Chamada Assíncrona de Métodos Remotos em Java

Wendell Figueiredo Taveira[1] **, Marco Túlio de Oliveira Valente**[2] **,**
Mariza Andrade da Silva Bigonha[1] **,**
Roberto da Silva Bigonha[1]

[1]Departamento de Ciência da Computação – Instituto de Ciências Exatas
Universidade Federal de Minas Gerais
Caixa Postal 702 – 30123-970 – Belo Horizonte, MG

[2]Instituto de Informática – PUC Minas
Av. Dom José Gaspar, 500 – Coração Eucarístico
Prédio 34 – CEP 30535-610 – Belo Horizonte, MG

{taveira,mariza,bigonha}@dcc.ufmg.br, mtov@pucminas.br

Abstract. *Java RMI is the 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 paper it is presented Java ARMI, a Java based system that supports asynchronous invocation of remote methods. The system is completely implemented in Java, making use of the reflection and dynamic proxy facilities of this language. The implementation is also compatible with standard Java RMI distributed systems.*

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 o Java ARMI, uma ferramenta totalmente compatível com a linguagem Java que permite invocar métodos remotos assincronamente. A ferramenta foi implementada utilizando os recursos de reflexão computacional e de proxy dinâmico fornecidos por Java, tornando a solução simples e engenhosa.*

1. Introdução

A noção de sistemas distribuídos tem se firmado cada vez mais como o padrão de plataforma de desenvolvimento de *software*. Em contraste com sistemas centralizados, sistemas distribuídos são constituídos por um conjunto de processadores autônomos, interligados por uma rede. Cada um destes processadores é denominado de nó ou nodo. A Internet caracteriza bem este novo tipo de ambiente, onde processadores heterogêneos, provavelmente situados em locais distintos, formam uma rede de enorme proporção [Haridi et al., 1998].

Este artigo foi desenvolvido como parte de um projeto de pesquisa apoiado pela FAPEMIG (processo CEX 488/2002).

Sistemas de objetos distribuídos, tais como CORBA [OMG, 2001] e Java RMI (*Remote Method Invocation*) [Arnold and Gosling, 1997, Sun Microsystems, 2001], tradicionalmente fornecem apenas um mecanismo de comunicação entre os nodos, a saber, a comunicação síncrona. Chamadas síncronas são adequadas para redes locais onde se pode destacar 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 fio. 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 estes, pode ser interessante sobrepor computações para fins de otimizações, por meio de chamadas assíncronas [Vinoski, 1997].

Existe uma tendência atual em se incorporar mecanismos de comunicação assíncrona em sistemas distribuídos. Prova disto, é que a última versão de CORBA passou a incorporar chamadas assíncronas. 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 disso, Java RMI consiste na base de novas tecnologias tais como Jini [Waldo, 2001], um *middleware* adequado para redes reconfiguráveis.

O objetivo deste trabalho é, portanto, investigar o desenvolvimento de sistemas de objetos distribuídos que permitam chamadas assíncronas de métodos remotos, quando for desejado ou necessário. Sendo assim, propomos e implementamos uma extensão de Java RMI com chamadas assíncronas, denominada de Java ARMI. Nosso sistema foi implementado tendo como base Java RMI e sempre tivemos a preocupação de mantê-lo totalmente compatível com o mesmo.

É importante ressaltar que Java ARMI corresponde a um modelo híbrido no qual é possível invocar métodos remotos de forma síncrona ou assíncrona. Nada impede, por exemplo, que um método seja chamado sincronamente em algum ponto do programa e assincronamente em outro. Cabe ao programador a decisão de escolher como a chamada será realizada.

O trabalho desenvolvido se diferencia dos demais sistemas de chamadas assíncronas propostos na literatura [Raje et al., 1997, Karaorman and Bruno, 1998, Falkner et al., 1999] em dois pontos principais:

1. utilização de mecanismos de reflexão fornecidos pela linguagem Java;

2. consideração de diferentes modelos de chamadas assíncronas propostos na literatura.

O restante do artigo está organizado da seguinte maneira: a Seção 2 faz uma breve revisão da literatura, apresentando o estado da arte em relação aos ambientes que disponibilizam chamadas assíncronas de métodos remotos. A Seção 3 descreve o modelo RMI de Java. A Seção 4 apresenta um pequeno roteiro de como implementar programas utilizando Java ARMI. A Seção 5 expõe detalhes importantes utilizados na implementação do Java ARMI. A Seção 6 apresenta uma avaliação do Java ARMI. E finalmente, a Seção 7 apresenta as conclusões e trabalhos futuros.

2. Revisão da Literatura

A construção de ambientes que permitem chamadas assíncronas de métodos remotos já foi assunto de várias pesquisas presentes na literatura.

Em Multilisp [Halstead, 1985], é mostrado como incorporar paralelismo a programas, introduzindo a construção `future`, que fornece a idéia de *contratos futuros* ou de *promessa de entrega*. Deste modo, é modelada a forma de concorrência entre a computação de um valor e a disponibilidade ou uso do mesmo.

Em [Liskov, 1988], é mostrado o projeto da construção `promises` que foi influenciada pelo mecanismo da construção `future` de Multilisp. Como em `futures`, `promises` permite que o resultado de uma chamada seja demandado e utilizado somente em algum ponto mais adiante no programa. Ademais, `promises` são fortemente tipadas, eliminando a necessidade de verificação de tipos em tempo de execução. Permitem também a propagação de exceções de maneira mais conveniente.

Em [Raje et al., 1997], é apresentado um sistema construído tendo como base RMI que permite execução concorrente de computações locais e remotas. O sistema utiliza o conceito de *future*, ou seja, quando o cliente chama um método remoto assincronamente, o método retorna uma instância da classe *Future*, criada por um objeto *stub* especial. Entre outras funções, um objeto *future* é usado para que o servidor envie exceções para os clientes.

Em [Karaorman and Bruno, 1998], um programa A-RMI (*Active Remote Method Invocation*) pode referenciar ou invocar métodos de objetos remotos de forma síncrona ou assíncrona, usando a mesma sintaxe como se os objetos fossem locais. Três abstrações importantes são fornecidas: objetos ativos remotos com escalonamento feito pelo usuário; invocação assíncrona de métodos *data-driven* e sincronização não bloqueante por meio do uso de manipuladores de chamadas; e criação transparente de objetos remotos.

Em [Falkner et al., 1999], é apresentada uma extensão da implementação padrão de Java RMI por meio de um novo mecanismo para gerar *stubs*, incluindo protocolos de comunicação adicionais. Para indicar quais os métodos utilizarão comunicação assíncrona, são usados objetos *Future* definindo uma pseudo-interface Java com as palavras-chaves adicionais `async` e `future`. O mecanismo de *stub* estendido proposto produz uma melhora significativa no desempenho, removendo atrasos desnecessários causados pelo bloqueio nas invocações síncronas de Java RMI.

Todos os trabalhos mencionados acima propõem um mecanismo de chamada assíncrona de métodos remotos e apresentam propostas de melhora de desempenho, sobretudo em situações de sincronização.

Além de Java RMI, outros trabalhos propõem solução para o problema de comunicação assíncrona em outras tecnologias. Em [Schmidt and Vinoski, 1999], é apresentado um trabalho cujo modelo utilizado é o de CORBA. O trabalho mostra como implementar programas C++ que utilizam os modelos de *polling* e *callback* para invocação assíncrona de métodos remotos.

Todos os trabalhos mencionados requerem alterações na linguagem Java ou no compilador que gera os *stubs* e *skeletons*. O estudo que apresentamos nas Seções 4, 5 e 6 consiste em uma solução para contornar estes problemas.

3. Java RMI

Java RMI [Sun Microsystems, 2001, Wollrath et al., 1996, Baclawski, 1998, Thuan-Duc, 1998] é um modelo de objetos distribuídos para a plataforma Java. Especificamente, Java RMI permite aos desenvolvedores manipular objetos remotos muito similarmente ao modo como são manipulados objetos locais, 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 `RemoteExceptions` quando um método é chamado em um objeto remoto.

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 servidor. Objetos remotos são descritos por uma ou mais interfaces. 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 1 mostra o relacionamento entre estas interfaces e classes.

O 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 [Thuan-Duc, 1998, Wollrath et al., 1996]. 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*.

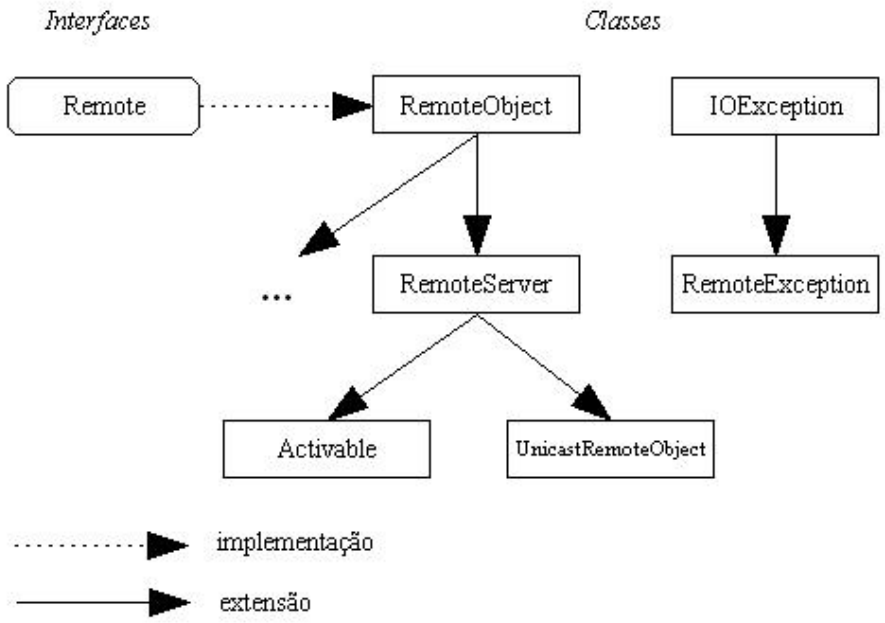

Figura 1: Relacionamento entre Interfaces e Classes em Java RMI

- 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.

3.1. Arquitetura do Sistema Java RMI

No modelo de objetos distribuídos de Java [Wollrath et al., 1996, Jaworski, 1999], um objeto cliente nunca referencia um objeto remoto diretamente. Ao invés disso, ele referencia uma interface remota que é implementada pelo objeto remoto. O uso de interfaces remotas permite aos objetos servidores diferenciar entre suas interfaces locais e remotas. Por exemplo, um objeto poderia prover métodos a objetos que executam na mesma JVM, além de métodos que seriam providos via interface remota. O uso de interfaces remotas também permite aos objetos servidores apresentarem diferentes modos de acessos remo-

tos. Por exemplo, um objeto servidor pode prover interfaces diferentes para diferentes grupos de usuários. Por último, o uso de interfaces remotas permite que a posição do objeto servidor dentro de sua hierarquia seja abstraída. Com isso, objetos clientes podem ser compilados usando somente a interface remota, sem a necessidade dos arquivos das classes do servidor.

4. Descrição do Java ARMI

Esta seção apresenta um pequeno roteiro de como utilizar o sistema Java ARMI (*Asynchronous Remote Method Invocation*). O Java ARMI é um mecanismo que permite invocar métodos de objetos que existem em um outro espaço de endereçamento, que podem estar ou não na mesma máquina. A principal vantagem do Java ARMI sobre o Java RMI é a possibilidade de poder invocar métodos que executarão de modo assíncrono.

4.1. Utilização do Java ARMI

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 processo 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 Java ARMI:

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 por dois modos diferentes:
   1.1. No espaço de endereçamento onde o objeto foi criado, pode-se usá-lo como qualquer outro objeto.
   1.2. Em outros espaços de endereçamento, o objeto pode ser referenciado usando um manipulador de objetos[2]. 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` serão apresentados na Seção 4.1.1. Apesar de haver algumas restrições no modo de utilização de um manipulador de objetos, na maioria dos casos, o objeto pode ser usado como se fosse um objeto normal. Se um objeto remoto é passado como parâmetro ou valor de retorno, então o manipulador de objetos é copiado de um espaço de endereçamento para outro.

[2]Do inglês, *object handler*.

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**[3]. Se um objeto serializável é passado como parâmetro ou valor de retorno em uma chamada a um método remoto, então o valor do objeto é copiado de um espaço de endereçamento para outro.

4.1.1. O Pacote Java ARMI

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

A classe `ARMI` possui os seguintes métodos:

- `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 valor do manipulador de objetos estiver disponível. Retorna `false`, caso contrário.
- `public Object getResult()`: retorna o valor do manipulador de objetos. 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`.

4.2. 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.

[3]Do inglês, *serializable object*.

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 4.2.1, 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 manipulador de objetos, enquanto a classe remota representa o tipo de um objeto. 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.

4.2.1. Classes Serializáveis

Uma classe é serializável se ela implementa a interface `java.io.Serializable`. Subclasses de uma classe serializável também são classes serializáveis. Detalhes sobre estas classes podem ser encontrados em [Arnold and Gosling, 1997].

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.

4.2.2. Programando um Servidor

Uma vez discutido como definir as classes remota e serializável, será discutido como programar o cliente e o servidor. No programa de exemplo mostrado a seguir, têm-se uma classe remota e sua correspondente interface remota que serão, respectivamente, chamadas de `Alo` e `AloInterface`. A seguir, mostra-se o arquivo `AloInterface.java`:

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

O arquivo `Alo.java` é exibido abaixo:

```
01 import java.rmi.*;
02 import java.rmi.server.*;
03 public class Alo extends UnicastRemoteObject
                    implements AloInterface {
04    private String mensagem;
05    public Alo (String msg) throws RemoteException {
06       mensagem = msg;
07    }
```

```
08    public Promises say() throws RemoteException {
09       Promises h = new Promises();
10       h.setResult(message);
11       return h;
12    }
13 }
```

É importante notar que o método `say()` será 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`.

Neste ponto, é necessário salientar a importância do compilador `armic`. Suponhamos que existam métodos no servidor cujos tipos de retorno sejam diferentes de `Promises`. Até agora, não há como invocar estes métodos assincronamente. No entanto, é por meio do compilador `armic` que tal limitação é superada, fazendo com que a escolha entre a chamada síncrona ou assíncrona passe a ser função de quem solicita o serviço, no caso o cliente, e não mais de quem implementou o serviço. Para tanto, basta introduzir o prefixo `async_` ao nome do método que se queira invocar remotamente.

Do lado do servidor, também precisa existir uma classe que irá 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[] argv) {
05       try {
06          Naming.rebind ("Alo", new Alo ("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[4] 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 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.

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

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 `armic`, disponível no pacote Java ARMI. Detalhes sobre o `armic` podem ser encontrados na Seção 5.4. 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`.

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

O método `ARMI.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 `ARMI.lookup` deve ser moldado[5] para o tipo da interface remota.

A chamada de método remoto neste exemplo de cliente é `alo.say()`, retornando um objeto do tipo `Promises`, que é, por sua vez, serializável. Caso o método

[5] Do inglês, *cast*.

`say()` fosse síncrono, uma chamada como aquela feita na linha 5 retornaria o valor de retorno do próprio método. No entanto, `say()` é um método assíncrono e 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.

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.

4.2.4. 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`.

4.2.5. 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, AloInterface, AloInterface_ARMI (gerada pelo compilador `armic`) e Alo_Stub (gerada pelo compilador rmic) estejam disponíveis na máquina cliente. Em particular, os arquivos `AloCliente.class`, `AloInterface.class`, `AloInterface_ARMI.class` e `Alo_Stub.class` devem estar em algum diretório especificado no CLASSPATH.

5. Implementação de Java ARMI

A implementação de Java ARMI foi feita tendo como base toda a estrutura já disponibilizada pelo Java RMI padrão. Os protocolos de comunicação, que em última instância 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 e a invocação remota [Falkner et al., 1999], são realizadas do mesmo modo como em Java RMI.

Dois conceitos principais formam o núcleo do nosso sistema. São eles:

- reflexão;
- *threads*.

Uma chamada remota assíncrona é realizada disparando, em uma *thread* que executa em paralelo com a *thread* principal, a computação que deve ser realizada. Identificar se a comunicação deve ocorrer síncrona ou assincronamente, é uma questão de diferenciar o tipo de retorno da chamada realizada. Em Java ARMI, chamadas assíncronas são identificadas pelo tipo de retorno, a saber `Promises`[6]. Deste modo, toda vez que

[6] Aqui se incluem os métodos definidos originalmente para retornar `Promises`, bem como as interfaces geradas dinamicamente pelo `armic`, ou seja, com o nome dos métodos perfixado por `async_`.

uma chamada é realizada, é possível identificar o tipo de retorno por meio da reflexão fornecida por Java. Resumindo, a chamada assíncrona é realizada quando o tipo de retorno `Promises` é identificado, criando, para isso, uma nova *thread* a partir da *thread* em que foi feita a invocação.

A seguir, discutimos alguns detalhes de implementação do Java ARMI.

5.1. A Classe `Promises`

A classe `Promises` representa a estrutura utilizada para armazenar o valor de retorno de chamadas remotas assíncronas. 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. Para este caso, temos um *flag* que indica a ocorrência ou não de exceções. Em caso positivo, o objeto armazenará informações sobre a exceção e, não mais, o valor de retorno da chamada.

5.2. A Classe `Mediator`

A classe `Mediator` é responsável por implementar um objeto da classe *proxy* dinâmica utilizado para interceptar chamadas e distingui-las entre síncronas ou 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 através de uma interface uniforme.

Como dito anteriormente, as chamadas são diferenciadas pelo tipo de retorno. Quando o tipo de retorno `Promises` é identificado, é tarefa do mediador construir uma nova *thread* e dispará-la para execução.

É importante ressaltar que o objeto mediador precisa ser criado para poder capturar as chamadas feitas. Em nossa implementação, no momento em que o cliente obtém a referência a um objeto remoto, automaticamante é criado um mediador associado a este objeto remoto. Para tanto, o método `lookup` da classe `Naming`, foi encapsulado em um novo método `lookup` presente na classe `ARMI`.

5.3. Escalonador de Execução

A criação de *threads* de forma não controlada poderia se tornar um gargalo em nosso sistema, afetando significamente o desempenho de Java ARMI. Pensando nisso, foi implementado um escalonador de *threads* responsável por gerenciar os processos de criação, gerenciamento e eliminação de *threads*.

O escalonador executa utilizando listas de *threads*. Cada cliente possui uma lista na qual todas as chamadas serão enfileiradas. Tais listas possuem as informações necessárias para a execução das *threads* e são constantemente inspecionadas de modo a garantir a execução simultânea de um número máximo de *threads*[7].

[7]Este valor é um parâmetro do sistema. Nesta implementação o valor padrão é 8.

5.4. O Compilador armic

O compilador `armic` surgiu da necessidade de clientes poderem invocar assincronamente métodos existentes no servidor que haviam sido projetados para executar sincronamente. Em outras palavras, estes métodos não retornam o tipo `Promises`.

Nosso sistema contorna tal limitação por meio do compilador `armic`. Este compilador acessa o código compilado dos objetos remotos e cria uma nova interface contendo uma versão assíncrona de todos os métodos já existentes.

Para implementar o compilador `armic`, utilizamos o mecanismo de reflexão fornecido por Java. A reflexão permite que um programa Java seja examinado e manipulado internamente por ele mesmo ou por um outro programa. Sendo assim, foi possível descobrir qual o tipo de retorno dos métodos, bem como examinar seus nomes e identificar se havia ou não o prefixo `async_`.

É função do mediador, identificar as chamadas de métodos presentes nesta interface. Como não há uma implementação de tais métodos, é preciso que as versões síncronas dos mesmos sejam disparadas em outras *threads*, simulando assim a comunicação assíncrona.

6. Avaliação

Destacamos que nossa aplicação permitiu a introdução de assincronismo em Java sem a necessidade de alterar a linguagem nem suas ferramentas. Java ARMI é compatível 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, eliminando 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 Java ARMI à linguagem Java e ao seu ambiente de desenvolvimento.

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

7. Conclusão e Trabalhos Futuros

Java RMI é o modelo de computação distribuída utilizado por aplicações desenvolvidas totalmente em Java. Apesar de ser um modelo bem consolidado, o fato de não permitir a invocação assíncrona de métodos restringe a sua utilização em alguns casos. Deste modo, a nossa extensão, Java ARMI, vem contribuir no sentido de superar esta deficiência apresentada por Java RMI.

Nossa implementação pode ser considerada totalmente compatível com Java RMI. Além disso, fornecemos meios para que serviços já existentes também possam ser utilizados de modo assíncrono.

O pacote Java ARMI já está totalmente operacional. No entanto, algumas extensões podem ser implementadas tornando-o mais eficiente e funcional. Deste modo, dois pontos de grande importância são considerados como trabalhos futuros: a implementação do mecanismo de *callback* e um tratamento específico de exceções.

Referências

Arnold, K. and Gosling, J. (1997). *The Java Programming Language*. Addison-Wesley, 2nd edition.

Baclawski, K. (1998). Java RMI Tutorial. http://www.ccs.neu.edu.

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

Halstead, R. (1985). Multilisp: A language for concurrent symbolic computation. *ACM Transaction on Programming Languages and Systems*, 4(7):1–1.

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

Jaworski, J. (1999). *Java 2 Plataform - Unleashed*. Sams, 1st edition.

Karaorman, M. and Bruno, J. (1998). 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)*, Santa Barbara, California, Nevada USA.

Liskov, B. (1988). Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the SIGPLAN '88*, Atlanta, Georgia USA.

OMG (2001). The Common Object Request Broker: Architecture and Specification. *Object Management Group*.

Raje, R. R., William, J. I., and Boyles, M. (1997). 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.

Schmidt, D. C. and Vinoski, S. (1999). Programming Asynchronous Method Invocations with CORBA Messaging. *SIGS C++ Report Magazine*.

Sun Microsystems (2001). Sun Microsystems Java Remote Method Invocation Specification.

Thuan-Duc, M. (1998). Test Bed for Distributed Object Technologies using Java 490.45DT Project Report. citeseer.nj.ncc.com/511765.html.

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

Waldo, J. (2001). *The Jini Specifications*. Addison-Wesley, 2nd edition.

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