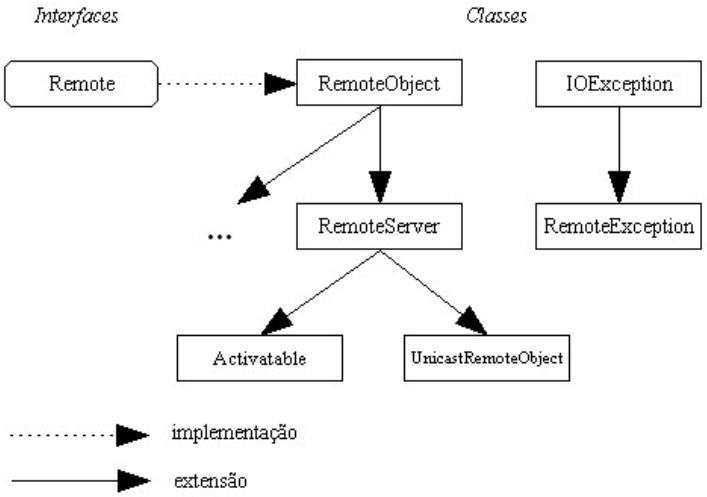

Figura 2.1: Relacionamento entre Interfaces e Classes em Java RMI [24].

Java RMI introduziu um novo modelo de objetos que estendeu o modelo de objetos utilizado até o JDK 1.1, o **Modelo de Objetos Distribuídos de Java**. A seguir, apresenta-se uma comparação do modelo de objetos distribuídos e o modelo de objetos Java [30, 35]. As semelhanças entre os modelos são:

- uma referência a um objeto remoto pode ser passada como um argumento ou retornada como resultado em qualquer método chamado, seja ele local ou remoto;
- um objeto remoto pode ser modelado[1] para qualquer conjunto de interfaces remotas suportado pela implementação usando a sintaxe de modelagem padrão de Java;
- o operador `instanceof` pode ser usado para testar interfaces remotas suportadas por um objeto remoto.

As diferenças básicas entre os modelos são:

[1]Do inglês, *cast*.

- clientes de objetos remotos interagem com interfaces remotas e não com as classes que implementam estas interfaces;
- clientes devem tratar exceções adicionais para cada chamada remota de métodos. Isto é importante porque falhas em uma chamada remota são mais complexas do que falhas em uma chamada local;
- a semântica de passagem de parâmetros é diferente em chamadas a objetos remotos. Em particular, Java RMI utiliza o serviço de serialização de objetos que converte objetos Java em uma cadeia serial de tal forma que o estado do objeto possa ser preservado e recuperado após a transmissão em uma rede;
- a semântica dos métodos de `Object` é definida de acordo com o modelo de objetos remotos;
- mecanismos extras de segurança são introduzidos para controlar as atividades que um objeto remoto pode executar em um sistema. Um objeto `Security Manager` precisa ser instalado para verificar todas as operações de um objeto remoto no servidor.

2.1.1.1 Interface `java.rmi.Remote`

Em Java RMI, uma interface remota é uma interface que declara um conjunto de métodos que podem ser invocados em uma JVM remota. Uma interface remota deve ter as seguintes propriedades [24, 35]:

- Uma interface remota deve estender, direta ou indiretamente, a interface `java.rmi.Remote`. Esta interface é abstrata e não possui métodos.
- Cada declaração de método em uma interface remota deve seguir as seguintes regras:
  - A declaração de um método remoto deve incluir a exceção `java.rmi.RemoteException`. Se `RemoteException` for ativada durante uma chamada remota, então alguma falha de comunicação ocorreu durante a chamada. Estas falhas não podem ser escondidas do programador, uma vez que elas não podem ser mascaradas pelo sistema operacional [34].

  - Em uma declaração de método remoto, um objeto remoto declarado como um parâmetro ou valor de retorno deve ser declarado usando-se a interface remota e não a classe que implementa a mesma.

- Uma interface remota também pode estender qualquer outra interface não-remota desde que todos os métodos da interface estendida satisfaçam aos requisitos da declaração de métodos remotos.

  Por exemplo, o trecho de código a seguir define uma interface remota para uma conta bancária que contém métodos que efetuam depósitos e saques, e que calculam o saldo bancário:

```
import java.rmi.*;
public interface ContaBancaria extends Remote {
   public void deposito (float quantia) throws RemoteException;
   public void saque (float quantia) throws RemoteException;
   public float saldo () throws RemoteException;
}
```

2.1.1.2 Classe `RemoteException`

A classe `java.rmi.RemoteException` é a superclasse de todas as exceções que podem ser ativadas pela JVM durante a chamada de um método remoto. Para assegurar a robustez das aplicações que utilizam Java RMI, cada método remoto declarado em uma interface remota deve especificar `java.rmi.RemoteException` na cláusula *throws*.

A exceção `java.rmi.RemoteException` é ativada quando a chamada a um método remoto falha por alguma razão. Possíveis causas de falhas são:

- Falha de comunicação: servidor não disponível, conexão encerrada pelo servidor, problemas no canal de comunicação, etc.

- Falha durante o processo de *marshalling* ou *unmarshalling*[2] de parâmetros ou valores de retorno.

- Erros no protocolo.

[2]Java RMI utiliza a abstração de *marshall stream* para transmitir objetos de um espaço de endereçamento para outro.

A classe `RemoteException` é uma exceção verificada[3], ou seja, deve ser manipulada pelo chamador de um método remoto e é verificada pelo compilador, não sendo, portanto, uma exceção do tipo `RuntimeException`.

2.1.1.3 Classe `RemoteObject` e suas Subclasses

Funções do servidor RMI são fornecidas pela classe `java.rmi.server.RemoteObject` e por suas subclasses:

- `java.rmi.server.RemoteServer`;
- `java.rmi.server.UnicastRemoteObject`;
- `java.rmi.activation.Activatable`.
- A classe `java.rmi.server.RemoteObject` provê implementações para os seguintes métodos de `java.lang.Object`: `hashCode()`, `equals()` e `toString()`.
- Os métodos necessários para criar objetos remotos e exportá-los são fornecidos pelas classes `UnicastRemoteObject` e `Activatable`. A subclasse identifica a semântica para referências remotas, por exemplo, se o servidor é um simples objeto remoto ou se é um objeto remoto ativado automaticamente, ou seja, é executado quando invocado.
- A classe `java.rmi.server.UnicastRemoteObject` define um objeto remoto *singleton* cujas referências são válidas somente enquanto o processo servidor está executando.
- A classe `java.rmi.activation.Activatable` é uma classe abstrata que define um objeto remoto ativável que inicia a execução quando seus métodos remotos são invocados.

2.1.1.4 Implementação de Interfaces Remotas

Há duas formas de implementar uma interface remota, tal como a interface `ContaBancaria` mostrada na Seção 2.1.1.1. O modo mais simples é implementar

[3]Do inglês, *checked exception*.

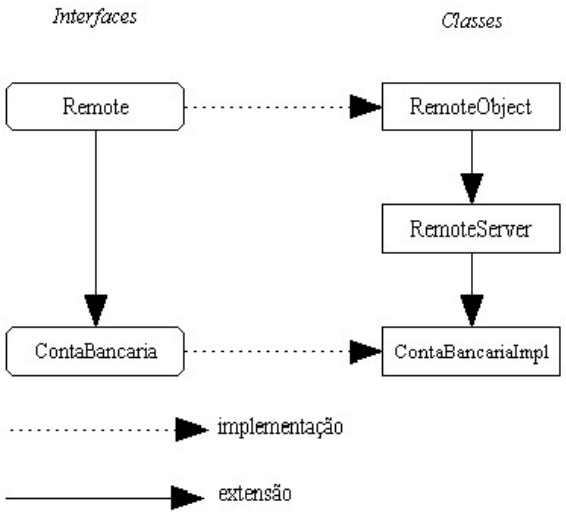

Figura 2.2: Reuso de Implementação Remota [35].

uma classe, por exemplo, `ContaBancariaImpl`, que estende a classe `RemoteServer`. Este esquema é chamado de reuso de implementação remota. A Figura 2.2 mostra a interface e a hierarquia de classes para interfaces e implementações remotas neste esquema.

O construtor padrão para `RemoteServer` é responsável por disponibilizar o objeto remoto para os clientes, por meio da exportação da implementação de tais objetos. A classe `RemoteObject` sobrecarrega métodos herdados de `Object` de modo que a semântica se torne compatível com o modelo de objetos distribuídos.

No segundo esquema de implementação, chamado de reuso de implementação local, a implementação da classe para o objeto remoto não estende `RemoteServer` mas pode, quando apropriado, estender qualquer outra implementação local de classe. Contudo, a implementação deve explicitamente exportar o objeto para torná-lo acessível remotamente.

A Figura 2.3 ilustra o esquema de reuso de implementação local. Neste caso, uma classe pode reusar o código da implementação de tal forma que não é necessário que ela trate os detalhes de como instâncias desta classe podem ser acessíveis remotamente. O processo de exportação é feito pelo construtor `RemoteServer` usado no primeiro esquema.

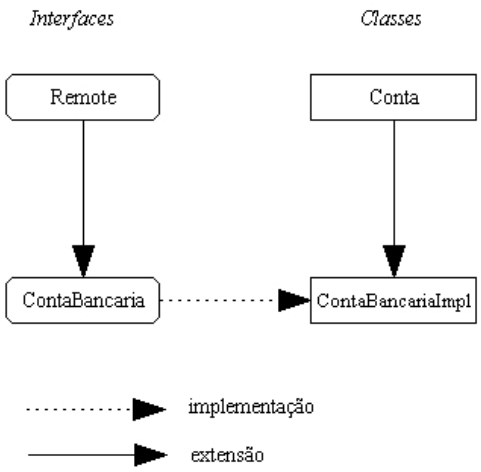

Figura 2.3: Reuso de uma Classe com Implementação Local [35].

De um modo geral, podemos enumerar as seguintes regras para implementar uma interface remota:

1. A classe geralmente estende `java.rmi.server.UnicastRemoteObject` e, portanto, herda o comportamento remoto fornecido pelas classes: `java.rmi.server.RemoteObject` e `java.rmi.server.RemoteServer`.

2. A classe pode implementar um número qualquer de interfaces remotas.

3. A classe pode definir métodos que não estão na interface remota, isto é, que podem ser usados somente localmente e que não estão disponíveis remotamente.

Por exemplo, a classe `ContaBancariaImpl` mostrada a seguir, implementa a interface remota `ContaBancaria` e estende a classe `java.rmi.server.UnicastRemoteObject`:

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ContaBancariaImpl
      extends UnicastRemoteObject
      implements ContaBancaria {
```

```
   private float saldo = 0.0;

   public ContaBancariaImpl (float saldoInicial)
         throws RemoteException {
      saldo = saldoInicial;
   }

   public void deposito (float quantia)
         throws RemoteException {
      ...
   }

   public void saque (float quantia)
         throws SemFundoException, RemoteException {
      ...
   }
   ...
}
```

É importante ressaltar que, se necessário, uma classe que implementa uma interface remota pode estender outras classes que não seja `java.rmi.server.UnicastRemoteObject`. Contudo, a implementação da classe deve assumir a responsabilidade pela exportação do objeto e pela implementação, se necessária, da semântica remota correta dos métodos `hashCode()`, `equals()` e `toString()`, herdados da classe `java.lang.Object`.

2.1.1.5 Passagem de Parâmetros em Java RMI

Um parâmetro de qualquer tipo de Java pode ser passado em uma chamada remota, incluindo tipos primitivos de Java e objetos remotos ou não-remotos de Java.

A semântica da passagem de parâmetros em chamadas remotas é a mesma para passagem de parâmetros em chamadas locais, exceto por:

- objetos não-remotos contidos em um parâmetro de uma chamada remota são passados por cópia, isto é, os objetos são serializados usando o mecanismo de serialização da plataforma Java;

- objetos não-remotos retornados como resultado de uma chamada remota também são passados por cópia.

Quando um objeto não-remoto é passado em uma chamada remota, o estado do objeto não-remoto é copiado antes de se fazer a chamada. Portanto, não há nenhuma relação entre o objeto não-remoto que o cliente possui e o objeto enviado para o servidor em uma chamada.

Contudo, quando um objeto remoto é passado como parâmetro ou retornado de uma chamada, o seu próprio *stub* [4] é passado do cliente para o servidor.

2.1.1.6 Localização de Objetos Remotos

Um nome de servidor é fornecido para se armazenar referências a objetos remotos. Uma referência a um objeto remoto pode ser armazenada usando a interface baseada em URL (*Uniform Resource Locator*) `java.rmi.Naming`.

Para um cliente chamar um método em um objeto remoto, ele primeiro precisa obter uma referência ao objeto. Tal referência é obtida geralmente como um valor de retorno em uma chamada de um método. Desta forma, é possível obter objetos remotos em vários hospedeiros. A interface `Naming` provê métodos baseados em URL para procurar, associar, reassociar, desassociar e listar os pares nome/objeto em *hosts* e portas particulares.

O exemplo a seguir mostra como associar e procurar por objetos remotos:

```
ContaBancaria conta = new ContaBancariaImpl();
URL url = new URL("rmi://pegasus/conta");

//associa url ao objeto remoto
java.rmi.Naming.bind (url, conta);

//  ...
// procura pela conta
conta = java.rmi.Naming.lookup(url);
```

[4] Um *stub* de um objeto remoto é o *proxy* do objeto remoto no cliente. Tal *stub* implementa todas as interfaces que são suportadas pela implementação do objeto remoto.

2.1.1.7 Arquitetura do Sistema Java RMI

O sistema Java RMI consiste em um modelo composto por três camadas, a saber:

1. *stubs/skeletons*;
2. camada remota de referência;
3. camada de transporte.

Classes *stub* funcionam como *proxies* locais para objetos remotos. Classes *skeleton* funcionam como *proxies* remotos. As duas classes implementam a interface remota do objeto servidor. Um cliente chamando um método de um objeto servidor remoto utiliza, na verdade, um *stub* ou *proxy* para o objeto remoto. Uma referência do cliente para um objeto remoto é uma referência para um *stub* local. O *stub* local se comunica com um *skeleton* remoto, que por sua vez, se comunica com o objeto servidor. Em uma chamada de método remoto, o objeto servidor retorna um valor para o objeto *skeleton*, o objeto *skeleton* retorna o valor para o objeto *stub* e o objeto *stub* retorna o valor para o cliente. A Figura 2.4 ilustra o uso de *stubs* e de *skeletons*.

A camada de referência remota suporta comunicação entre *stubs* e *skeletons*. Se o *stub* se comunica com mais de uma instância de *skeleton*, diz-se que o objeto *stub* se comunica com múltiplos *skeletons* de maneira *multicast*. Contudo, a API de Java RMI atualmente disponível somente define classes que suportam comunicação *unicast* entre um *skeleton* e um *stub*. A camada de referência remota pode também ser usada para ativar objetos servidores quando estes são invocados remotamente.

A camada de referência remota em um hospedeiro local se comunica com a camada de referência de um hospedeiro remoto via a camada de transporte RMI. A camada de transporte configura e gerencia conexões entre espaços de endereçamento local e remoto, mantém o caminho de objetos que podem ser acessados remotamente e determina quando conexões se tornam inoperáveis.

A Figura 2.5 ilustra a arquitetura de três camadas utilizada para implementar Java RMI. Nesta figura, o objeto cliente invoca métodos do *stub* local de um objeto servidor. O *stub* local utiliza a camada de referência remota para se comunicar com

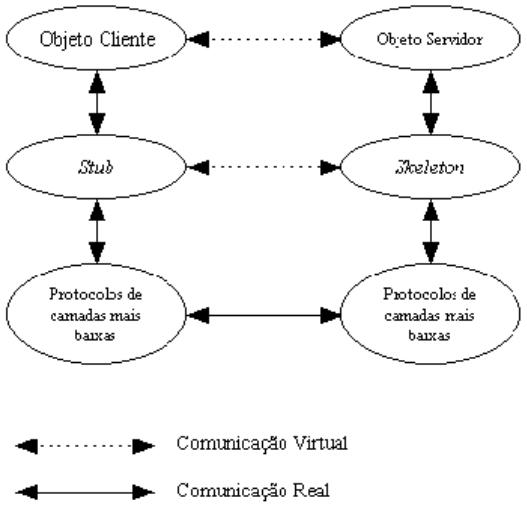

Figura 2.4: O Uso de *Stubs* e *Skeletons* no Modelo de Objetos Distribuídos de Java

skeleton servidor. A camada de referência remota utiliza a camada de transporte para configurar uma conexão entre os espaços de endereçamento local e remoto, e para obter uma referência ao *skeleton* servidor.

2.1.2 CORBA

O *Object Management Group* (OMG) [14, 31, 20] é um consórcio de companhias e de outras organizações fundado em 1989 para promover o desenvolvimento de *software* baseado em componentes por meio do estabelecimento de padrões. Hoje em dia, a OMG possui cerca de 700 membros, entre desenvolvedores, fabricantes e usuários.

CORBA *(Common Object Request Broker Architecture)* é resultado do esforço da OMG no sentido de padronizar o desenvolvimento de sistemas de objetos distribuídos. Sua arquitetura especifica como objetos clientes escritos em uma determinada linguagem podem invocar métodos de objetos servidores desenvolvidos em uma outra linguagem.

A especificação do CORBA 2.0 lançada em 1995 prevê as seguintes características [31]:

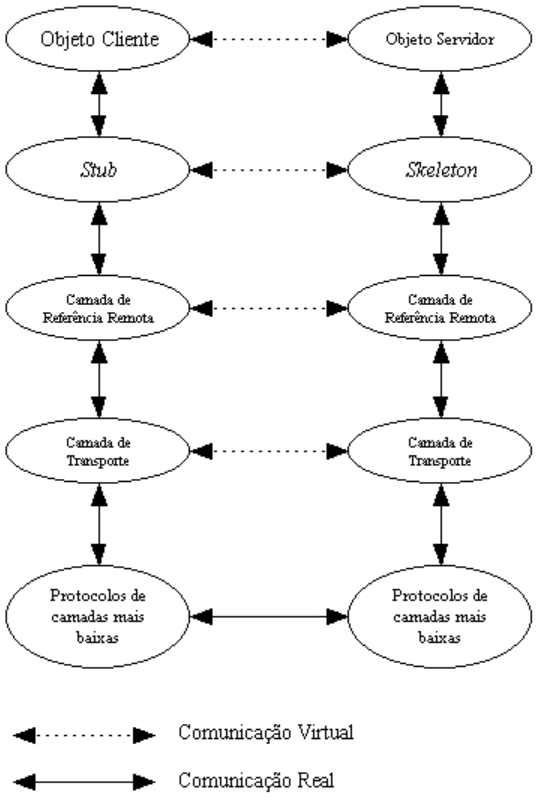

Figura 2.5: O Modelo de Três Camadas de Java RMI

- *Núcleo ORB*
- *Protocolos Intra-ORB*
- *Linguagem para Definição de Interface (OMG IDL)*
- *Repositório de Interface*
- *Language Mappings*
- *Stubs e Skeletons*
- *Interface de Invocação Dinâmica*
- *Adaptadores de Objetos (OA)*

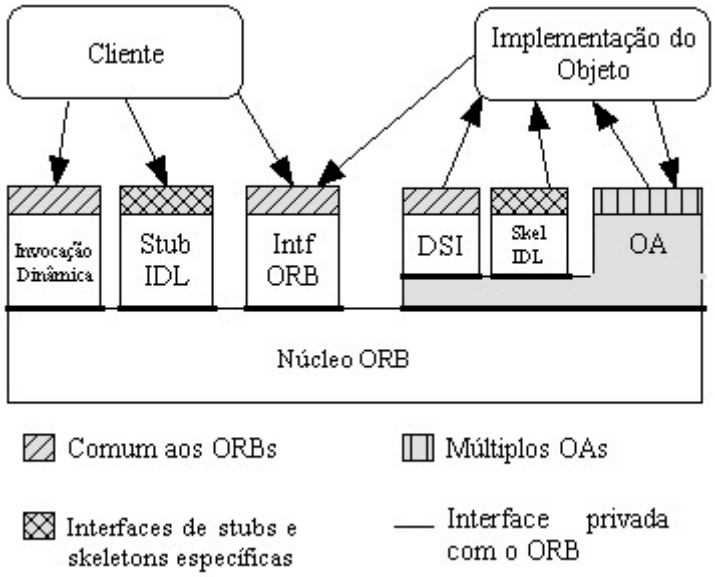

Figura 2.6: Arquitetura do *Common Object Request Broker* [31].

Em 1998, a especificação do CORBA 3.0 foi lançada, adicionando as seguintes características [32]:

- *Adaptador de Objetos Portáveis*
- *CORBA Messaging*
- *Passagem de Objetos por Valores*

A Figura 2.6 ilustra os vários componentes de CORBA, bem como as relações entre eles. Nas próximas seções, discutem-se as principais características de CORBA.

2.1.2.1 Núcleo ORB e Protocolos Intra-ORBs

O *Object Request Broker* (ORB) é o componente responsável pelas funcionalidades de comunicação entre clientes e servidores. Existem quatro categorias de interface de objeto [31]:

1. Serviços (*Object Services*)

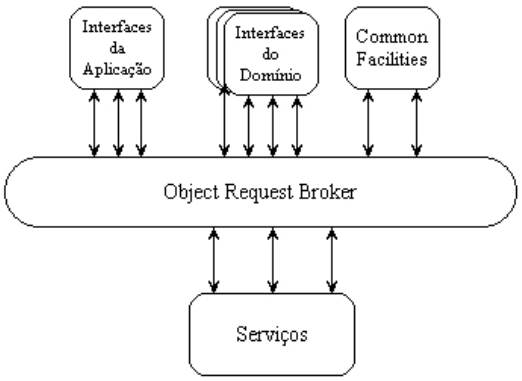

Figura 2.7: Modelo de Referência OMA [31].

2. Facilidades Gerais (*Common Facilities*)

3. Interfaces do Domínio (*Domain Interfaces*)

4. Interfaces da Aplicação (*Application Interfaces*)

A Figura 2.7 ilustra a relação das categorias de interface com o ORB.

Como o ORB é o principal componente no processo de comunicação, é sua função entregar requisições aos objetos e retornar a resposta aos clientes que fizeram a requisição. A transparência na comunicação entre clientes e servidores é possível devido às interfaces escritas em uma linguagem neutra. Para um cliente, detalhes como localização, implementação, estado de execução e mecanismos de comunicação são totalmente abstraídos.

O Protocolo Geral Intra-ORBs *(GIOP - General Inter-ORB Protocol)* é a interface comum usada para suportar a comunicação entre ORBs. O GIOP especifica uma sintaxe e um conjunto de formatos de mensagens para a comunicação entre ORBs. Desta forma, ORBs podem ser implementados em redes que utilizam uma variedade de protocolos de transportes, tal como TCP/IP, IPX ou SAX.

O Protocolo de Internet Intra-ORB *(IIOP- Internet Inter-ORB Protocol)* é usado para mapear o protocolo GIOP para o protocolo TCP/IP. Diferentes ORBs podem se comunicar entre si em uma rede TCP/IP usando os protocolos GIOP e IIOP, como mostrado na Figura 2.8

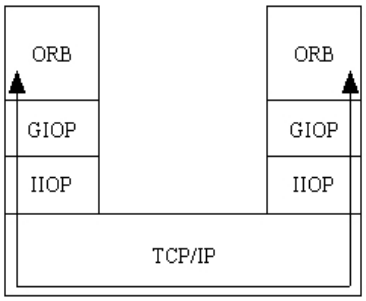

Figura 2.8: Comunicação entre ORBs Usando GIOP e IIOP em uma Rede TCP/IP

2.1.2.2 *Stubs* e *Skeletons*

CORBA utiliza *stubs* e *skeletons* do mesmo modo que Java RMI (v. Seção 2.1.1). Um *stub* é um *proxy* local para o objeto remoto. Ele contém as mesmas interfaces do objeto remoto, mas executa na máquina cliente. Um *skeleton* é uma interface remota para a implementação do objeto servidor. Ele executa no mesmo computador do objeto servidor e provê uma interface entre a implementação do objeto servidor e outros objetos.

Stubs e *skeletons* são conectados via ORB, sendo que o ORB transmite chamadas do *stub* para o *skeleton* por meio de um objeto especial chamado de Adaptador de Objeto (OA)[5]. A Figura 2.9 mostra uma visão geral de como objetos clientes e servidores se comunicam. É importante notar que quando um objeto cliente chama um método do objeto servidor, a invocação é feita via *stub* local do objeto cliente.

O *stub* local comunica-se com um ORB fornecendo a este último, o nome do objeto, o método sendo chamado e quaisquer argumentos para a invocação do método. O ORB encontra o objeto referenciado na rede e acessa o OA que estiver executando na mesma máquina do servidor. O objeto servidor processa o método e retorna os resultados ao *skeleton*, que por sua vez transmite ao ORB [14].

[5]Do inglês, *Object Adapter*.

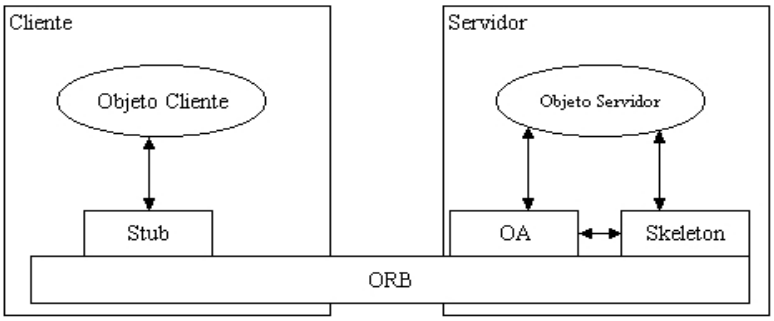

Figura 2.9: Comunicação entre Cliente e Servidor Utilizando CORBA

2.1.2.3 Interface de Invocação Dinâmica

Stubs e *skeletons* permitem invocação estática. Por outro lado, CORBA provê também mecanismos para invocação dinâmica por meio de duas interfaces:

- Interface de Invocação Dinâmica (DII) - corresponde a um *stub genérico* e suporta invocações dinâmicas das requisições de clientes.
- Interface de *Skeleton* Dinâmico (DSI) - corresponde a um *skeleton genérico* e provê disparo[6] dinâmico para objetos.

DII permite aos clientes de uma aplicação invocar métodos de qualquer objeto sem que seja necessário conhecer as interfaces deste objeto em tempo de compilação. Além disso, DII é muito útil para programas interativos, tais como *browsers*.

É importante ressaltar que antes que uma requisição seja invocada, os argumentos devem ter sido fornecidos para a mesma por meio de uma invocação de operações diretamente no pseudo-objeto `Request`. Uma vez que o pseudo-objeto `Request` tenha sido criado e os argumentos tenham sido fornecidos, existem três formas de execução de uma invocação:

- **Invocação Síncrona**: a requisição do cliente é feita e ocorre um bloqueio na execução até que a resposta seja obtida. Este é o método mais comum de invocação em aplicações CORBA.

[6]Do inglês, *dispatch*.

- **Invocação Síncrona Adiada**[7]: após a requisição do cliente ser feita, o processamento da computação continua na sequência e, somente mais tarde, a reposta é coletada. Este método permite que um conjunto de requisições sejam feitas em paralelo e que suas respostas sejam coletadas à medida que as mesmas estiverem disponíveis[8].

- **Invocação Unidirecional**[9]: a requisição do cliente é feita e o processamento continua normalmente, uma vez que não há resposta.

DSI permite que servidores sejam escritos sem precisar dos *skeletons* para os objetos que estão sendo invocados estaticamente em tempo de compilação. Neste caso, um objeto deve estar apto a funcionar tanto como cliente quanto como servidor, ou seja, deve traduzir requisições de um objeto em um sistema qualquer em requisições CORBA, bem como traduzir requisições CORBA em requisições de um sistema qualquer.

2.1.2.4 Adaptador de Objetos Portáveis

Em CORBA, adaptadores de objetos são a ponte que liga o mundo dos objetos CORBA e o mundo das implementações em várias linguagens de programação, chamadas de *servants*. Um novo padrão para Adaptador de Objetos foi definido na especificação de CORBA 2.2, a partir da qual passou a se chamar de Adaptador de Objetos Portáveis (POA). O termo **portável** refere-se à nova capacidade das aplicações e seus *servants* de serem portáveis entre ORBs de vários fabricantes.

A Figura 2.10 mostra o fluxo de uma requisição entre um cliente e uma aplicação servidora. Os passos são os seguintes:

1. O cliente invoca a requisição usando uma referência a objeto que referencia o objeto servidor.

2. O ORB no servidor recebe a requisição. Parte da requisição é mantida no ORB e corresponde a uma chave do objeto no servidor.

[7]Do inglês, *deferred*.
[8]Alguns autores chamam este tipo de requisições de requisições assíncronas por *polling*. Neste trabalho, utiliza-se esta terminologia
[9]Do inglês, *Oneway Invocation*.

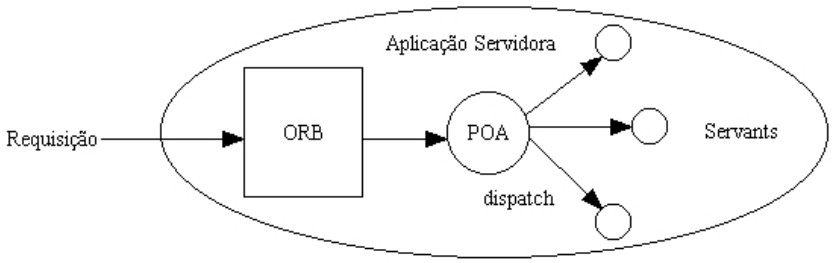

Figura 2.10: Fluxo de uma Requisição em um Servidor POA

3. Parte da chave, chamada de identificador do objeto, é utilizada para determinar a associação entre o objeto servidor e o *servant*.

4. O POA despacha a requisição para o *servant* que, por sua vez, conduz a requisição e entrega os resultados de volta ao POA, que devolve para o ORB e finalmente chega até o cliente.

POA suporta dois tipos de objetos CORBA: o objeto persistente, já existente em versões anteriores, e um objeto transiente. O tempo de vida de um objeto transiente é limitado pelo tempo de vida do POA no qual foi criado. Portanto, objetos transientes são úteis em situações que requerem objetos temporários.

2.1.2.5 *CORBA Messaging*

A especificação de CORBA Messaging [19], adotada pela OMG, adiciona assincronismo na troca de mensagens, invocação independente de tempo[10] e facilidades para especificação de QoS (*Quality of Service*).

Como mostrado na Seção 2.1.2.3, as primeiras versões de CORBA proviam três modelos diferentes para invocação de requisição: síncrona, síncrona adiada e unidirecional. Além de manter estes três modelos, a especificação de CORBA Messaging adicionou dois outros modelos[11]:

[10]Do inglês, *time-independent*.

[11]A ausência de chamadas destes modelos em Java RMI motivou o desenvolvimento desta dissertação.

1. **Callback**: o cliente fornece um parâmetro adicional que corresponde a uma referência a um objeto. Quando a resposta se torna disponível, o ORB utiliza esta referência para entregar a resposta à aplicação.

2. **Polling**: o cliente invoca uma operação, que imediatamente retorna um valor de um tipo especial que pode ser usado para testar a disponibilidade da resposta no futuro.

Outra característica de CORBA Messaging é a possibilidade de permitir aos clientes especificarem o QoS que eles requerem para diversos serviços tais como entrega, enfileiramento e prioridades de mensagens. As políticas de QoS podem ser especificadas em termos de ORB para afetar todas as mensagens, em termos de *threads* para afetar somente as requisições exceutando nesta *thread* ou em termos de referência a objeto para afetar somente aquelas requisições feitas ao objeto.

2.1.2.6 Linguagem para Definição de Interfaces (IDL)

Esta seção trata um dos mais importantes elementos de CORBA, a saber, IDL. Como CORBA possui como filosofia prover uma estrutura na qual objetos criados em várias linguagens de programação possam interagir entre si, é necessário haver um mecanismo que faça a interface entre as diversas linguagens e o ORB. IDL provê exatamente esta interface.

As principais características de IDL são:

1. provê uma forma neutra, ou seja, independente da linguagem de programação, para descrever interfaces dos objetos;

2. descreve uma interface de maneira semelhante à Java, definindo os métodos, os argumentos e os valores de retorno;

3. não trata da questão de implementação das interfaces.

A interface de um objeto servidor é especificada em IDL, cuja compilação produz o *stub* e o *skeleton* que são utilizados por aquele objeto. Compiladores de IDL estão disponíveis para C, C++, Smalltalk, Ada e Java. Estes compiladores traduzem IDL em *stubs* e *skeletons* escritos nestas linguagens fontes. A partir deste

ponto, utiliza-se um compilador específico para compilar os *stubs* e *skeletons* em código binário.

2.1.3 COM/DCOM

A Microsoft introduziu o *Distributed Component Object Model* (DCOM) [6], em 1996, como uma extensão do modelo COM para aplicações distribuídas em rede. O principal objetivo da Microsoft com o modelo COM/DCOM pode ser entendido como um esforço para contornar o fato de que os modelos de objetos até então existentes não eram adequados para o reuso de componentes [9]. Assim como Java RMI, DCOM não possui mecanismos de comunicação assíncronos.

Podemos destacar as seguintes vantagens do DCOM [29]:

- os diversos componentes podem compartilhar a memória entre si, por meio de um novo esquema de gerenciamento;
- transparência de comunicação e interoperabilidade;
- carga e descarga dinâmicas de componentes;

2.1.3.1 Arquitetura do Modelo DCOM

Existem três meios pelos quais o modelo DCOM pode executar as operações de requisição [9]:

- se o objeto servidor estiver disponível em uma biblioteca de ligação dinâmica[12] (DLL) na mesma máquina que o objeto cliente, a DLL é carregada e a requisição é implementada como uma chamada a um método local;
- se o objeto servidor estiver localizado na mesma máquina, mas executando em um processo diferente (servidor EXE), o DCOM utiliza uma variação do mecanismo de chamada a procedimento remoto (RPC), que não necessita de qualquer rede para o transporte dos objetos;
- se o objeto servidor não estiver disponível localmente, é utilizado o mecanismo de RPC, desta vez completamente funcional.

[12]Do inglês, *dynamic link library*.

A Figura 2.11 mostra a arquitetura genérica do DCOM utilizada para implementar requisições de objetos que se encontram em máquinas remotas.

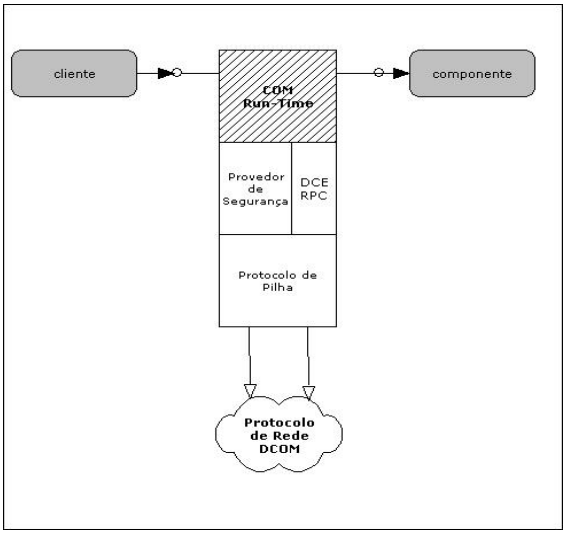

Figura 2.11: Arquitetura DCOM: componentes em máquinas diferentes[30].

A máquina de execução de componentes DCOM provê serviços orientados a objetos para os componentes clientes. São ainda utilizados RPC e um provedor de segurança para gerar pacotes de rede conforme o protocolo de rede em uso.

2.2 Reflexão Computacional e Classes *Proxy* Dinâmicas

Reflexão computacional e classes de *proxy* dinâmicas são dois conceitos fundamentais utilizados na implementação de FlexRMI. A seguir, apresenta-se uma discussão sobre cada um destes conceitos.

2.2.1 Reflexão Computacional

Por meio de reflexão computacional [17], a linguagem Java permite a um programa se auto-examinar e manipular propriedades internas a ele. Por exemplo, é possível a uma classe Java obter os nomes de todos os seus membros em tempo de

execução. O código a seguir mostra a obtenção dos nome de todos os métodos da classe `java.util.Stack`.

```
import java.lang.reflect.*;

public class ListaMetodos {
   public static void main (String args[]) {
      try {
         Class c = Class.forName(args[0]);
         Method m[] = c.getDeclaredMethods();
         for (int i = 0; i < m.length; i++)
            System.out.println (m[i].toString());
      }
      catch (Throwable e) {
         System.out.println (e);
      }
   }
}
```

A saída do programa para a chamada `java ListaMetodos Stack` será:

```
public java.lang.Object java.util.Stack.push(
   java.lang.Object)
public synchronized java.lang.Object
   java.util.Stack.pop()
public synchronized java.lang.Object
   java.util.Stack.empty()
public boolean java.util.Stack.empty() public synchronized
   int java.util.Stack.search(java.lang.Object)
```

As classes de reflexão são encontradas em `java.lang.reflect` e para utilizá-las deve-se:

1. obter um objeto da classe `java.lang.Class` para a classe que se queira manipular;
2. chamar um método para se obter informações sobre a classe que está sendo manipulada, tais como métodos, atributos, superclasses, interfaces, e assim por diante;

3. utilizar a API de reflexão para manipular a informação.

No exemplo a seguir, que lista informações sobre os construtores de uma classe, os três passos citados anteriormente são combinados. Temos, então:

```
import java.lang.reflect.*;

public class Construtor1 {
   public Construtor1()
   {
   }

   protected Construtor1 (int i, double d)
   {
   }

   public static void main (String args[])
   {
      try {
         Class cls = Class.forName("Construtor1");
         Constructor ctorlista[]
               = cls.getDeclaredConstructor(); //construtores
         for (int i = 0; i < ctorlista.length; i++) {
            Constructor ct = ctorlista[i];
            // nomes dos construtores
            System.out.println ("nome = " + ct.getName());
            // declaracoes
            System.out.println (decl classe = " +
                                ct.getDeclaringClass();
            Class pvec[] = ct.getParameterTypes();
            for (int j = 0; j < pvec.length; j++)
               System.out.println ("param #" + j + " " + pvec[j]);
            // excecoes
            Class evec[] = ct.getExceptionTypes();
            for (int j = 0; j < evec.length; j++)
               System.out.println ("excecao #" + j + " " +
                                   evec[j]);
            System.out.println ("-------");
         }
      }
      catch (Throwable e) {
         System.err.println(e);
      }
```

```
    }
}
```

Como estamos interessados em informações sobre construtores, nenhuma informação sobre tipo de retorno é listada. Para outros tipos de métodos, é possível ainda obter esta informação.

A execução do exemplo possui a seguinte saída:

```
nome = construtor1
decl classe = class construtor1
-------
nome = construtor1
decl classe = class construtor1
param #0 int
param #1 double
-------
```

2.2.2 Classes *Proxy* Dinâmicas

Uma classe *proxy* dinâmica [26, 25] é uma classe que implementa uma lista de interfaces especificadas em tempo de execução tal que a invocação de um método de uma das interfaces é codificada e despachada para um outro objeto por meio de uma interface uniforme. Em outras palavras, invocações de métodos de objetos da classe *proxy* dinâmica resultam na chamada de um método de um manipulador de chamadas[13] e são codificadas em um objeto da classe `java.lang.reflect.Method`, que identifica o método que foi chamado, e mais um vetor de objetos do tipo `Object` contendo os argumentos. Desta maneira, os métodos da classe `Proxy` podem ser examinados a fim de se obter os argumentos, registrar frequência e tempo das chamadas entre outros, antes e depois da chamada do método original.

A partir da versão 1.3 do J2SE (*Java Standard Edition*), uma classe *proxy* dinâmica pode ser encontrada no pacote `Reflection` de Java. Classes e instâncias de Proxy são criadas utilizando métodos `static` da classe `java.lang.reflect.Proxy`. Dados o carregador[14] e um vetor de interfaces, o método `Proxy.getProxyClass`

[13] Do inglês, *invocation handler*.
[14] Do inglês, *loader*.

retorna um objeto de `java.lang.Class` para uma classe *proxy*. A classe *proxy* estará definida no carregador de classe especificado e implementará todas as interfaces fornecidas [13].

Cada classe `Proxy` possui um construtor público que recebe um argumento que corresponde a uma implementação da interface `java.lang.reflect.InvocationHandler`, além de um carregador e um vetor de interfaces. O construtor é dado por:

```
public static Object newProxyInstance (ClassLoader loader,
                                       Class[] interfaces,
                                       InvocationHandler h)
                              throws IllegalArgumentException
```

Por exemplo:

```
public class Mediador
implements java.lang.reflect.InvocationHandler {
   public static Object newInstance (Object obj) {
      return java.lang.reflect.Proxy.newProxyInstance (
                obj.getClass().getClassLoader(),
                obj.getClass().getInterfaces(),
                new Mediador(obj));
      }
      ...
   }
}
```

Tendo sido criada a instância do *proxy*, é permitido modelá-la em qualquer uma das interfaces que a implementam e chamar qualquer método que seja suportado pela interface. Por exemplo:

```
Alo a = (Alo) Mediador.newInstance(new AloImpl());
a.aloMundo();
```

Invocações de métodos de objetos da classe *proxy* dinâmica são despachadas para o método `invoke` do manipulador de chamadas. Haverá um objeto da classe `java.lang.reflect.Method` encapsulando o método chamado e um vetor de objetos da classe `Object` contendo os argumentos. A implementação do método `invoke` varia de acordo com o interesse e a necessidade de cada implementador.

A seguir, é mostrado um exemplo de programa que faz interposição por meio de uma classe *proxy* dinâmica.

- Arquivo Principal - **Exemplo.java**:

```
01 public class Exemplo {
02
03   public static void main(String[] args) {
04
05      Alo a = (Alo) Mediador.newInstance(new AloImpl());
06      a.aloMundo (2003);
07      a.adeusMundo (2004);
08   }
09 }
```

A saída da classe Exemplo é:

```
Cheguei ao mundo em 2003
Deixei o mundo em 2004
```

- Arquivo de definição da interface **Alo**:

```
01 import java.io.*;
02
03 public interface Alo {
04   public void aloMundo (int ano);
05   public void adeusMundo (int ano);
06 }
```

- Arquivo de implementação da interface **Alo**:

```
01 import java.io.*;
02
03 public class AloImpl implements Alo {
04   public void aloMundo(int i) {
05
06      System.out.println ("Cheguei ao mundo em " + i);
07
08   }
09
```

```
10   public void adeusMundo(int i) {
11
12      System.out.println ("Deixei o mundo em " + i);
13
14   }
15 }
```

- Arquivo da classe *proxy* dinâmica **Mediador.java**:

```
01 import java.lang.reflect.*;
02
03 public class Mediador
04 implements java.lang.reflect.InvocationHandler {
05
06   private Object obj;
07
08   public static Object newInstance (Object obj) {
09
10      return java.lang.reflect.Proxy.newProxyInstance (
11                obj.getClass().getClassLoader(),
12                obj.getClass().getInterfaces(),
13                new Mediador(obj));
14
15   }
16
17   public Mediador (Object obj) {
18
19      this.obj = obj;
20
21   }
22
23   public Object invoke (Object proxy,
24                         Method m,
25                         Object[] args)
26                     throws Throwable {
27
28      Object result;
29      try {
30
31         // recupera o nome do metodo chamado
32         System.out.println ("inicio do metodo " +
33                              m.getName());
34
```

```
35          // imprime lista de argumentos do metodo chamado.
36          // Um detalhe importante e´ a possibilidade de alteracao
37          // do vetor contendo os argumentos.
38          System.out.println (" - Argumentos - ");
39          for (int i = 0; i < args.length; i++) {
40
41             System.out.println (" " + args[i].toString());
42
43          }
44
45          result = m.invoke(obj, args);
46
47       } catch (InvocationTargetException e) {
48          throw e.getTargetException();
49       } catch (Exception e) {
50          throw new RunTimeException (" excecao - " +
51          e.getMessage());
52       } finally {
53          System.out.println ("fim do metodo " + m.getName());
54       }
55       return result;
56    }
57 }
```

2.3 Modelos de Comunicação

Por se tratar de uma questão de grande importância no presente trabalho, nesta seção apresenta-se um estudo mais teórico sobre os modelos de comunicação síncrona e assíncrona [7].

2.3.1 Modelos de Comunicação Síncrona e Assíncrona

Os diferentes tipos de *middlewares* vistos na Seção 2.1 tradicionalmente trabalham com o **modelo de comunicação síncrona**. A principal característica de uma comunicação síncrona é o fato do cliente ficar bloqueado enquanto o servidor estiver executando a computação solicitada pelo mesmo. Seja **A** um processo que esteja precisando de um determinado serviço fornecido pelo processo **B**, ou seja, o processo **A** é o cliente e o processo **B**, o servidor. Inicialmente, o controle da

execução é exercido por **A** que ao invocar um serviço em **B**, transfere o controle da execução para este. Somente duas situações tornam possível a transferência do controle da execução de **B** para **A**:

- a execução em **B** terminou e o resultado foi retornado para **A**;
- a execução em **B** falhou e **A** é notificado da ocorrência de falha.

Embora o modelo de execução síncrona seja utilizado em muitos casos, há situações em que o tempo que o cliente fica esperando até receber a resposta do servidor pode degradar o desempenho do sistema como um todo. Por exemplo, considere o caso em que um componente de interface com o usuário requisite operações no servidor. Se o tempo de execução destas operações ultrapassar o limite estabelecido pelos requisitos de tempo de resposta, o modelo síncrono não pode ser usado. Considere ainda uma situação mais crítica. Se várias operações independentes devem ser executadas por diferentes servidores, pode ser adequado que estas operações sejam executadas simultaneamente a fim de aproveitar a vantagem da distribuiçao dos servidores. Com a execução síncrona, o cliente pode requisitar a execução de uma única tarefa por vez. Nestes casos, o **modelo de comunicação assíncrona** é mais adequado.

A seguir, consideram-se as diferentes abordagens que podem ser utilizadas quanto ao aspecto de sincronização da execução de processos [9].

2.4 Sincronização em Chamadas de Métodos

2.4.1 Requisições Síncronas

Requisições síncronas foram discutidas na Seção 2.1.2. Praticamente, todos os *middlewares* orientados a objetos suportam requisições deste tipo. A Figura 2.12 apresenta um diagrama de seqüência em UML [3] para ilustrar a abordagem síncrona.

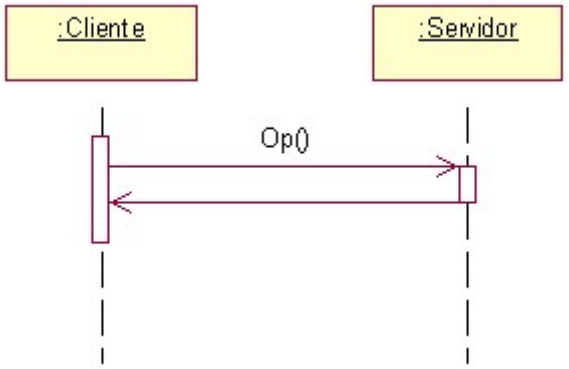

Figura 2.12: Requisições Síncronas [9].

2.4.2 Requisições Unidirecionais

Requisições unidirecionais[15] retornam o controle ao cliente tão logo os *middlewares* tenham aceito a requisição. A operação requisitada e o cliente são executados concorrentemente e não são sincronizados. Requisições unidirecionais podem ser usadas se não há necessidade por parte do cliente de esperar pelo resultado da operação. Isto significa que a semântica do cliente não depende do resultado da operação requisitada. Este caso ocorre quando a operação não produz um resultado que seja utilizado pelo cliente e que não possa violar a integridade do servidor por meio da propagação de exceções. O diagrama de seqüência da Figura 2.13 ilustra as requisições unidirecionais.

2.4.3 Requisições Assíncronas por *Polling*

Assim como as requisições unidirecionais, requisições assíncronas por *polling* retornam o controle ao cliente tão logo a requisição tenha sido aceita pelo *middleware*. Entretanto, requisições assíncronas por *polling* também podem ser utilizadas em situações onde haja a necessidade de se transferir o resultado para o cliente. Uma vez retornado o controle ao cliente, o mesmo pode consultar o resultado mais tarde. O diagrama da Figura 2.14 exemplifica o processo de requisições assíncronas por *polling*.

[15]Do inglês, *oneway requests*.

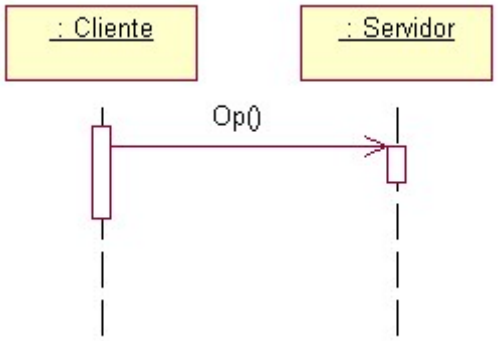

Figura 2.13: Requisições Unidirecionais [9].

Uma desvantagem de requisições assíncronas por *polling* é que os clientes ficam encarregados da sincronização com o servidor quando o resultado se fizer necessário. Caso o resultado ainda não esteja disponível, o cliente é bloqueado até recuperar o resultado ou uma nova inspeção sobre a disponibilidade do resultado deve ser feita mais adiante.

2.4.4 Requisições Assíncronas por *Callback*

A necessidade da operação de *polling* pode ser evitada utilizando requisições assíncronas por *callback*. Quando um cliente utiliza requisições assíncronas por *callback*, ele também obtém o controle tão logo a requisição tenha sido aceita pelo *middleware*. A operação é executada pelo servidor sendo que, ao término da operação, ele explicitamente invoca um método no cliente por meio do qual o resultado é transferido do servidor para o cliente. Esta operação é denominada de *callback* e é ilustrada pela Figura 2.15.

2.5 Sistemas de Chamada Assíncrona de Métodos Remotos

A construção de ambientes que permitem chamada assíncrona de métodos remotos já foi assunto de várias pesquisas presentes na literatura [11, 16, 21, 15, 10,

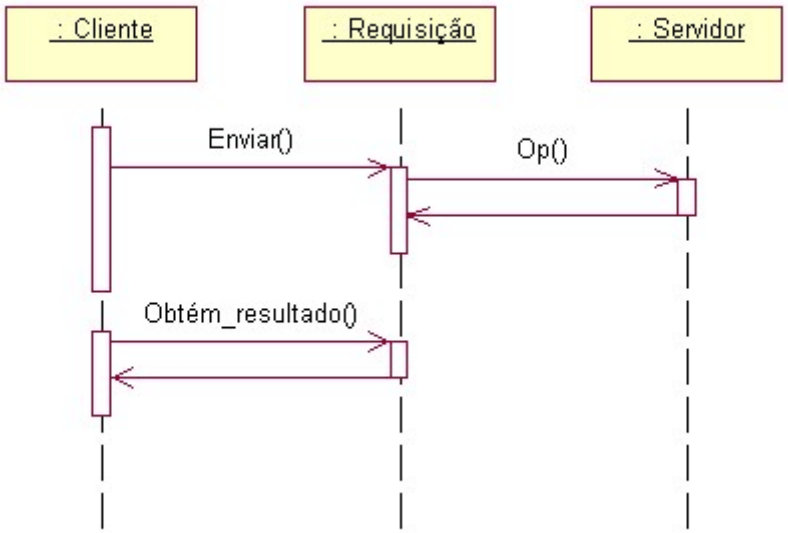

Figura 2.14: Requisições Assíncronas por *Polling* [9].

22, 18, 4].

2.5.1 MultiLisp

Multilisp [11] é uma versão estendida da linguagem de programação Scheme [1]. Usada para manipulação simbólica de programas, Multilisp incorporou ainda construções para expressar concorrência em programas.

A principal construção disponível em Multilisp para a criação e gerenciamento de programas concorrentes é o *future*. A construção `future X` imediatamente retorna um objeto `future` para o valor da expressão `X` e concorrentemente inicia a avaliação de `X`. Inicialmente indeterminado, o objeto `future` se torna determinado quando o valor de `X` for computado.

Em Multilisp, há dois tipos de operações: o primeiro tipo é composto por operações que necesitam conhecer o valor de um objeto `future` indeterminado. Neste caso, a execução é suspensa e só recomeça quando o objeto `future` se tornar determinado. O segundo tipo refere-se a operações que não precisam conhecer o valor de seus operandos e que podem, então, executar com objetos `future` indeterminados, introduzindo um alto grau de paralelismo nos programas.

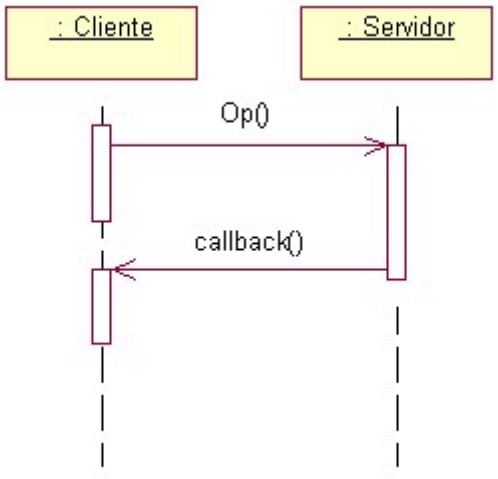

Figura 2.15: Requisições Assíncronas por *Callback* [9].

Um tipo de concorrência bastante comum, que ocorre entre a computação de um valor e a disponibilidade ou uso deste valor, pode ser tratada em Multilisp utilizando objetos `future`. Por exemplo, a expressão `(pcall cons A B)` avalia concorrentemente `A` e `B` e constrói uma estrutura contendo os dois valores. Durante a criação da estrutura de dados, não é necessário conhecer os valores reais de `A` e `B`. Na verdade, pode-se trabalhar com a idéia de *contratos futuros* ou de *promessa de entrega* para os valores `A` e `B`. Caso esta estrutura de dados não seja utilizada imediatamente, é possível introduzir concorrência entre a avaliação de `A` e `B` e operações que se seguem após a construção da estrutura de dados. Desta forma, o paralelismo pode ser obtido escrevendo-se `(cons (future A) (future B))`.

2.5.2 Promises

Liskov [16] mostra o projeto da construção `promises` que foi influenciada pelo mecanismo da construção `future` de Multilisp. Assim como em `future`, `promises` permite que o resultado de uma chamada seja demandado e utilizado em algum ponto mais adiante no programa. A construção `promises` é fortemente tipada, portanto elimina a necessidade de verificação de tipos em tempo de execução. Permitem ainda a propagação de exceções de maneira mais conveniente.

`Promises` está diretamente associada com o desenvolvimento de *call-stream* como mecanismo de comunicação que pode ser utilizado em várias plataformas. *Call-streams* utiliza o mecanismo de chamada de procedimento remoto e troca de mensagens. Chamadas de procedimentos remotos são simples de compreender e explicitam claramente o envio e recepção de mensagens. Porém, em chamadas remotas, ocorre o bloqueio da entidade que invoca um método remoto até que este último retorne um valor.

A introdução de *streams* em uma linguagem de programação conduz ao problema de composição das mesmas. Por exemplo, em

```
a := p(x)
b := q(y)
```

temos duas chamadas que ocorrem em um mesmo *stream*, sendo que é desejado que `q` execute independente de `p` ter terminado ou não a sua execução. A introdução de `promises` permite que se indique que:

1. `p` e `q` devem executar paralelamente;
2. resultados das duas chamadas sejam recuperados sem erro ou confusão;
3. exceções sejam tratadas.

Desta forma, `promises` se mostra como uma boa solução para a introdução de chamadas assíncronas de procedimentos remotos. Entretanto, seu modelo computacional é de muito baixo nível incluindo, por exemplo, a manipulação de portas de comunicação e o tratamento de interrupções.

2.5.3 Extensões de RMI com Chamada Assíncrona de Métodos

A extensão de Java RMI com chamadas assíncronas proposta por Raje [21] é um sistema para Java no qual a execução de métodos remotos é sobreposta com a execução de métodos locais, permitindo ao cliente realizar outras tarefas enquanto o objeto remoto está executando o método remoto. Para tanto, é introduzido

um novo mecanismo para retorno de valor durante a chamada assíncrona de um método.

Assim como em MultiLisp, descrito na Seção 2.5.1, o conceito de `future` é utilizado, de tal forma que quando uma chamada assíncrona é feita, um objeto do tipo `Future` é criado e retornado ao cliente. Este objeto, ao término da computação remota, recebe o valor de retorno. O cliente pode, então, recuperar este valor ou pode bloquear a execução caso o mesmo não esteja disponível ainda.

A extensão proposta por Karaorman [15] pode referenciar ou invocar métodos de objetos remotos de modo síncrono ou assíncrono, usando a mesma sintaxe como se os objetos fossem locais. Para tanto, foi construído um conjunto de classes e de ferramentas, além de um novo método de projeto que provêm abstrações de alto nível para a construção de aplicações distribuídas.

São três as principais abstrações fornecidas:

- objetos ativos remotos com escalonamento feito pelo cliente;
- invocação assíncrona de métodos e sincronização não bloqueante por meio do uso de manipuladores de chamadas;
- criação transparente de objetos remotos.

Entretanto, o sistema A-RMI se apresenta bastante extenso. Por exemplo, é fornecido um manipulador de chamadas para cada tipo primitivo de Java, adicionando muitas classes ao modelo original. Isto torna difícil a adaptação de aplicações já existentes ao modelo proposto.

Falkner [10] apresenta uma extensão da implementação padrão de Java RMI por meio de um novo mecanismo para a geração de *stubs*, incluindo protocolos de comunicação adicionais. Para indicar quais métodos utilizarão comunicação assíncrona, são usados objetos `future` definindo uma pseudo-interface Java com as palavras-chaves adicionais `async` e `future`.

Como é necessário alterar a definição das interfaces dos programas Java RMI e gerar novos *stubs* para indicar chamadas assíncronas, a utilização desta extensão com sistemas RMI síncronos fica limitada aos casos em que o código-fonte esteja disponível.

A Tabela 2.1 apresenta uma comparação dos três sistemas apresentados anteriormente.

	Extensões de RMI com Chamadas Assíncronas		
	Raje	Falkner	Karaorman
Alterações em Java	Não	Sim	Não
Alterações em RMI	Sim	Sim	Sim
Resultados via *Polling*	Sim	Sim	Sim
Resultados via *Callback*	Não	Não	Não
Integração com sistemas RMI síncronos	Não	Não	Não

Tabela 2.1: Comparação entre Extensões Assíncronas de Java RMI.

2.5.4 Chamadas Assíncronas em CORBA

Além de Java RMI, outros trabalhos propõem soluções para o problema de comunicação assíncrona em outras tecnologias. Schmdit e Vinoski [22] desenvolveram um trabalho cujo modelo utilizado é o de CORBA. Apesar de CORBA já incluir chamadas assíncronas via *polling*, o sistema implementado se mostra interessante por apresentar a utilização dos dois modelos de comunicação assíncrona: *polling* e *callback*.

2.6 Conclusão

Este capítulo procurou fornecer uma visão geral dos *middlewares* mais utilizados na construção de sistemas distribuídos. Em destaque, foram apresentados Java RMI, utilizado em sistemas puramente Java, e CORBA, um *middleware* amplamente utilizado e independente de linguagem de programação. Além disso, o capítulo procurou fornecer subsídios teóricos sobre tópicos utilizados nesta dissertação. Neste sentido, foram mostrados os conceitos de reflexão computacional e classes de *proxy* dinâmicas. Discutiram-se ainda alguns aspectos relacionados aos modelos de comunicação e de sincronização em chamadas de métodos. Por fim, apresentou-se uma revisão dos trabalhos relacionados à chamada assíncrona de métodos presentes na literatura.

Capítulo 3

Descrição de FlexRMI

3.1 Introdução

Esta seção apresenta um pequeno tutorial de como utilizar o sistema FlexRMI. FlexRMI é um mecanismo que permite invocar métodos de objetos existentes em um outro espaço de endereçamento, que pode estar ou não na mesma máquina. A principal vantagem de FlexRMI sobre Java RMI é a possibilidade de poder invocar assincronamente métodos remotos, usando dois mecanismos distintos: *polling* e *callback*.

3.2 Utilização de FlexRMI

Há basicamente três elementos que devem ser considerados quando é feita uma chamada a métodos remotos:

1. O **cliente** é o processo que invoca um método de um objeto remoto.

2. O **servidor** é o processo que contém o objeto remoto. O objeto remoto é um objeto que estende a classe `UnicastRemoteObject` e que executa no espaço de endereçamento do processo servidor.

3. O **Registrador de Objetos** é o nome do servidor que relaciona nomes a objetos. Objetos são registrados por meio do Registrador de Objetos. Uma

vez registrado, pode-se obter acesso a um objeto remoto usando o nome do mesmo no Registrador de Objetos.

Existem dois tipos de classes que podem ser usadas em FlexRMI:

1. Uma classe remota é uma classe cujos métodos podem ser chamados remotamente. Uma classe remota estende a classe `UnicastRemoteObject` e implementa uma interface que por sua vez estende a interface `Remote`. Um objeto desta classe pode ser referenciado de duas formas:

    1. No espaço de endereçamento onde o objeto foi criado, pode-se usá-lo como qualquer outro objeto.
    2. Em outros espaços de endereçamento, o objeto pode ser referenciado usando um manipulador de objetos[1]. Uma chamada assíncrona retorna um manipulador de objetos, que é um valor do tipo `Promises`, o qual permite inspecionar o estado da chamada. Detalhes da classe `Promises` são apresentados na Seção 3.2.1.

2. Uma classe serializável é uma classe cujas instâncias podem ser copiadas de um espaço de endereçamento para outro. Uma instância da classe serializável pode ser chamada de **objeto serializável**[2]. Se um objeto serializável é passado como parâmetro ou valor de retorno em uma chamada a método remoto, então o valor do objeto é copiado de um espaço de endereçamento para outro.

3.2.1 O Sistema FlexRMI

Nesta seção é apresentada uma descrição das classes `Promises` e `FlexRMI` que fazem parte do sistema FlexRMI. A classe `Promises` permite ao usuário acessar o manipulador de objetos usado em chamadas assíncronas e a classe `FlexRMI` controla o processo de inicialização necessário para o funcionamento do sistema. As demais classes do pacote não são usadas pelos usuários no desenvolvimento de programas que utilizam FlexRMI.

[1] Do inglês, *object handler*.
[2] Do inglês, *serializable object*.

A classe `FlexRMI` possui o seguinte método:

- `public static Object lookup (String name)`: este método é utilizado pelos clientes da aplicação para encontrar o objeto remoto denominado por `name`. Corresponde ao método `Naming.lookup(String name)` de Java RMI.

Os métodos da classe `Promises` são os seguintes:

- `public boolean isAvailable()`: retorna `true` se o resultado associado ao manipulador de objetos estiver disponível. Retorna `false` caso contrário.
- `public Object getResult()`: retorna o valor do manipulador de objetos que corresponde ao resultado de uma chamada assíncrona. Em caso de falhas, o valor retornado é um descendente da classe `Exception`. Além disso, se o valor não estiver disponível, a execução é bloqueada até que o mesmo seja obtido.
- `public void setResult (Object result)`: atribui ao manipulador de objetos o valor do parâmetro `result`.

3.2.2 Classes e Interfaces Remotas

Mostra-se nesta seção como definir uma classe remota. Uma classe remota possui duas partes: a interface e a classe propriamente dita. A interface remota deve possuir as seguintes propriedades:

1. A interface deve ser pública.
2. A interface deve estender a interface `java.rmi.Remote`.
3. Todo método na interface deve declarar que ele ativa a exceção `java.rmi.RemoteException`. Esta exceção é ativada caso haja alguma falha de comunicação com o servidor durante o processamento de uma chamada. Outras exceções também podem ser ativadas.

A classe remota deve ter as seguintes propriedades:

1. Deve implementar a interface remota.

2. Deve estender a classe `java.rmi.server.UnicastRemoteObject`. Objetos de tal classe existem no espaço de endereçamento do servidor e podem ser invocados remotamente.

3. Podem existir métodos que não estão na interface remota. Neste caso, estes métodos podem ser invocados somente localmente.

Diferentemente do caso de classes serializáveis, discutidas na Seção 3.2.3, não é necessário que o cliente e o servidor tenham acesso à definição da classe remota. O servidor requer a definição da classe e da interface remotas, mas o cliente somente usa a interface remota. Em outras palavras, a interface remota representa o tipo de um objeto remoto. Se um objeto remoto está sendo usado remotamente, seu tipo deve ser declarado como do tipo de uma interface remota, e não do tipo de uma classe remota.

3.2.3 Classes Serializáveis

Uma classe é serializável se ela implementa a interface `java.io.Serializable`. Subclasses de uma classe serializável também são serializáveis. Detalhes sobre estas classes podem ser encontrados em [2].

Usar um objeto serializável em uma chamada de método é bem simples. Basta passar o objeto como parâmetro ou valor de retorno. Os programas cliente e servidor devem ter acesso à definição das classes serializáveis que estão sendo usadas. Se os programas cliente e servidor estão em máquinas diferentes, então a definição das classes serializáveis precisa ser copiada de uma máquina para a outra.

3.3 Programando Chamadas Assíncronas via *Polling*

Uma vez discutido como definir classes remotas e serializáveis, o próximo passo é mostrar como programar o cliente e o servidor. Nesta seção, será tratada a

comunicação assíncrona implementada via *polling*, como descrita na Seção 2.4.3.

3.3.1 Programando um Servidor

No programa exemplo mostrado a seguir, têm-se uma classe remota e sua correspondente interface remota que são, respectivamente, chamadas de `Alo_Impl` e `Alo`. A seguir, mostra-se o arquivo `Alo.java`:

```
01 import java.rmi.*;
02 public interface Alo extends Remote {
03    public Promises say() throws RemoteException;
04 }
```

O arquivo `Alo_Impl.java` é exibido abaixo:

```
01 import java.rmi.*;
02 import flexrmi.*;
03 public class Alo_Impl extends UnicastRemoteObject
                   implements Alo {
04    private String mensagem;
05    public Alo_Impl (String msg) throws RemoteException {
06       mensagem = msg;
07    }
08    public Promises say() throws RemoteException {
09       Promises h = new Promises();
10       h.setResult(mensagem);
11       return h;
12    }
13 }
```

É importante notar que o método `say()` é chamado de modo assíncrono, uma vez que seu tipo de retorno foi declarado como sendo do tipo `Promises`, como mostrado na linha 08 do código da classe `Alo_Impl`.

Do lado do servidor, também precisa existir uma classe para instanciar os objetos remotos. Não é necessário que esta classe seja uma classe remota ou serializável, embora o servidor utilize estas classes. Pelo menos um objeto remoto deve estar registrado no Registrador de Objetos. O comando para registrar um objeto é: `Naming.rebind (objectName, object)`, onde `object` é o objeto remoto

que está sendo registrado e `objectName` é a *string* que nomeia o objeto remoto. Um exemplo de tal classe é mostrado a seguir:

```
01 import java.rmi.*;
02 import java.rmi.server.*;
03 public class AloServidor {
04    public static void main (String[] args) {
05       try {
06          Naming.rebind ("Alo", new Alo_Impl ("Alô, mundo"));
07          System.out.println ("Servidor Alô preparado!");
08       }
09       catch (Exception e) {
10          System.out.println ("Falha no servidor Alo: " + e);
11       }
12    }
13 }
```

O Registrador de Objetos, denominado `rmiregistry`, somente aceita requisições para associar e desassociar[3] objetos executando na mesma máquina, de forma que nunca é necessário especificar o nome da máquina quando um objeto está sendo registrado.

O código para criar o servidor pode ser implementado em qualquer classe. No exemplo do servidor anterior, ele foi implementado em uma classe chamada de `AloServidor` que contém somente o programa acima.

Todas as classes e interfaces devem ser compiladas usando-se o `javac`. Além disso, os arquivos com interfaces remotas precisam ser compiladas com o compilador `flexrmic`, disponível no sistema FlexRMI. Detalhes sobre o `flexrmic` podem ser encontrados na Seção 3.3.2. Uma vez compilados, os arquivos *stubs* e *skeletons* devem ser gerados, usando-se o compilador de *stub*, chamado `rmic`. O *stub* e o *skeleton* da interface remota de exemplo são compilados usando-se o seguinte comando: `rmic Alo_Impl`.

3.3.2 Utilizando o Compilador `FlexRMIc`

Em FlexRMI, um método cujo tipo de retorno seja `Promises` vai executar assincronamente como, por exemplo, o método `say()` da classe `Alo_Impl`. No en-

[3]Do inglês, *bind* e *unbind*, respectivamente.

tanto, suponha que existam objetos no servidor que foram implementados sem que seus métodos retornem um objeto da classe `Promises`. A princípio, tais métodos não poderiam executar assincronamente, pois não utilizam a sintaxe utilizada pelo FlexRMI para identificar métodos assíncronos.

Contudo, existem casos como, por exemplo, sistemas de objetos legados cujo código fonte não está disponível, mas que, dependendo da situação, é necessário ou desejável chamar um ou mais métodos destes objetos de forma assíncrona. Como não é possível alterar a implementação de tais métodos, uma vez que esta mudança poderia afetar outros objetos clientes, disponibilizou-se uma ferramenta chamada `flexrmic`, a qual gera uma nova interface, com métodos prefixados com `async_`, os quais são chamados assincronamente. Observe que a implementação do objeto no servidor não é alterada, sendo de responsabilidade do sistema FlexRMI identificar métodos com o prefixo `async_` e invocá-los assincronamente.

O compilador `flexrmic` é usado com a seguinte linha de comando: `FlexRMIc <arquivo_interface>`. Por exemplo, se não tivéssemos acesso ao código-fonte da classe `Alo_Impl`, bastaria invocar `FlexRMIc Alo` para que fosse possível chamar métodos assincronamente. Como saída, gera-se um arquivo com o sufixo `_FlexRMI`, no caso `Alo_FlexRMI.java`, listado a seguir:

```
01 /* Codigo gerado automaticamente pelo FlexRMIc */
02
03 public interface Alo_FlexRMI extends Alo {
04
05    public Promises async_say() throws java.rmi.RemoteException;
06
07    public void async_say(RL_Impl rl) throws java.rmi.RemoteException;
08
09 } /* fim da interface */
```

Repare que a interface Alo_FlexRMI possui a declaração de dois métodos, nas linhas 5 e 7, que serão chamados assincronamente:

- `Promises async_say()`: será chamado assincronamente via *polling*;
- `void async_say(RL_Impl rl)`: será chamado assincronamente via *callback*.

Como dito anteriormente na Seção 3.3.1, este arquivo deve ser compilado com `javac`.

3.3.3 Programando um Cliente

O cliente é um programa em Java como qualquer outro programa. A chamada de um método remoto pode retornar um objeto remoto como seu valor de retorno. O nome de um objeto remoto inclui as seguintes informações:

1. O nome ou endereço da máquina que está executando o Registrador de Objetos na qual o objeto remoto está sendo registrado. Se o Registrador de Objetos está sendo executado na mesma máquina na qual estão sendo feitas requisições, o nome da máquina pode ser omitido.

2. A porta na qual o Registrador de Objetos está esperando por requisições. A porta padrão é 1099. Caso a porta padrão esteja sendo usada, não é necessário incluí-la no nome.

3. O nome local do objeto remoto registrado no Registrador de Objetos.

A seguir, mostra-se um exemplo de um programa cliente:

```
01 public class AloCliente {
02    public static void main (String[] args) {
03       try {
04          Alo alo = (Alo) FlexRMI.lookup ("Alo");
05          Promises h = alo.say();
06
07          /*
08             Realiza outras tarefas. Por exemplo, pode-se ter
09             uma espera ocupada utilizando o método
10             isAvailable().
11          */
12          // Espera pelo valor de retorno
13          h.getResult();
14
15       }
16       catch (Exception e) {
17          System.out.println ("Exceção em AloCliente: " + e);
18       }
19    }
20 }
```

O método `FlexRMI.lookup` obtém um manipulador de objeto do Registrador de Objetos executando na própria máquina, uma vez que nenhuma informação sobre o endereço foi fornecida. Além disso, utiliza-se a porta padrão. É importante ressaltar que o resultado de `FlexRMI.lookup` deve ser convertido[4] para o tipo da interface remota, como feito na linha 4 da classe AloCliente, mostrada acima.

A chamada de método remoto neste exemplo de cliente é `alo.say()`. O mesmo retorna um objeto do tipo `Promises`, uma vez que estamos trabalhando com o assincronismo via *polling*. Caso o método `say()` fosse síncrono, uma chamada como aquela feita na linha 05 retornaria o valor de retorno do próprio método. No entanto, `say()` é um método assíncrono e como tal, pode acontecer do valor não estar disponível quando o mesmo for requisitado. Deste modo, torna-se necessário testar em algum momento se o valor já está disponível na variável do tipo `Promises`. Isto é feito por meio do método `getResult()` e este ponto do programa funciona como um ponto de sincronização. Uma outra alternativa é utilizar o método `isAvailable()` do objeto `h` em uma espera ocupada cujo tempo pode ser utilizado para realizar outras tarefas, enquanto o resultado não estiver disponível. O código para o cliente pode ser colocado em qualquer classe. Neste exemplo, foi colocado em uma classe `AloCliente` que contém somente o programa acima.

3.3.4 Iniciando o Servidor

Antes de iniciar o servidor, é preciso iniciar o Registrador de Objetos e deixá-lo executando em segundo plano. Isto é feito com o comando: `rmiregistry &` .

O servidor pode então ser iniciado, usando o comando: `java AloServidor`.

3.3.5 Executando um Cliente

O cliente executa como qualquer outro programa Java, por meio do comando: `java AloCliente`.

Para executar, é preciso que as classes `AloCliente`, `Alo`, `Alo_FlexRMI`, geradas pelo compilador `flexrmic`, e `Alo_Impl_Stub`, gerada pelo compilador `rmic`, estejam disponíveis na máquina cliente. Em particular, os arquivos `AloCliente.class`,

[4]Do inglês, *cast*.

`Alo.class`, `Alo_FlexRMI.class` e `Alo_Impl_Stub.class` devem estar em algum diretório especificado no CLASSPATH.

3.4 Programando Chamadas Assíncronas via *Callback*

Nesta seção, é apresentado como programar em FlexRMI utilizando o mecanismo de *callback*, descrito na Seção 2.4.4. Ao contrário do mecanismo de *polling*, quando um método é chamado via *callback* nenhum valor é retornado. Entretanto, veremos que é possível registrar um método que deverá ser executado quando o resultado da chamada estiver disponível.

Para exemplificar o uso de *callbacks*, considere o problema de se determinar quais números entre 0 e 999 são primos. Basicamente, tem-se um objeto no servidor contendo um método que determina se um número é primo ou não. No lado do cliente, tem-se um vetor cujo tamanho corresponde à quantidade de números que serão manipulados. Para cada elemento do vetor, é feita uma chamada ao servidor, sendo que há ainda um contador que contabiliza as respostas que são enviadas do servidor para o cliente.

3.4.1 Programando um Servidor

Neste exemplo, a classe remota é representada pela classe `Primo_Impl` e sua interface remota correspondente é `Primo`. A seguir, apresenta-se o arquivo `Primo.java` que contém a interface `Primo`.

```
01 import java.rmi.*;
02 import java.util.*;
03 public interface Primo extends Remote {
04   public boolean ePrimo (int x) throws RemoteException;
05 }
```

O objeto remoto implementa o método `ePrimo()`, como especificado na linha 04 da listagem da interface `Primo`.

O código para a classe `Primo_Impl` é listado a seguir:

```
01 import java.rmi.*;
02 import java.rmi.server.*;
03 public class Primo_Impl extends UnicastRemoteObject implements Primo{
04
05    public boolean ePrimo (int x) throws RemoteException {
06       boolean valorRetorno = true;
07
08       if (x == 0) {
09          valorRetorno = false;
10          return valorRetorno;
11       }
12       if (x == 1) {
13          valorRetorno = true;
14          return valorRetorno;
15       }
16       for (int i = 2; i <= x-1; i++) {
17          if ((x % i) == 0) {
18             valorRetorno = false;
19             break;
20          }
21       }
22
23       return valorRetorno;
24    }
25 }
```

O código para o servidor é o seguinte:

```
01 import java.rmi.*;
02 import java.rmi.server.*
03 public class PrimoServidor {
04    public static void main (String[] args) {
05       try {
06          Naming.rebind ("Primo", new Primo_Impl());
07       }
08       catch (Exception e) {
09          System.out.println ("Falha no servidor Primo: " + e);
10       }
11    }
12 }
```

Como ocorre com o mecanismo de *polling*, todas as classes e interfaces devem ser compiladas. Além disso, os arquivos com interfaces remotas precisam

ser compiladas com o compilador `flexrmic`, disponível no pacote FlexRMI (vide Seção 3.3.2). O *stub* e o *skeleton* da interface remota devem ser compilados com o `rmic`.

3.4.2 Programando um Cliente

O cliente da aplicação `Primo` chama o método remoto `ePrimo()` para cada uma das posições do vetor `boolVetor`. Ao final da execução, **boolVetor[i]** tem um valor `true` se `i` for primo e `false` se não for primo. O código do cliente é listado a seguir:

```
01 import java.util.*;
02 public class PrimoCliente {
03    private static boolean[] boolVetor = new boolean[1000];
04    private static int contador = 0;
05
06    public static void main (String[] args) {
07       try {
08          Primo_FlexRMI primoObj = (Primo_FlexRMI) FlexRMI.lookup ("Primo");
09          for (int i = 0; i < boolVetor.length; i++) {
10             primoObj.async_ePrimo (new RL_Impl(i), i);
11          }
12          while (contador <= (boolVetor.length -1)){
13              // realiza outras tarefas
14          }
15          imprimeVetor();
16       }
17       catch (Exception e) {
18          System.out.println ("Exceção em PrimoCliente: + e.getMessage());
19       }
20    }
21
22    public static void imprimeVetor() {
23       for (int i = 0; i < boolVetor.length; i++) {
24          if (boolVetor[i]) {
25             System.out.println ("Número " + i + " é primo!);
26          }
27          else {
28             System.out.println ("Número " + i + " não é primo!);
29          }
30      }
```

```
31    }
32
33    public synchronized static void setValor (Object x, int indice) {
34       int pos;
35       String valorBool = (String) x;
36
37       contador++;
38       boolVetor[indice] = new Boolean(valorBool).booleanValue();
39   }
40 }
```

No código anterior, a chamada assíncrona remota é realizada na linha 10 por meio da expressão `primoObj.async_ePrimo (new RL_Impl(i), i)`. Observe que esta chamada utiliza o mecanismo de *callback*, uma vez que:

- não é retornado nenhum valor;
- o primeiro argumento para o método é um objeto da classe `RL_Impl`.

Na linha 12, temos uma espera ocupada, onde é verificado o valor da variável `contador`. `Contador` é incrementado toda vez que é feita uma chamada do método `setValor()`. A espera ocupada só termina quando todos os membros do vetor forem processados.

Precisamos agora configurar o método `setValor()` como sendo o método a ser invocado quando um resultado vier do servidor e chegar ao cliente, ou seja, quando a operação de *callback* acontecer. Para tanto, basta implementar a interface `ResultListener`. Esta interface possui apenas o método `resultHandler(Object x)` e, para o nosso exemplo, temos o seguinte código:

```
01 public class RL_Impl implements ResultListener {
02    int indice;
03    public RL_Impl (int indice) {
04       this.indice = indice;
05    }
06    public void resultHandler (Object x) {
07       PrimoCliente.setValor(x, indice);
08    }
09 }
```

Portanto, como mostrado na linha 07, o método `setValor()` da classe PrimoCliente é invocado por ocasião da ocorrência da operação de *callback*.

3.4.3 Executando o Programa

Para executar o programa, devemos:

1. iniciar o servidor, como mostrado na Seção 3.3.4, alterando apenas o nome da classe que será executada;
2. iniciar o cliente, como mostrado na Seção 3.3.5.

3.5 Conclusão

Neste capítulo, apresentou-se o sistema FlexRMI. Dois modelos de programação foram apresentados: *polling* e *callback*. Com *polling*, vimos que o cliente, após realizar a chamada assíncrona, precisa verificar a disponibilidade do resultado da chamada em algum momento mais adiante da execução. Já com *callback*, isto não é necessário, uma vez que neste modelo um método do cliente é automaticamente invocado quando o resultado se tornar disponível.

Capítulo 4

Implementação de FlexRMI

A implementação de FlexRMI foi feita tendo como base toda a estrutura já disponibilizada por Java RMI padrão. Os protocolos de comunicação, que em última análise são utilizados pelos *stubs* e pelos *skeletons*, permanecem inalterados. Tarefas, como manutenção dos *sockets* de comunicação com o servidor remoto, *marshaling* ou serialização de parâmetros para os métodos invocados [10], são realizadas do mesmo modo como em Java RMI.

Dois conceitos principais formam a base de nosso sistema. São eles:

- reflexão computacional;
- *threads*

Uma chamada remota assíncrona é realizada disparando em uma *thread*, que executa em paralelo com a *thread* principal, a computação que deve ser executada. Identificar se a comunicação deve ocorrer síncrona ou assincronamente é uma questão de diferenciar o tipo de retorno da chamada realizada ou uma questão de verificar o prefixo do nome do método , como é mostrado na Seção 4.1.2.1

A seguir, discute-se, em detalhes, alguns pontos importantes da implementação de FlexRMI.

4.1 A Estrutura das Classes de FlexRMI

Nesta seção, são apresentadas as classes que compõem o sistema FlexRMI. São elas: `Promises`, `Mediador`, `AThread` e `FlexRMI`.

4.1.1 A Classe `Promises`

A classe `Promises` representa a estrutura utilizada para armazenar o valor de retorno de chamadas remotas assíncronas do tipo *polling*, como mostrado na Seção 3.3. Para tanto, um objeto do tipo `Promises` é passado como parâmetro durante a criação da *thread* que executará a computação em paralelo.

Este objeto da classe `Promises` deve ser consultado em algum ponto após a chamada, para se descobrir se o valor de retorno já está disponível. Além disso, o objeto pode também armazenar a informação sobre exceções levantadas durante a chamada remota. Em caso positivo, o objeto armazena informações sobre a exceção e, não mais, o valor de retorno da chamada.

4.1.1.1 Implementação da Classe `Promises`

A classe `Promises` é uma extensão da classe `Serializable`, uma vez que objetos desta classe serão passados do cliente ao servidor e do servidor ao cliente. Possui os seguintes atributos:

- `Object value`: contém o valor de retorno de uma chamada remota assíncrona via *polling*.
- `boolean exception`: *flag* que indica se o valor armazenado em `value` representa uma exceção ou não.
- `boolean valueAvailable`: *flag* que indica se a chamada remota foi feita e seu valor de retorno pode ser encontrado em `value`.

A classe *Promises* possui métodos para configurar os *flags* `exception` e `valueAvailable`. No entanto, seus principais métodos são aqueles que manipulam o atributo `value`. Têm-se:

- `Object getResult()`: este método é responsável por retornar o valor do atributo `value` do objeto `Promises` corrente. Este método funciona como um sincronizador, uma vez que só se retorna dele após o valor de retorno da chamada estiver disponível. Para tanto, implementa-se uma espera ocupada, como mostrado a seguir:

```
while (!isAvailable()) {
    /* nao faz nada, apenas espera pelo valor */
}
```

- `void setResult(Object result)`: atribui ao atributo `value` o valor passado como parâmetro em `result`.

- `void copyPromises(Object source)`: copia o valor armazenado em `source` para o atributo `value`.

4.1.2 A classe `Mediador`

A classe `Mediador` é responsável por implementar um objeto da classe *proxy* dinâmica utilizado para interceptar as chamadas de métodos e distingui-las entre síncronas e assíncronas. Uma classe *proxy* dinâmica é uma classe que implementa uma lista de interfaces especificadas em tempo de execução tal que a invocação de um método em um objeto possa ser capturada e despachada para um outro objeto por meio de uma interface uniforme.

É importante ressaltar que o objeto mediador precisa se registrar para poder capturar as chamadas realizadas. Em nossa implementação, no momento em que o cliente obtém a referência a um objeto remoto, automaticamente é criado um mediador que trabalhará especificamente para este objeto remoto. Para tanto, o método `lookup` da classe `Naming` foi encapsulado em um novo método `lookup` presente na classe `FlexRMI`.

4.1.2.1 Implementando Chamadas Assíncronas

Nesta seção é mostrado como foram implementadas as chamadas assíncronas. Como dito anteriormente, o objeto mediador é responsável por interceptar as cha-

madas e decidir se serão chamadas síncronas ou assincronas.

Em linhas gerais, pode-se dizer que uma chamada remota feita por meio de Java RMI consiste na comunicação entre dois objetos, sendo que um objeto, o cliente, invoca métodos de um outro objeto, o servidor. A Figura 4.1 ilustra este processo, onde o cliente A requisita a execução do método `s()` no servidor B.

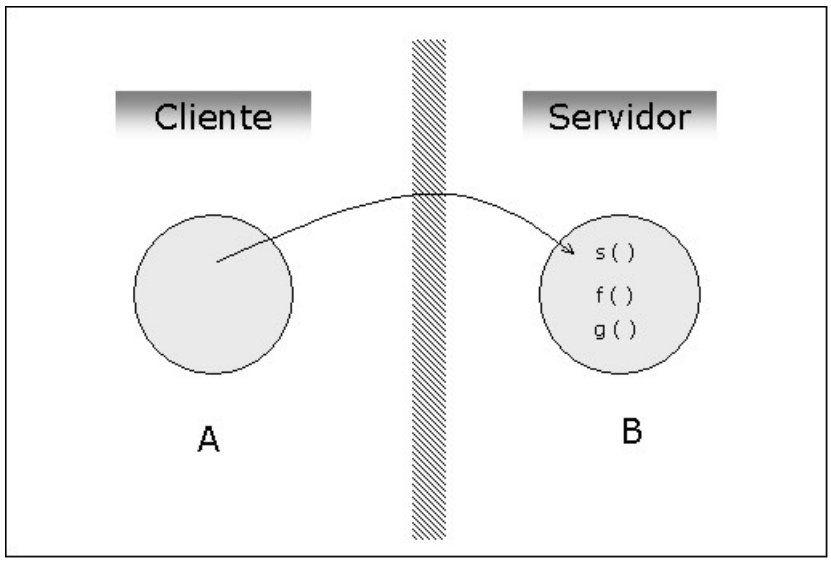

Figura 4.1: Comunicação via Java RMI

Podemos notar que, por meio de Java RMI, chamadas remotas podem ser feitas como se fossem chamadas locais, uma vez que detalhes tais como *sockets* e protocolos de comunicação são totalmente abstraídos. Entretanto, para a implementação de FlexRMI, foi necessário considerar o mecanismo de comunicação em um nível mais baixo. Na verdade, a comunicação entre os objetos cliente e servidor acontece por meio de objetos *stubs* e *skeletons*, que são responsáveis por todas as tarefas relacionadas à camada de rede, como pode ser visto na Figura 4.2.

A solução adotada para implementar chamadas assíncronas remotas consistiu em inserir uma nova camada entre a camada da aplicação e a camada de *stubs/skeletons*. Deste modo, toda chamada referente ao objeto remoto passará necessariamente pelo objeto mediador, como ilustrado na Figura 4.3.

Portanto, fica a cargo do mediador distinguir entre chamadas síncronas e assíncronas. Uma chamada assíncrona pode ser identificada por dois modos:

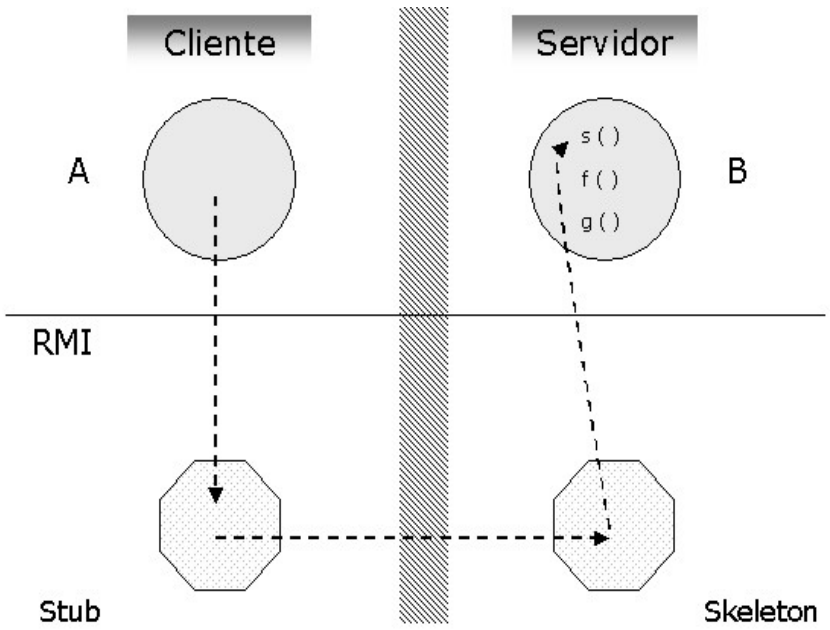

Figura 4.2: Comunicação via Java RMI Utilizando *Stubs* e *Skeletons*

1. Tipo de retorno `Promises`.

2. Nome do método prefixado por `async_`.

Identificada a chamada assíncrona, o mediador cria uma nova *thread* que fica responsável por se comunicar com o objeto *stub*. Assim que esta *thread* for instanciada, o controle da execução volta ao cliente que pode realizar novas chamadas.

Quando for identificada uma chamada síncrona, o mediador será responsável por se comunicar com o objeto *stub*. Neste caso, o mediador ficará bloqueado até que o resultado da chamada esteja disponível.

Por exemplo, na Figura 4.4, o método `g()` do objeto B foi chamado assincronamente pelo objeto A. Como a chamada é assíncrona, o mediador ainda fica livre após esta chamada. Porém, chamando o método `s()` sincronamente, tem-se o bloqueio do mediador que ficará à espera do resultado da chamada a `s()` para poder realizar outras chamadas.

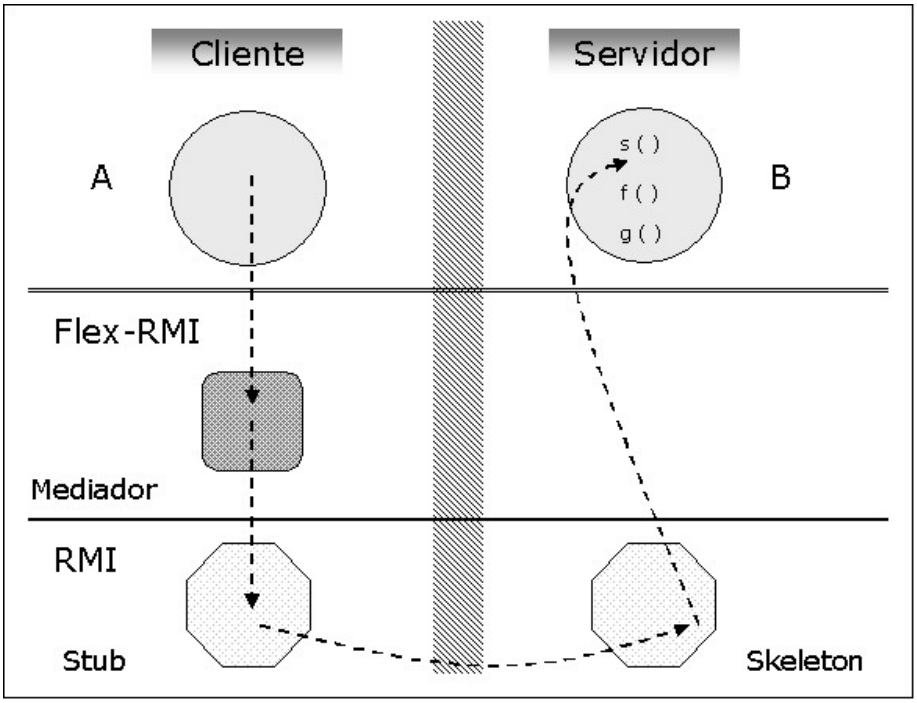

Figura 4.3: Comunicação via FlexRMI

4.1.2.2 Implementação da Classe `Mediador`

A implementação da classe `Mediador` utilizou amplamente o conceito de reflexão computacional. Seu principal método, cuja assinatura é `public Object invoke (Object proxy, Method m, Object[] args)` é descrito em detalhes.

O método `invoke()` será executado toda vez que um método for chamado no cliente. Entretanto, a computação só será feita em métodos que executarão assincronamente. Para tanto, verifica-se o prefixo do nome do método, bem como o tipo de retorno do mesmo. Sendo assim, métodos assíncronos são identificados pelo seguinte trecho de código:

```
String tipoRetorno = tipoRetorno().toString();
if ((m.getName().indexOf("async_") != -1) ||
    (tipoRetorno.equals("class Promises"))) {
    ...
}
```

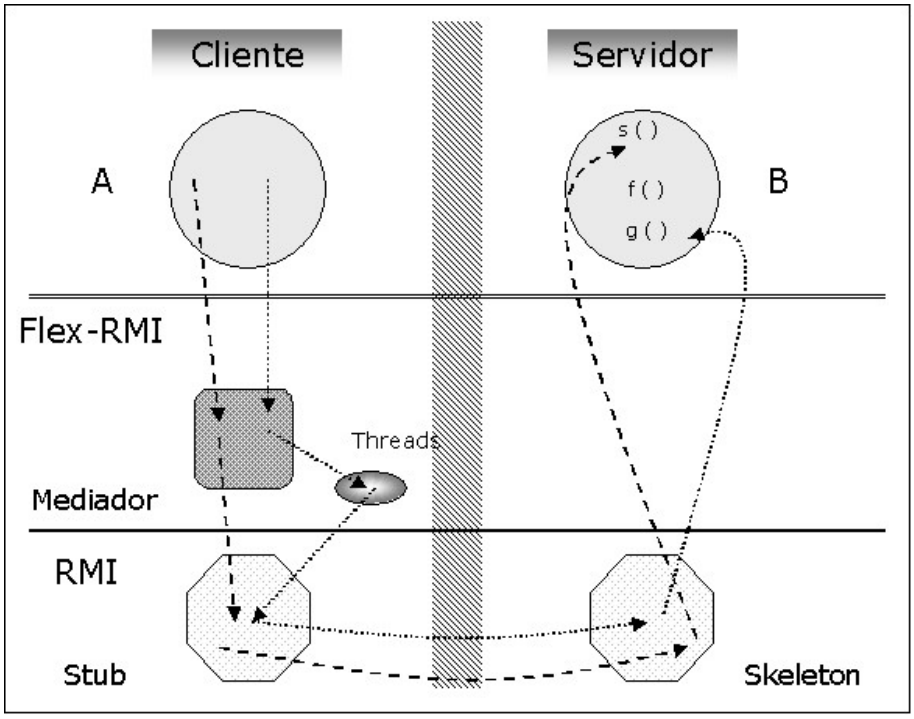

Figura 4.4: Comunicações Síncrona e Assíncrona via FlexRMI

Uma vez identificado que o método será executado assincronamente, o próximo passo é decidir se o assincronismo será via *polling* ou via *callback*. No caso de *callback*, o primeiro argumento será do tipo da classe que implementa a interface `ResultListener`, no caso `RL_Impl`. A identificação é feita pelo seguinte trecho de código:

```
Class[] argsTipos = m.getParameterTypes();
String primeiroArgumento = "";
if (argsTipos.length > 0) {
   primeiroArgumento = (String) argsTipos[0].toString();
}
```

A seguir, é preciso manipular a lista de argumentos de modo que seja possível invocar um método implementado no servidor. Sendo assim, é preciso alterar a lista de argumentos no caso de *callback*, eliminando o primeiro argumento. Tem-se:

```
if (primeiroArgumento.equals("class RL_Impl")){
   argsTiposCallback = new Class[argsTipos.length-1];

   for (int i = 1; i < argsTipos.length; i++)
      argsTiposCallback[i-1] = argsTipo[i];

   argsTipos = new Class[argsTiposCallback.length];

   for (int i = 0; i < argsTiposCallback[i]
      argsTipos[i] = argsTiposCallback[i];
}
```

Neste ponto, o atributo `argsTipos` conterá a lista de argumentos que deverá ser passada como parâmetro em chamadas via *callback*. O próximo passo é montar o nome correto do método que será invocado, e finalmente realizar a chamada.

```
String nomeMetodo = m.getName();
int k = nomeMetodo.indexOf("async_");
if (k != -1)
   nomeMetodo = nomeMetodo.substring(k+6, nomeMetodo.length());
if (primeiroArgumento.equals("classRL_Impl")) {

   Object[] argsTemp;

   isCallBack = true;
   resultListener = (RL_Impl) args[0];
   argsTemp = new Object[args.length-1);

   for (int i = 1; i < args.length; i++)
      argsTemp[i-1] = args[i];

   args = new Object[argsTemp.length];

   for (int i = 0; i < argsTemp.length; i++)
      args[i] = argsTemp[i];

   /* Chamada do método */

   if (isCallback)

      /* callback */
      c = new AThread (promises, obj, m2, args, resultListener);
```

```
    else

      /* polling */
      c = new AThread (promises, obj, m2, args, null);

    c.start ();
}
```

4.1.2.3 Implementação da Classe `AThread`

A classe `AThread` estende a classe `Thread` permitindo que chamadas assíncronas possam ser simuladas. Cada chamada assíncrona será feita por uma *thread* específica. O seu método principal é `run()`, sendo invocado toda vez que o método `start` de uma *thread* for invocado. Neste método, é feita a chamada do método. Se a chamada for via *callback*, é preciso ainda invocar o método definido como *callback*.

```
public void run() {
   result = m.invoke(obj, args);
   promises.copyPromises (result);

   if (resultListener != null)
      /* callback */
      resultListener.resultHandler(result);
}
```

4.2 Conclusão

Procurou-se destacar neste capítulo alguns pontos importantes a respeito da implementação de FlexRMI. Dois pontos mereceram ser destacados: a utilização de um interceptor de chamadas representado pelo mediador e o uso de *threads* para implementação de chamadas assíncronas.

Capítulo 5

Avaliação e Comparação de Resultados

A fim de avaliar o desempenho da implementação de FlexRMI, foram realizados alguns experimentos. Os testes foram feitos em uma máquina Pentium IV 2.4 GHz com 256 Mb de memória RAM, executando o Windows XP. Além disso, não havia nenhuma outra aplicação executando simultaneamente no momento dos testes e os objetos cliente e servidor estavam executando na mesma máquina.

Os experimentos consistiram em realizar diversas chamadas de métodos remotos de modo síncrono e assíncrono. Dois parâmetros foram avaliados:

1. o tempo da tarefa remota;
2. o número de *threads* na chamada assíncrona.

Além destes dois parâmetros, os testes permitiram medir a sobrecarga introduzida pelo uso do FlexRMI, uma vez que todas as chamadas passaram a ser interceptadas pelo mediador. A aplicação utilizada para os testes permitia configurar tarefas remotas com diferentes tempos, simulados por meio de um laço `for`, bem como alterar o númeo de *threads* que executavam simultaneamente.

5.1 Variando o Tempo da Tarefa Remota

Nesta seção, estuda-se o comportamento da execução de FlexRMI variando o tempo da tarefa remota. Para cada valor do número de *threads*, 2, 4, 8, 16 ou 32, executando simultaneamente, variou-se o tempo da tarefa remota no intervalo de 0ms a 15000ms. Para melhor visualização dos resultados, os valores relativos ao tempo de execução, eixo y, foram plotados em escala logarítmica. Os resultados são apresentados nos gráficos a seguir.

Inicialmente, utilizou-se 2 *threads*, como mostrado na Figura 5.1. Como pode ser visto no gráfico, existem dois momentos distintos que devem ser destacados:

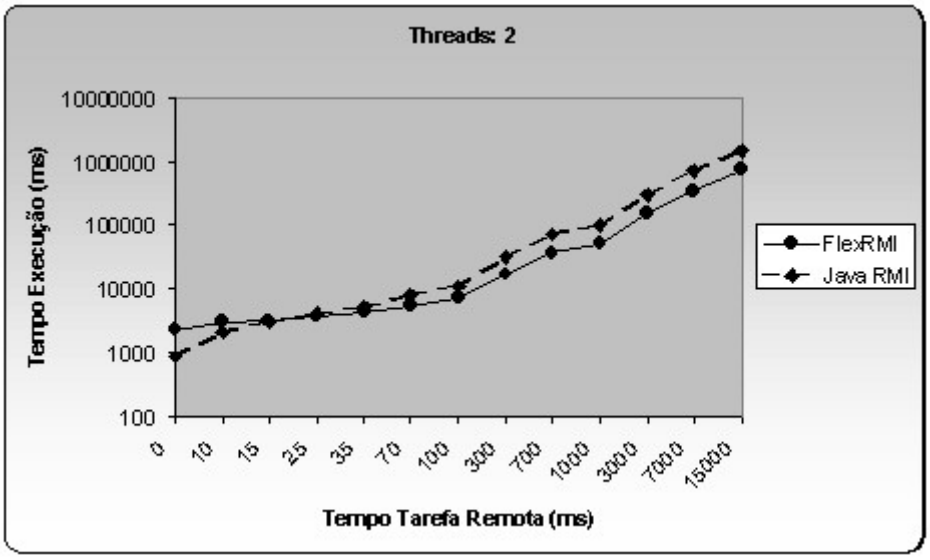

Figura 5.1: Gráfico - 2 *Threads*

- Inicialmente, Java RMI executa em um tempo menor do que FlexRMI.
- Após um determinado ponto, as curvas se cruzam e tem-se FlexRMI executando em menor tempo do que Java RMI. A partir daí, a tendência é que a configuração das curvas se mantenha desta maneira.

A Figura 5.2 ilustra o resultado para 4 *threads*. Pode-se notar que o comportamento das curvas se mantém inalterado, ou seja, existe um ponto no qual FlexRMI

se torna mais eficiente do que Java RMI. Entretanto, o ponto de cruzamento entre as curvas se deslocou para a esquerda, indicando que FlexRMI se tornou eficiente em tarefas com tempos de execução menores.

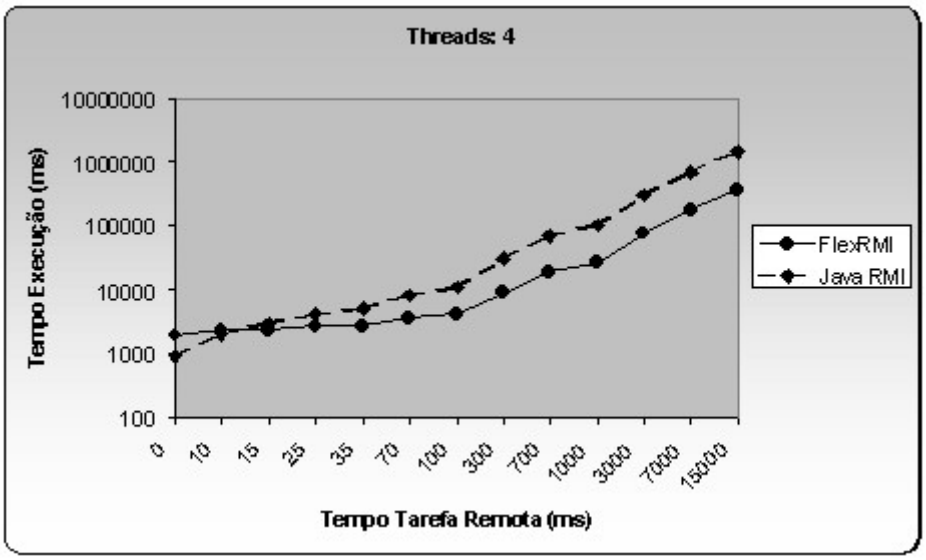

Figura 5.2: Gráfico - 4 *Threads*

Para os demais casos, 8, 16 e 32 *threads*, ilustrados nas Figuras 5.3 a 5.5, nota-se um comportamento semelhante, indicando que deve ser esta a configuração das curvas de Java RMI e FlexRMI, de acordo com a variação do número de *threads*.

Uma observação que pode ser feita, é que as curvas se distanciam mais uma da outra à medida que o número de *threads* cresce. Isto é bem natural, uma vez que quanto mais *threads*, maior será o assincronismo e mais eficiente será a execução.

5.2 Variando o Número de *Threads* Simultâneas

Nesta seção, estuda-se o comportamento da execução de FlexRMI variando o número de *threads* executando simultaneamente. Para cada valor do tempo da tarefa remota, 0, 10, 100 e 3000, variou-se o número de *threads* no intervalo de 2 a 32. Os resultados são apresentados nos gráficos a seguir.

Na Figura 5.6, pode-se observar o comportamento dos sistemas quando o *loop*

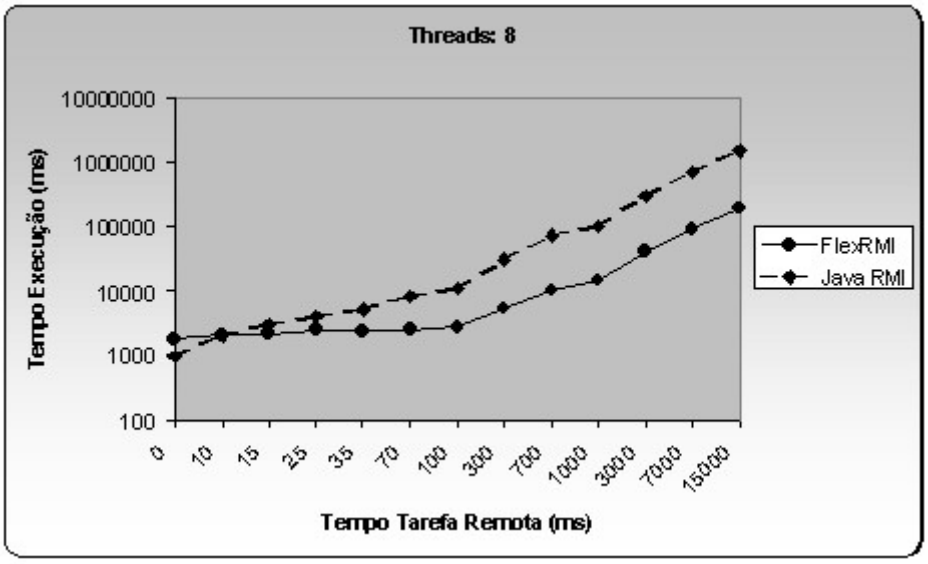

Figura 5.3: Gráfico - 8 *Threads*

do método não for executado. No caso de Java RMI, espera-se um comportamento constante, uma vez que este não é afetado pelo número de *threads*. Entretanto, este caso é interessante por mostrar a sobrecarga imposta pela criação e gerenciamento de *threads*, além da introdução da camada extra do mediador.

Como é de se esperar, Java RMI é mais eficiente do que FlexRMI neste caso, como ilustrado na Figura 5.6. Entretanto, pode-se notar na Figura 5.7 que aumentando o mínimo possível no tempo da tarefa remota, FlexRMI consegue ser mais eficiente a partir de um determinado valor.

Para outros valores de tempo da tarefa remota apresentados nas Figuras 5.8 e 5.9, FlexRMI é mais eficiente já para o número mínimo de *threads*. Pode-se notar pelo formato das curvas, o quanto FlexRMI é sensível à variação do número de *threads* simultâneas.

5.3 Discussão dos Resultados

Destaca-se que a aplicação permitiu a introdução de assincronismo em Java sem a necessidade de alterar a linguagem nem suas ferramentas. FlexRMI é compatível

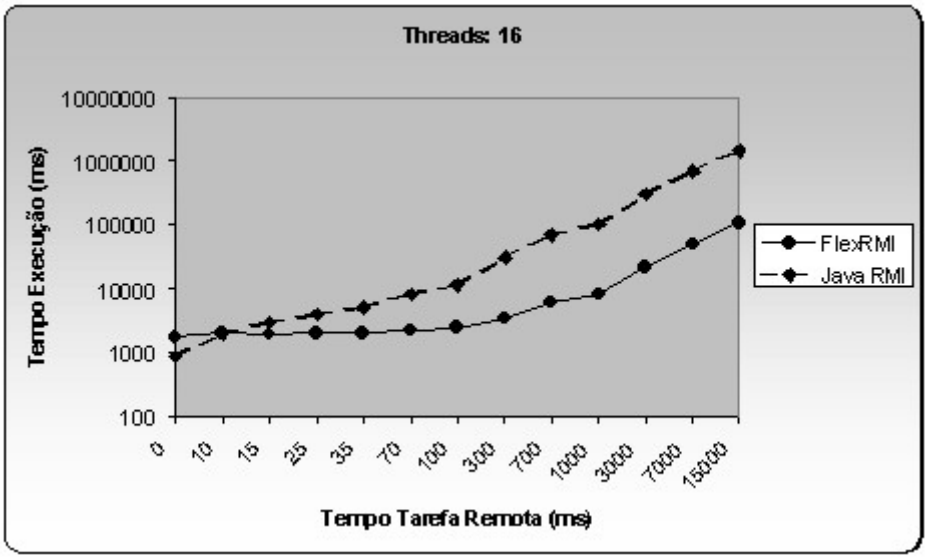

Figura 5.4: Gráfico - 16 *Threads*

com as versões anteriores de Java, permitindo, inclusive, chamadas assíncronas a métodos de objetos que foram projetados para executarem sincronamente. Além disso, a interface de nosso sistema é simples e de fácil uso, reduzindo a necessidade de tempo extra para aprendizagem.

Por outro lado, o uso de *proxy* implica em níveis de indireção das referências, podendo ocasionar uma certa degradação no desempenho do sistema, mas este custo é compensado pela aderência de FlexRMI à linguagem Java e ao seu ambiente de desenvolvimento.

Por ser dependente dos recursos de Java para reflexão computacional, FlexRMI não pode ser utilizado em ambientes que não forneçam tal propriedade. Sendo assim, seu uso fica restrito às versões J2SE e J2EE (*Java Enterprise Edition*), não podendo ser utilizado por J2ME (*Java Micro Edition*).

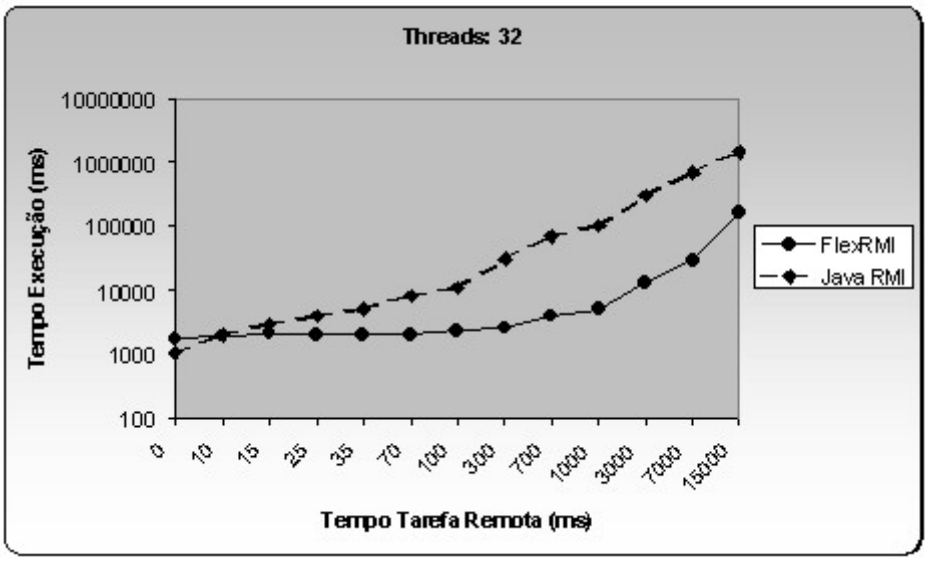

Figura 5.5: Gráfico - 32 *Threads*

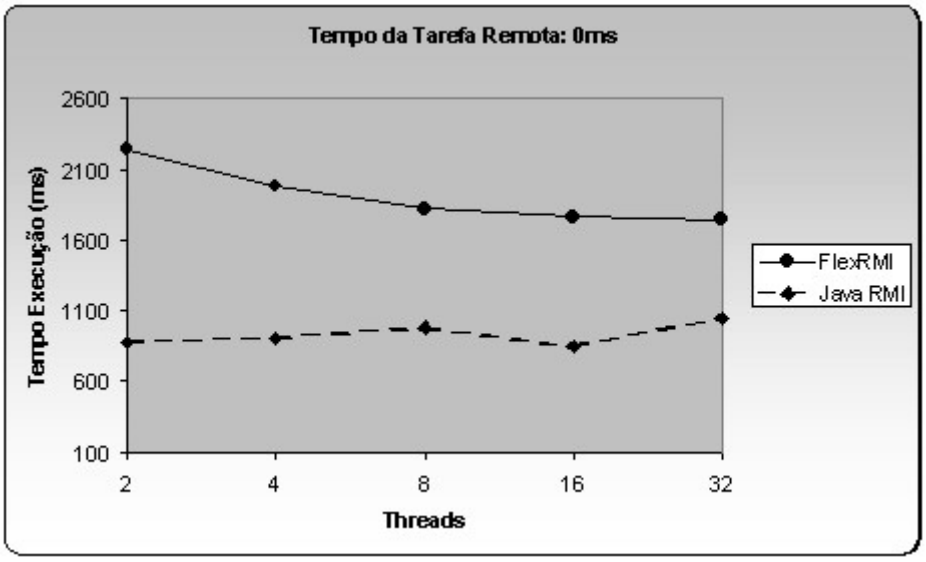

Figura 5.6: Gráfico - Tarefa Remota de 0ms

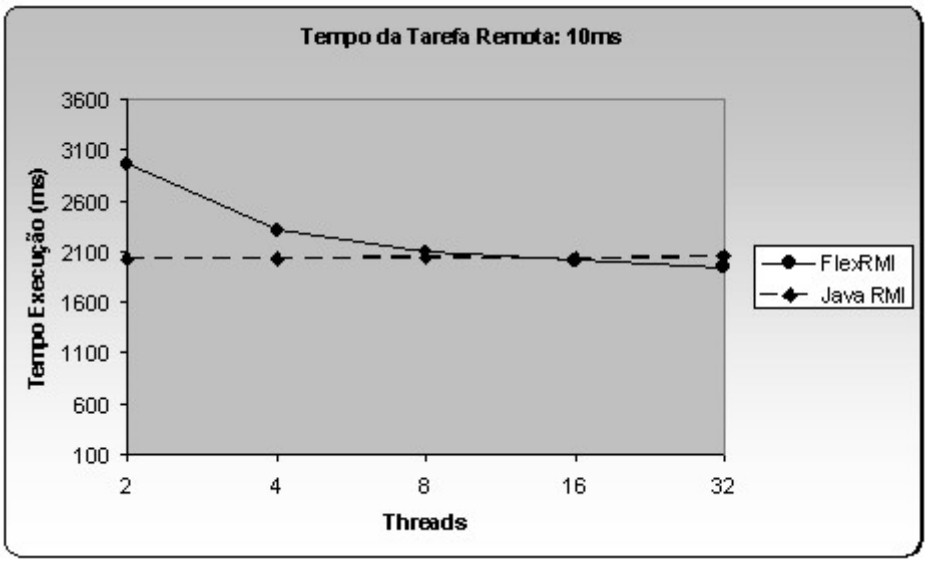

Figura 5.7: Gráfico - Tarefa Remota de 10ms

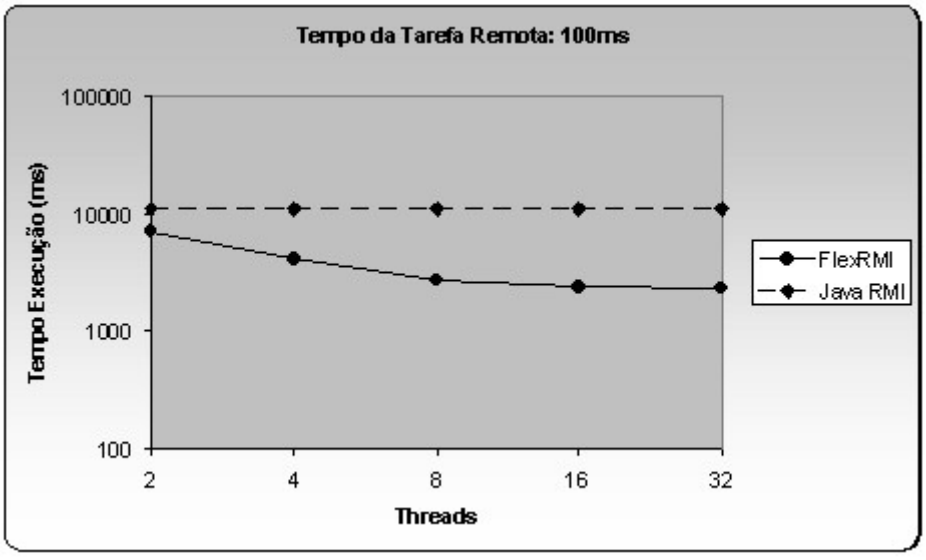

Figura 5.8: Gráfico - Tarefa Remota de 100ms

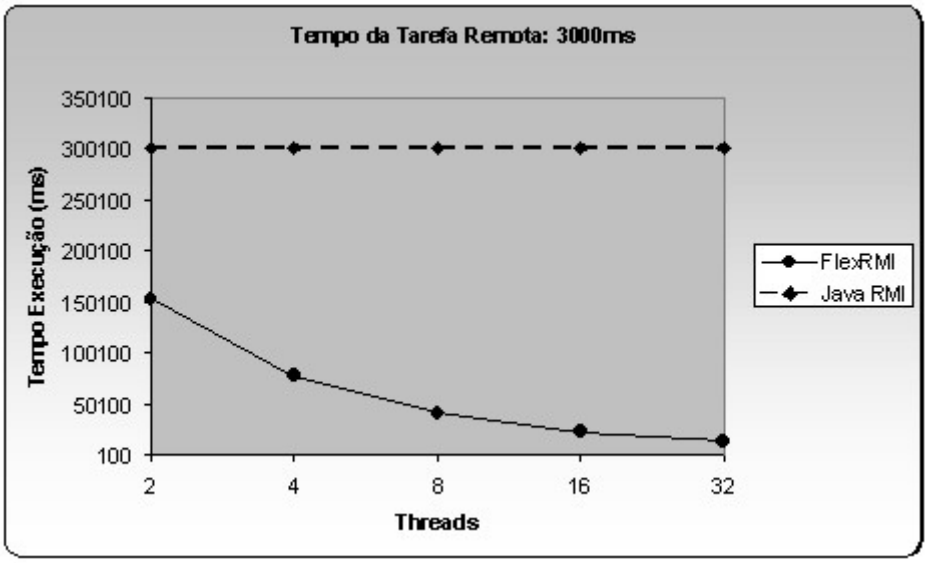

Figura 5.9: Gráfico - Tarefa Remota de 3000ms

Capítulo 6

Conclusões e Trabalhos Futuros

O estabelecimento de novos tipos de redes de computadores, tais como redes sem fio e redes de sensores, com características distintas daquelas apresentadas por redes locais levou à necessidade de adaptação do modelo de comunicação entre os computadores destas redes. Basicamente, o modelo de comunicação síncrona, utilizado pela maioria dos *middlewares*, não atende integralmente aos requisitos de redes não-locais. Nestes ambientes, propriedades como previsibilidade, latência com limite superior definido e largura de banda constante e confiável, não podem ser observadas. Desta forma, passou-se a incorporar mecanismos de comunicação assíncrona aos sistemas distribuídos a fim de atender aos requisitos próprios destas novas categorias de redes.

Em uma chamada assíncrona de métodos, o cliente que realizou a invocação não fica bloqueado à espera do resultado, ou seja, após feita a chamada, o cliente continua a execução de outras tarefas normalmente. Com isso, é possível sobrepor computações locais com computações remotas, resultando em uma otimização do tempo de execução de tais tarefas.

Embora já existam propostas que procuram introduzir assincronismo em sistemas de objetos distribuídos, como Java RMI, tais trabalhos apresentam características que tornam seu uso restrito e de certa forma inconveniente: há alteração na sintaxe da linguagem fonte e/ou modificações nas ferramentas de desenvolvimento. A Tabela 6.1 apresenta uma comparação destes trabalhos e de FlexRMI.

Comparando-se FlexRMI com CORBA, pode-se concluir que ambos os sistemas

	Extensões de RMI com Chamadas Assíncronas			
	Raje	Falkner	Karaorman	FlexRMI
Alterações em Java	Não	Sim	Não	Não
Alterações em RMI	Sim	Sim	Sim	Não
Resultados via *Polling*	Sim	Sim	Sim	Sim
Resultados via *Callback*	Não	Não	Não	Sim
Integração com sistemas RMI síncronos	Não	Não	Não	Sim

Tabela 6.1: Comparação entre Extensões Assíncronas de Java RMI.

	FlexRMI	CORBA
Polling	Sim	Sim
Callback	Sim	Sim
Integração com sistemas RMI síncronos	Sim	Sim

Tabela 6.2: Comparação entre FlexRMI e CORBA.

possuem funcionalidades semelhantes, como mostrado na Tabela 6.2.

6.1 Conclusões

Este trabalho reuniu um conjunto de fatores que justificam o desenvolvimento de sistemas de comunicação assíncrona, tendo como alvo Java RMI, o mecanismo de comunicação remota para aplicações desenvolvidas totalmente em Java.

Apesar de ser um modelo bem consolidado, o fato de não permitir a invocação assíncrona de métodos restringe a utilização da versão oficial de Java RMI em alguns casos. Deste modo, a extensão apresentada neste trabalho, FlexRMI, vem contribuir no sentido de superar esta deficiência. Além disso, a implementação apresentada é totalmente compatível com Java RMI, dando ao sistema uma face híbrida, uma vez que um mesmo método pode ser chamado sincronamente em um local e assincronamente em outro. Uma outra característica de FlexRMI é a possibilidade de realizar chamadas assíncronas em objetos já implementados e em produção, desenvolvidos unicamente para suportar chamadas síncronas. Destaca-

se ainda, como principal vantagem de FlexRMI, o fato do mesmo não ter alterado a sintaxe nem a semântica de Java RMI, fazendo com que a introdução de assincronismo seja natural, reduzindo o tempo extra para aprendizagem.

Baseado nos modelos de *polling* e *callback* para comunicação assíncrona, a ferramenta FlexRMI foi implementada de forma a oferecer aos desenvolvedores um ambiente simples no qual pudessem incorporar o assincronismo a suas aplicações distribuídas. No modelo de *polling*, o controle é repassado ao cliente logo após a chamada ter sido feita, sendo que o cliente pode inspecionar explicitamente o resultado da chamada em algum momento mais tarde. No modelo de *callback*, o controle também é passado ao cliente logo após a chamada ter sido feita, porém ao término da mesma, é invocado automaticamante um método no cliente por meio do qual o resultado é transferido do servidor ao cliente. Como no *callback* não é necessário ficar verificando a disponibilidade do resultado periodicamente, concluise que o modelo de eventos fornecido por ele é mais adequado para a maioria dos casos onde chamadas assíncronas devem ser realizadas.

Nos testes realizados, mostrou-se a viabilidade de se utilizar FlexRMI para introdução de assincronismo. Os resultados mostram que FlexRMI é mais eficiente do que Java RMI na maioria dos casos, mesmo que a utilização do primeiro venha a trazer uma sobrecarga devido à utilização de uma camada adicional ao sistema.

Finalmente, destacam-se dois aspectos importantes de FlexRMI:

- A utilização de conceitos como reflexão computacional e classes de *proxy* dinâmicas permitiu resolver um problema até então só possível por meio de alterações profundas nas ferramentas de desenvolvimento.

- A face híbrida do sistema permitindo chamar métodos de forma síncrona ou assíncrona, inclusive de objetos que originalmente só foram projetados para executar chamadas síncronas.

6.2 Trabalhos Futuros

No desenvolvimento deste trabalho, foram encontrados alguns aspectos que, se tratados, trariam grandes benefícios para o sistema FlexRMI. A seguir, é apresentada uma pequena descriçao de cada um destes aspectos:

- **Computação em Modo Desconectado** Uma das características das redes não-locais, como redes sem fio ou móveis, é a alta taxa de desconexões. Se em redes locais, desconexões são exceções, nestes novos tipos de redes, são eventos normais que não devem ser motivos para descontinuar a execução de uma computação. Sendo assim, seria interessante poder recuperar computações iniciadas mas que foram interrompidas pela ocorrência de uma desconexão, de acordo com alguns parâmetros fornecidos pelo programador.

- **Adaptação de FlexRMI ao J2ME** A tecnologia Java emergente para dispositivos limitados computacionalmente, tais como telefones celulares ou eletrodomésticos, é o J2ME.

  A plataforma Micro Edition, ou J2ME, veio suprir as necessidades de um mercado cada vez maior de dispositivos computacionais, que vão desde pequenos commodities, como *pagers*, até aparelhos televisores com acesso à Internet. Existem duas categorias principais de dispositivos que são atendidos pela plataforma J2ME. O critério para esta divisão é a capacidade computacional dos aparelhos.

  Tendo sido projetada para dispositivos limitados em termos de capacidade computacional, a versão J2ME possui diversas restrições que não estão presentes em outras versões da linguagem Java. Dentre estas limitações cita-se a não existência do tipo ponto flutuante, utilizado para representar números reais. Outra limitação importante é a ausência de suporte aos mecanismos de reflexão computacional presentes na linguagem Java, os quais permitem que informações sobre a estrutura interna de um programa, por exemplo, o tipo dinâmico de um objeto, sejam conhecidas enquanto o mesmo está sendo executado. Uma conseqüência desta última limitação é a impossibilidade de utilizar neste ambiente o pacote Java RMI, pois este faz uso de reflexão computacional para passar objetos serializados como parâmetros de invocações remotas ou como valores de retorno das mesmas [24].

  Portanto, por não possuir alguns recursos básicos para a utilização de FlexRMI, nosso sistema não é aplicável diretamente ao J2ME. Seria, portanto, importante um estudo no qual fosse possível migrar a atual implementação de FlexRMI para J2ME.

- **Estabelecimento das Condições de Uso de FlexRMI** Como mostrado nos testes realizados, existem alguns pontos de interesse onde FlexRMI passa a ser mais eficiente do que Java RMI. Sendo assim, a elaboração de um modelo analítico que permita identificar tais pontos é um trabalho interessante e de grande importância no uso de FlexRMI.

Bibliografia

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. Massachusetts Institute of Technology Press, Cambridge, 1984.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3nd edition, 2000.

[3] G. Booch, , J. Rumbaugh, and I. Jacobson. *The Complete UML Training Course*. Addison-Wesley, 2000.

[4] C. Bryce, C. Razafimahefa, and M. Pawlak. Lana: An approach to programming autonomous distributed systems. In *ECOOP 2002 - Object-Oriented Programming, 16th European Conference*, Malaga, Spain, June 2002.

[5] Luca Cardelli. Abstractions for mobile computation. *Secure Internet Programming*, pages 51–94, 1999.

[6] Microsoft Corporation. DCOM Technical Overview. http://msdn.microsoft.com/library/en-us/dndcom, 1996.

[7] Flaviu Cristian. Synchronous and asynchronous group communication. *Communications of the ACM*, 39(4):88–97, 1996.

[8] M. T. de O. Valente. *Mobilidade e Coordenação de Aplicações em Redes sem Fio*. PhD thesis, Universidade Federal de Minas Gerais, Belo Horizonte/MG, 2002.

[9] W. Emmerich. *Engineering Distributed Objects*. Wiley, 1st edition, 2001.

[10] K. E. Kerry Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. In *In Proceedings of Parallel and Real Time Systems (PART'99)*, Melbourne, AUS, July 1999.

[11] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transaction on Programming Languages and Systems*, 4(7):501–538, October 1985.

[12] S. Haridi, P. Van Roy, P. Brand, and C. Schulte. Programming Languages for Distributed Applications. *New Generation Computing*, 16(3):223–261, 1998.

[13] Tom Harpin. Using java.lang.reflect.proxy to interpose on java class methods. http://developer.java.sun.com/developer/technicalArticles/JavaLP/Interposing/, July 2001.

[14] J. Jaworski. *Java 2 Plataform - Unleashed*. Sams, 1st edition, 1999.

[15] M. Karaorman and J. Bruno. A-RMI: Active Remote Method Invocation System for Distributed Computing Using Active Java Objects. In *In Proceedings for the Technology of Object Oriented Languages and Systems (TOOLS-26)*, pages 1–13, Santa Barbara, California, Nevada USA, August 1998.

[16] B. Liskov. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the SIGPLAN '88*, pages 260–267, Atlanta, Georgia USA, June 1988.

[17] Glen McCluskey. Using java reflection. http://developer.java.sun.com/developer/technicalArticles/ALT/Reflection, January 1998.

[18] N. Benton e L. Cardelli e C. Fournet. Modern Concurrency Abstractions for C#. In Boris Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference*, Lecture Notes in Computer Science 2374, pages 415–440, Malaga, Spain, June 2002.

[19] Object Management Group. CORBA Messaging Joint Revised Submission. *Object Management Group, Framingham, document orbos/98-05-05*, 1998.

[20] OMG. The Common Object Request Broker: Architecture and Specification. *Object Management Group*, December 2001.

[21] R. R. Raje, J. I. William, and M. Boyles. An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java. In *On-line Proceedings of the ACM 1997 Workshop on Java for Science and Engineering Computation*, Las Vegas, Nevada USA, 1997.

[22] D. C. Schmidt and S. Vinoski. Programming Asynchronous Method Invocations with CORBA Messaging. *SIGS C++ Report Magazine*, pages 1–6, February 1999.

[23] A. Silberschatz and P.B. Galvin. *Operating System Concepts*. Addison-Wesley, 5th edition, 1998.

[24] Sun Microsystems. Sun Microsystems Java Remote Method Invocation Specification, December 2001.

[25] Inc. Sun Microsystems. Dynamic proxy classes. http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html, 1999.

[26] Inc. Sun Microsystems. Class proxy. http://java.sun.com/1.4/docs/api/java/lang/reflect/Proxy.html, 2002.

[27] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2001.

[28] W.F. Taveira, M.T. de O. Valente, M.A. da S. Bigonha, and R. da S. Bigonha. Chamada Assíncrona de Métodos Remotos em Java. In *VII Simpósio Brasileiro de Linguagens de Programação*, pages 78–91, Ouro Preto, Minas Gerais, Brasil, May 2003.

[29] D. Thompson, C. Exton, L. Garret, A.S.M. Sajeev, and D. Watkins. Distributed Component Object Model (DCOM). http://citeseer.nj.ncc.com/thompson97distributed.html, February 1997.

[30] Michael Thuan-Duc. Test Bed for Distributed Object Technologies using Java 490.45DT Project Report. citeseer.nj.ncc.com/511765.html, November 1998.

[31] Steve Vinoski. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), 1997.

[32] Steve Vinoski. New features for CORBA 3.0. *Communications of the ACM*, 41(10):44–52, 1998.

[33] J. Waldo. *The Jini Specifications*. Addison-Wesley, 2nd edition, 2001.

[34] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical report, Sun Microsystems Laboratories, 1994.

[35] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290, 1996.