UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Arthur Cattaneo Zavadski

UTILIZANDO REFLEXÃO COMPUTACIONAL NO DESENVOLVIMENTO DE APLICAÇÕES DISTRIBUÍDAS

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Orientador:

Prof. Dr. Murilo Silva De Camargo

Florianópolis, junho de 2003.

UTILIZANDO REFLEXÃO COMPUTACIONAL NO DESENVOLVIMENTO DE APLICAÇÕES DISTRIBUÍDAS

Arthur Cattaneo Zavadski

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação com área de concentração em Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Prof. Dr. Fernando Álvaro Ostuni Gauthier
Coordenador do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina

Banca Examinadora:

_____

Prof. Dr. Murilo Silva De Camargo
Orientador e Presidente da Banca
UFSC

_____

Prof. Dr. Rosvelter João Coelho da Costa
UFSC

_____

Prof. Dr. Vitório Bruno Mazzola
UFSC

"The show must go on! Yeah!
The show must go on!
I'll face it with a grin!
I'm never giving in!
On with the show!"

Queen/Roger Taylor

A todos que tiveram a paciência e o companheirismo
necessários para me acompanhar durante a longa
elaboração deste trabalho.

Agradecimentos

À Univel.

À Universidade Federal de Santa Catarina.

Ao Prof. Dr. Murilo Silva de Camargo pela orientação deste trabalho e pelo direcionamento durante minhas diversas “viagens”.

À Datacoper Software, especialmente aos Srs. Cézar e Valdir, que acreditaram neste investimento e permitiram minha ausência para a realização das disciplinas.

Ao amigo Agnaldo, pelo valioso apoio durante a etapa mais exaustiva deste trabalho.

À minha avó Aparecida, minhas tias Maria e Bernadete e meu tio Pedro pelo apoio e constante incentivo à educação.

À minha irmã (Dra.) Luciana por me contagiar com sua energia, vibração e alegria.

À minha mãe Eliza por me inspirar com seu amor, compreensão e com seu brilhante exemplo de vida.

À minha querida Elaine, por tornar minha vida mais radiante a cada dia com amor, paixão e inteligência.

A Deus, por tudo isso!

Sumário

Lista de Figuras

RESUMO

O desenvolvimento de aplicações distribuídas exige o uso de algum mecanismo que possibilite a comunicação entre os processos cliente e servidor. Tecnologias de distribuição como RMI, CORBA e XML-RPC/SOAP disponibilizam serviços que facilitam esta atividade. Todavia, a implementação baseada nas interfaces de programação destas especificações acaba por misturar o código necessário à distribuição com a funcionalidade da aplicação, tornando-a dependente do mecanismo escolhido. Minimizar o impacto diante de uma troca de tecnologia e, simultaneamente, permitir que o desenvolvedor do software esteja focado em requisitos funcionais, constitui o principal problema desta pesquisa. De forma a possibilitar o desenvolvimento de aplicações distribuídas independentemente da plataforma de distribuição, esta dissertação propõe um framework que isola as especificidades de cada padrão dos componentes funcionais. Técnicas de reflexão computacional são aplicadas na implementação, de maneira a eliminar a codificação de adaptadores e proxies exigida em outras abordagens. Ao separar os elementos necessários à distribuição dos elementos funcionais, o framework possibilita a substituição dos mecanismos através de uma configuração externa ao software, dispensando, desta maneira, alterações no código fonte.

Palavras-chave: aplicações distribuídas, reflexão, proxy, frameworks, padrões de projeto.

ABSTRACT

Development of distributed applications requires the use of some mechanism that enables the communication between the server and client processes. Technologies for distribution, such as RMI, CORBA and XML-RPC/SOAP make available services that facilitate this activity. However, the implementation based on the programming interfaces of these specifications leads to mix the needed distribution code with the application functionality, making it dependent of the mechanism used. Minimizing the impact face to a technology change and, simultaneously, allowing the software developer to be focused in functional requirements constitutes the main problem of this research. In order to make possible the development of distributed applications independently of the distribution platform, this dissertation propose a framework that isolates the specificities of each standard from the functional components. Computational reflection techniques are applied in the implementation, in way to eliminate the codification of adapters and proxies required in other approaches. When separating the necessary elements to the distribution from the functional elements, the framework makes possible the substitution of the mechanisms through a configuration external to the software, avoiding, this way, changes in the code source.

Keywords: distributed applications, reflection, proxy, frameworks, design patterns.

Introdução

O crescente interesse na construção de aplicações baseadas em componentes distribuídos levou ao surgimento de diversos padrões e produtos para sua implementação. Plataformas como Java 2 *Enterprise Edition* (J2EE) da Sun Microsystems, *Common Object Request Broquer* (CORBA) da OMG e .NET da Microsoft disponibilizam a infraestrutura necessária para que artefatos de software possam ser implementados contando com serviços previamente elaborados, como serviços de nome, diretório, transações, mensagens e, principalmente, distribuição. O objetivo desta infraestrutura é possibilitar que os desenvolvedores de aplicações se concentrem na criação de componentes que representam o domínio da aplicação, deixando a cargo dos servidores de aplicação o fornecimento dos serviços essencialmente não-funcionais.

Este cenário é, claramente, uma evolução em relação às primeiras iniciativas de desenvolvimento de aplicações distribuídas, nas quais mecanismos menos abstratos eram utilizados para realizar a comunicação entre os diversos processos que constituem uma aplicação. As alternativas resumiam-se ao o suporte oferecido pelos sistemas operacionais à comunicação inter-processo (*inter-process communications*-IPC), como arquivos, *pipes*, *sockets*, semáforos e compartilhamento de memória e, com exceção dos *sockets*, caracterizavam-se como alternativas vinculadas à determinada plataforma (SZYPERSKI, 1999).

Todavia a disponibilidade de mecanismos de distribuição mais sofisticados não elimina por completo a complexidade no desenvolvimento de aplicações desta natureza, uma vez que cada especificação impõe um modelo específico para a codificação da aplicação, conforme determinado por suas interfaces de programação (APIs). Diante disso os arquitetos de sistemas são forçados a decidir-se pelo mecanismo de distribuição em estágios iniciais dos projetos, com base em requisitos não-funcionais determinados em situações, por vezes, não adequadas.

Complementarmente, a utilização direta das APIs de determinada especificação traz um grande prejuízo à flexibilidade do aplicativo desenvolvido, dado que o código utilizado para realizar a distribuição é um exemplo de *crosscutting concern*, ou seja, de instruções que encontram-se intercaladas com os elementos funcionais da aplicação sem

trazer novas funcionalidades dirigidas a seus usuários (MILI; MCHEICK; SADOU, 2002).

A combinação destes elementos forma um contexto especialmente delicado diante de uma flagrante realidade: a constante mudança de requisitos e tecnologias. Frente a tais mudanças algumas questões podem ser formuladas: Como determinado *software* poderia se valer de um mecanismo de distribuição que atendesse mais adequadamente às necessidades de seus usuários, sabendo que sua arquitetura está intimamente ligada aos modelos de programação impostos pela plataforma de distribuição? Como viabilizar uma alteração do modelo de distribuição, conhecendo a amplitude das alterações necessárias ao sistema previamente implementado?

A resposta a estas questões pode ser encontrada em diversos estudos cujas abordagens são baseadas em implementações específicas (GOMAA; MENASCÉ, 2000; PRYCE, 2000; DASHOFY et al., 1999; SILVA et al, 1997; ALVES; BORBA, 2001; RINE et al, 1999) ou através de definições de conectores arquiteturais (SHAW et al., 1996; MEHTA et al., 2000). A abordagem mais comumente encontrada baseia-se na definição de uma camada de abstração que encapsula as instruções relativas à distribuição, através de implementações dos padrões de projeto *proxy* e *adapter* (GAMMA et al., 1995). Embora atinjam seu objetivo, a implementação tradicional de tais padrões implica na geração de, ao menos, um par de elementos para atender cada lado da comunicação, cliente e servidor, para todo objeto servidor disponível remotamente aumentando, conseqüentemente, o volume de código a ser escrito.

Como forma de obter os benefícios das abordagens citadas sem incorrer no excesso de codificação de programas cujo propósito seja somente encapsular a distribuição, optou-se pelo uso das técnicas de reflexão computacional para estabelecer um procedimento suficientemente genérico a partir da representação reflexiva dos elementos envolvidos da comunicação.

Neste contexto, esta dissertação propõe um *framework* baseado em mecanismos reflexivos projetado de maneira a esconder os detalhes de implementação e utilização de padrões de distribuição específicos. O principal objetivo é proporcionar um mecanismo através do qual os serviços especificados pela interface de um componente possam ser "consumidos" de maneira direta, independentemente do mecanismo utilizado na

comunicação e possibilitando, adicionalmente, a troca do padrão de distribuição sem incorrer em alterações na aplicação desenvolvida.

Complementarmente, pretende-se alcançar alguns objetivos específicos, a saber:

i. Facilitar o processo de desenvolvimento de aplicações distribuídas;
ii. Eliminar a necessidade de implementar um par de *proxies* ou *adapters* para cada objeto servidor;
iii. Demonstrar a aplicabilidade de técnicas reflexivas em aplicações distribuídas;

De maneira a atingir os objetivos propostos, esta dissertação está estruturada da seguinte maneira: o Capítulo 1 apresenta três mecanismos de distribuição e analisa suas características específicas, a partir das quais os componentes do *framework* serão definidos e implementados. No Capítulo 2 os conceitos de reflexão computacional são explanados, juntamente com as abordagens disponíveis na linguagem Java de forma a destacar seu papel na separação de características não-funcionais. No Capítulo 3 são demonstradas as características do *framework* proposto, os padrões de projeto utilizados em sua concepção e as estratégias utilizadas em sua elaboração. O capítulo também demonstra a implementação realizada para cada mecanismo de distribuição apresentado no Capítulo 1, juntamente com uma pequena aplicação utilizada para comprovar a aplicabilidade da solução e mensurar o impacto na performance. Finalmente, o Capítulo 4 concluí esta dissertação, discorrendo sobre as contribuições da pesquisa, os trabalhos relacionados e as propostas de continuação deste trabalho.

Capítulo 1

1 Mecanismos de Distribuição

Este capítulo discorre sobre os mecanismos de distribuição contemplados na implementação do *framework*, apresentada no Capítulo 3. A escolha dos padrões RMI, CORBA e SOAP (JAX-RPC) foi definida devido ao suporte oferecido pela linguagem Java para as três especificações, bem como pela abrangência da amostra: enquanto a primeira especificação trata de aplicações homogêneas (Java) as duas últimas são direcionadas a linguagens heterogêneas, evidenciando a aplicabilidade dos conceitos desta dissertação em outros ambientes. Embora longe de esgotar o conteúdo das especificações de cada mecanismos, a explanação aqui apresentada proporciona a fundamentação necessária para os objetivos propostos neste trabalho.

1.1 RMI

Remote Method Invocation (RMI) (SUN, 2002a) é o mecanismo disponibilizado na plataforma Java para possibilitar a invocação de métodos entre objetos distribuídos. Em RMI um objeto é considerado remoto quando seus métodos são invocados a partir de outra Java Virtual Machine (JVM), provavelmente executada em um dispositivo distinto, e interligados através de uma rede.

Assim como em aplicações baseadas em RPC, toda a complexidade envolvida em uma chamada remota é gerenciada pela especificação RMI. Embora tal analogia possa, a princípio, sugerir que as abordagens são idênticas, a especificação RMI vai além do modelo de chamadas a procedimentos ao definir um modelo de objetos distribuídos, cujo objetivo principal é proporcionar a mesma sintaxe e uma semântica similar àquela utilizada em objetos locais.

Uma aplicação desenvolvida via RMI é dividida em duas camadas: uma servidora e outra cliente. A função da aplicação servidora é instanciar os objetos que serão remotamente acessados, tornar suas referências remotas disponíveis, e aguardar pelas chamadas de métodos que deverão ser atendidas por estes objetos. A aplicação cliente, por sua vez, executa operações específicas para localizar o objeto remoto, obter sua referência e, a partir desta, invocar os métodos previstos na referência obtida.

1.1.1 Tratamento de parâmetros e referências remotas

A especificação RMI restringe o uso de parâmetros e valores de retorno de métodos remotos a elementos que sejam *serializáveis*. Serialização (SUN, 2001) é o processo de representar objetos ou tipos de dados em um fluxo de *bytes* (*byte streams*) e, posteriormente, recuperá-los a partir do mesmo fluxo. Em Java, tal característica é implementada como uma API que estende o mecanismo padrão de escrita e leitura da linguagem, adicionando suporte a objetos. São passíveis de *serialização* em Java as classes que implementam a interface j*ava.io.Serializable*, e os tipos de dados primários (int, float, long, etc). Além destes, também podem ser incluídos como argumentos de métodos remotos objetos que implementem uma interface remota (*java.rmi.Remote*).

Embora a passagem de parâmetros em uma chamada de método remoto seja sintaticamente idêntica a uma invocação local, algumas diferenças apresentadas pelo RMI devem ser consideradas (SUN, 2002a):

i. objetos não-remotos passados como parâmetro, ou recuperados no retorno de um método são copiados através do mecanismo de *serialização*. Quando tais objetos retornam de uma chamada remota, um novo objeto é criado na JVM que realizou a chamada;
ii. objetos remotos passados como parâmetro ou recuperados no retorno de um método, são representados na JVM cliente através de uma referência a sua interface remota, representada pelo seu *stub*.

Este comportamento implica em uma alteração no conceito de identidade dos objetos existente na linguagem Java, uma vez que a referência a uma interface remota não se caracteriza como uma referência ao objeto que a implementa, mas sim uma referência ao *stub* (proxy local) do objeto (SZYPERSKI, 1999).

Durante o processo de envio e recebimento de parâmetros e retorno de métodos, RMI se vale de um recurso que o torna extremamente flexível: o carregamento de classes dinâmico (*Dynamic Class Loading*) (SUN, 2002a). Durante o processo de *marshall* dos argumentos, é realizada uma anotação no fluxo de bytes de forma a identificar a localização da classe que define a instância sendo transmitida. No lado oposto, os parâmetros enviados serão reconstruídos, tornando-se objetos ativos na JVM que os receber. Como as definições de classe para cada objeto a ser instanciado são requeridas, o processo de *unmarshall* tenta, inicialmente, recuperar as classes através de

seu nome no contexto da JVM. Caso alguma classe não seja encontrada localmente, o carregamento de classes dinâmico usará a anotação realizada previamente no fluxo de *bytes* para localizar a classe e carregá-la a partir do ponto de origem da chamada.

1.1.2 Exceções

O desenvolvimento de uma aplicação distribuída exige que instruções de interceptação e tratamento de exceções cuja origem seja o mecanismo de distribuição sejam codificadas. A API RMI (SUN, 2002a) define diversas classes de exceção que representam erros ocorridos durante a execução de uma aplicação distribuída. A relação a seguir apresenta os principais pontos de origem de exceções, bem como as classes contidas no pacote *java.rmi* que as representam:

i. Exceções previstas durante a exportação de um objeto remoto. Representam que a tentativa de exportar um objeto que estende a classe UnicastRemoteObject não foi bem sucedida. StubNotFoundException, server.SkeletonNotFoundException, server.ExportException.

ii. Exceções previstas durante uma chamada de método remoto: UnknownHostException, ConnectException, ConnectIOException, MarshalException, NoSuchObjectException, StubNotFoundException, activation.ActivateFailedException.

iii. Exceções previstas durante o retorno de uma chamada de método remoto: UnmarshalException, UnexpectedException, ServerError, ServerException, ServerRuntimeException.

iv. Exceções ligadas ao serviço de nomes (*Naming*): AccessException, AccessException, NotBoundException, UnknownHostException.

1.1.3 Protocolos de transporte

Conforme citado anteriormente, todas as chamadas e seus valores de retorno são formatados usando a API de *serialização* disponibilizada pela plataforma Java. Uma vez representados desta maneira, os fluxos de *bytes* são enviados para seu destino através de uma conexão TCP direta entre os computadores participantes da comunicação. Existem, todavia, situações em que o protocolo pode não ser efetivo na comunicação. Uma situação comum se apresenta quando um *firewall* rejeita os pacotes por não reconhecer o protocolo, impossibilitando a conexão e ocasionando uma exceção na aplicação cliente (MCPHERSON, 1999).

A primeira alternativa para evitar este problema é disponibilizada pelo próprio RMI: uma vez que a conexão TCP tenha falhado, uma nova tentativa de conexão é realizada através do protocolo HTTP (FIELDING et al., 1999) que, por concepção, possui a característica de ser reconhecido como confiável pelos *firewalls*.

Complementarmente, RMI proporciona um *framework*[1] destinado à construção de *sockets* especializados, que podem ser utilizados para implementar novos meios de comunicação (com outros protocolos além do TCP e HTTP) além de efetuar operações sobre o conteúdo a ser enviado, tornando possível a construção de conexões com características de criptografia ou compactação.

1.1.4 Características de implementação

Esta seção demonstra a interface de programação disponibilizada pela especificação RMI através de um exemplo de aplicação distribuída. Embora extremamente simplificado este exemplo apresenta os passos e o código necessários no desenvolvimento baseado em RMI.

Conforme citado anteriormente, todo objeto remoto em RMI é uma instância de uma classe que implemente ao menos uma interface remota. Esta interface especifica quais métodos poderão ser invocados a partir de outra JVM. Para o exemplo proposto, a interface remota é definida pelo seguinte código:

```
package org.test.examples;

import java.rmi.Remote;
import java.rmi.RemoteException;
import org.test.examples.PedidoNaoEncontrado;
import org.test.examples.Pedido;

public interface SimpleService extends Remote {
  Pedido getPedido(int numero) throws RemoteException,
                                          PedidoNaoEncontrado;
}
```

Além da obrigatoriedade de estender a interface *java.rmi.Remote*, uma interface remota deve ser, preferencialmente, declarada como pública para evitar restrições de acesso durante sua utilização pela aplicação cliente que, possivelmente, estará implementada em outra estrutura de pacotes[2]. Complementarmente cada método

[1] *Framework* é um conjunto de classes que possibilita o reuso de mecanismos elaborados de forma a solucionar um problema de projeto. Um *framework* pode ser constituído de classes abstratas, exigindo uma implementação que adicione o comportamento desejado com base na interface previamente definida, ou de classes que disponibilizam uma implementação padrão que possa ser estendida através de classes especializadas (SZYPERSKI, 1999).

[2] Java organiza suas classes sob a forma de pacotes, que são, na verdade, a representação da estrutura de diretórios onde estão gravados os arquivos (GOSLING et al., 2000).

presente na interface deve declarar, além das exceções específicas do domínio da aplicação, a exceção *java.rmi.RemoteException* em sua cláusula de *trhows* devido à suscetibilidade a problemas de comunicação na chamada remota do método (WALDO et al., 1994).

Uma vez definida a interface remota é necessário criar uma classe que a implemente. Em Java, quando determinada classe implementa uma interface estabelece-se um contrato entre a classe e o compilador pelo qual a classe se compromete a prover a definição de cada método listado na interface (GOSLING et al., 2000). Desta forma a plataforma garante que para cada método previsto existe uma implementação correspondente.

Classes remotas utilizam os serviços de distribuição RMI a partir da declaração de sua superclasse. Neste exemplo, a declaração *extends UnicastRemoteObject* da classe SimpleServiceImpl garante que toda instância desta classe utilizará o mecanismo padrão de comunicação baseado em *sockets*, e será automaticamente exportada na execução de seu construtor, tornando-se apta a receber chamadas de método a partir de portas anônimas. Assim como nos demais métodos da interface, o construtor da classe também deve declarar a exceção *RemoteException*. Embora este seja o meio mais direto para implementar uma classe remota a herança pode ser realizada a partir de outras superclasses de modo que requisitos distintos, como o uso de diferentes protocolos de comunicação, possam ser atendidos. Diante disto, a classe SimpleServiceImpl é implementada pelo seguinte código:

```
public class SimpleServiceImpl extends UnicastRemoteObject
                                implements SimpleService {

   public SimpleServiceImpl() throws RemoteException {
   (...)
   }
   public Pedido getPedido(int numero) throws RemoteException,
                                              PedidoNaoEncontrado {
      Pedido pedido = null;
      (...)
      return pedido;
   }
}
```

Finalmente, para que os aplicativos sendo executados em outras máquinas virtuais possam utilizar os métodos implementados na classe remota, é necessário que uma instância da classe seja criada e registrada. Para realizar esta tarefa, a especificação RMI disponibiliza um serviço de nomes denominado *RMIRegistry*, que permite às aplicações cliente obter uma referência de um objeto remoto a partir de um nome aleatoriamente

atribuído. O processo de registro do objeto remoto, denominado *binding* (SUN, 2002a), é realizado da seguinte maneira:

```
public static void main(String args[]) {
   try {
      SimpleServiceImpl obj = new SimpleServiceImpl();

      Naming.rebind("//127.0.0.1/SimpleService", obj);

   } catch (RemoteException e) {
      e.printStackTrace();
   } catch (MalformedURLException e) {
      e.printStackTrace();
   }
}
```

Como pode ser observado no fragmento de código acima, o método *rebind* da classe *Naming* realiza o registro da instância desejada a partir da definição de uma *string* que representa a localização e o nome do objeto remoto. Por questões de segurança, a especificação RMI determina que o processo de registro de um objeto deve ser realizado por uma aplicação executada no mesmo computador que o *RMIRegistry*, evitando assim que os registros sejam alterados inadvertidamente. A partir deste momento, o objeto torna-se disponível para atender às solicitações remotas.

De maneira similar ao registro no servidor, a aplicação cliente também utiliza a classe *Naming* para localizar e obter uma referência remota do objeto a ser utilizado (SUN, 2002a). Porém ao invés de identificar o tipo da referência obtida como sendo uma instância da classe *SimpleServiceImpl*, o cliente a reconhece como uma instância da interface remota *SimpleService*. Esta abstração possibilita um isolamento entre as duas aplicações, possibilitando alterações na classe servidora que implementa a interface remota definida, sem exigir alterações no cliente. A partir da referência obtida, cliente pode realizar as invocações de método exatamente como em qualquer objeto local. O código abaixo demonstra a aplicação localizando o objeto, obtendo a referência remota e realizando a chamada a um método:

```
public void runTest() {
   SimpleService remoteObject = null;
   try {
      remoteObject = (SimpleService) Naming.
             lookup("//127.0.0.1/SimpleService");

      Pedido resultado = remoteObject.getPedido();

   } catch (MalformedURLException e) {
      e.printStackTrace();
   } catch (NotBoundException e) {
      e.printStackTrace();
   } catch (RemoteException e) {
      e.printStackTrace();
   }
}
```

Pode-se constatar que apesar da chamada ao método utilizar a sintaxe padrão da linguagem Java, é necessário prever as exceções próprias de uma aplicação distribuída como, por exemplo, a exceção *RemoteException* que pode eventualmente ser gerada.

1.2 CORBA

Common Object Request Broker Architecture (CORBA) (OMG, 2001) é um padrão de middleware definido pelo Object Management Group (OMG). Trata-se de uma especificação para a qual existem diversas implementações comercialmente disponíveis como o Visibroker da Borland e o Orbix da Iona, bem como opções de código aberto a exemplo do OpenORB.

Um dos principais objetivos da especificação CORBA é proporcionar um modelo de distribuição orientado a objetos independente da linguagem de programação utilizada no desenvolvimento de cada componente[3]. Para isso CORBA define uma linguagem puramente declarativa denominada *Interface Definition Language* (IDL), cujo intuito é oferecer suporte à definição de constantes, estruturas de dados e, principalmente, interfaces de objetos. Uma definição de interface de objeto em IDL pode conter:

i. os atributos de um objeto, seus tipos e os modificadores de acesso, como somente leitura ou leitura e escrita;
ii. os métodos implementados pelo objeto, com seus parâmetros, tipos, exceções e valores de retorno.

A partir das definições em IDL, as aplicações cliente e servidora podem ser implementadas em qualquer linguagem que disponha de um mapeamento entre os conceitos presentes na IDL e suas próprias estruturas. O padrão CORBA define mapeamentos para diversas linguagens de programação (OMG, 2003), possibilitando a representação de constantes, tipos de dados, interfaces e exceções diretamente na linguagem destino. O processo de mapeamento é normalmente automatizado através de um compilador de IDL, e resulta em código-fonte destinado a aplicação cliente (*stub*) e servidora (*skeleton*), os quais são posteriormente compilados e "linkados" diretamente nas aplicações.

[3] Embora o cliente CORBA reconheça os serviços como objetos, uma interface IDL não exige que o programa que a implemente seja orientado a objeto, ou seja, a implementação pode ser realizada em qualquer paradigma (OMG, 2001).

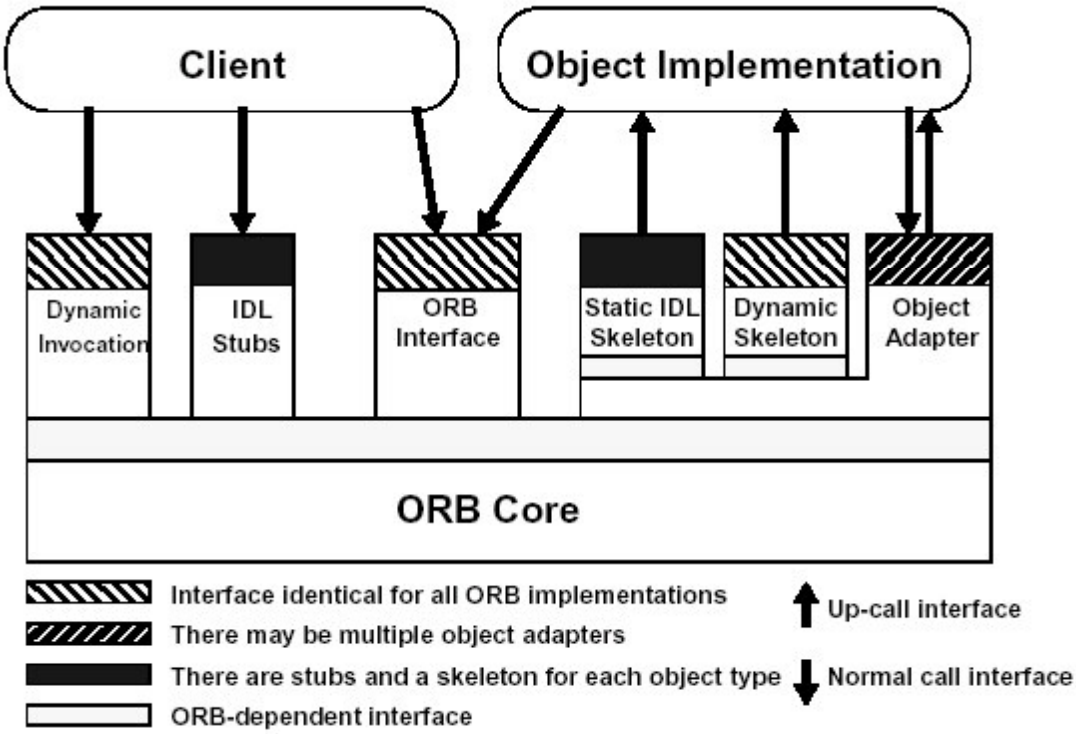
Assim como em RMI, a especificação CORBA abstrai os mecanismos envolvidos na utilização de um objeto remoto, proporcionando uma sintaxe idêntica àquela encontrada em objetos locais para invocação de métodos remotos, ou seja, os *stubs*, também conhecidos como objetos *proxy* cliente (SZYPERSKI, 1999), e os *skeletons* encapsulam a lógica necessária ao processo de *marshaling* e *unmarshaling* dos argumentos envolvidos na invocação, conforme o mapeamento de cada linguagem.

A invocação de métodos através de *stubs* e *skeletons* gerados pelo compilador de IDL ocorre de maneira estática, sendo que uma alteração na interface definida por um objeto exige uma nova compilação para habilitar a aplicação cliente a utilizar os serviços remotos. Como forma de flexibilizar esta situação, CORBA oferece um mecanismo de *Dynamic Invocation Interface* (DII) no lado cliente e *Dynamic Skeleton Interface* (DSI) no servidor. Estes elementos componentes da arquitetura possibilitam a representação dinâmica da chamada de um método a partir da definição do objeto que receberá a invocação, do método a ser executado e dos parâmetros utilizados na chamada. Estas informações são repassadas através do ORB que as executa normalmente no servidor, devolvendo ao cliente um valor de retorno ou uma exceção.

Uma vez compiladas, as definições IDL são inseridas num repositório de interfaces, através do qual poderão ser utilizadas pelas aplicações/cliente no momento da recuperação de uma referência remota. De maneira similar, o objeto servidor que implementa uma determinada interface IDL deve ser registrado no repositório de implementação, habilitando o ORB a reconhecer sua presença.

Além da de independência de implementação proporcionada pelo IDL (OMG, 2001), uma implementação CORBA deve também disponibilizar a transparência de acesso e localização de objetos. Tais características são implementadas pelo principal componente da arquitetura CORBA, o *Object Request Broker* (ORB) que se encarrega de localizar os objetos remotos, enviar as invocações de método a estes objetos, esperar pelos resultados e enviá-los ao cliente que realizou a chamada, intermediando, neste processo, os componentes anteriormente mencionados. A arquitetura do ORB é apresentada na figura 1:

Figura 1 A estrutura do *Object Request Broker*



Fonte: (OMG, 2001)

A seguir segue uma breve descrição dos componentes presentes na arquitetura do ORB (OMG, 2001):

i. Interface ORB: é uma interface de programação que proporciona acesso direto ao ORB, sendo padrão entre todas as implementações e independente dos objetos ou seus adaptadores. Seus serviços podem ser utilizados nas aplicações cliente e/ou servidoras;

ii. Repositório de interfaces: é um serviço que disponibiliza, em tempo de execução, um conjunto de objetos persistentes que representam as informações das IDLs registradas no ORB. Pode ser utilizada pela aplicação cliente para descobrir objetos cuja interface não seja previamente conhecida, e também para armazenar informações auxiliares a outros processos, como *debug*.

iii. Interface de invocação dinâmica: permite a construção de chamadas de método dinâmicas pelas aplicações cliente, dispensando os *stubs*;

iv. *Stubs* IDL: representam os objetos e suas operações disponíveis no ORB acessíveis pelas aplicações cliente, comunicando-se com este através de APIs específicas e otimizadas para cada implementação;
v. *Skeletons* IDL: representam os objetos exportados no servidor. Embora normalmente exista um *skeleton* para cada *stub*, esta regra não é obrigatória, pois os clientes podem invocar objetos remotos dinamicamente.
vi. Invocação dinâmica de *skeletons*: interface presente no servidor que possibilita ao ORB manusear dinamicamente as invocações de métodos, prescindindo o uso de um *skeleton* específico para o objeto;
vii. Adaptadores de objetos: é o mecanismo primário através do qual os objetos acessam os serviços do ORB.
viii. Repositório de implementação: contém as informações utilizadas pelo ORB para localizar e inicializar os objetos registrados sendo, geralmente, específicas para cada implementação ou ambiente operacional.

1.2.1 Tratamento de parâmetros e referências

CORBA determina a forma de passagem de parâmetros examinando o tipo formal dos argumentos recuperado a partir da assinatura do método a ser invocado (OMG, 2001). A partir de tal informação o ORB define se o argumento será passado por valor (cópia) ou por referência, de forma que (OMG, 2001; SZYPERSKI, 1999):

i. Objetos cujos propósitos sejam somente encapsular dados (denominados *value types*) serão sempre passados por valor. Esta regra inclui também tipos de dados como inteiros, floats, caracteres, strings, estruturas, seqüências, arrays entre outros;
ii. Objetos CORBA, que representam os serviços remotos, não podem ser passados por valor, portanto o ORB envia uma referência remota do objeto.

Como pode ser observado, o mecanismo adotado é similar ao visto anteriormente em Java RMI, inclusive quanto à identidade dos objetos e referências remotas que, também em CORBA, são diferentes dos objetos originais. Na realidade o processo de envio dos parâmetros não realiza o *marshal da* instância efetivamente, mas sim do estado da instância representado pelos valores de seus atributos (OMG, 2001). As informações contidas no fluxo de *bytes* são então utilizadas pela aplicação receptora

para criar uma nova instância através do processo de *unmarshall* dos dados que indicam o estado do objeto. Este processo assume que o contexto que recebe o parâmetro possui a implementação do tipo compatível com aquela presente no contexto que realizou o envio.

1.2.2 Exceções

CORBA divide suas exceções em dois grupos: aquelas especificadas pela OMG que fazem parte do padrão e exceções de usuário que são totalmente definidas pelo desenvolvedor da aplicação (SUN, 2002b).

A linguagem IDL dispõe de instruções específicas para a declaração de exceções de usuário. Uma exceção é definida por um identificador e uma declaração do tipo *exception,* juntamente com o tipo dos valores de retorno conforme relacionados em sua declaração "*member*". A sintaxe utilizada em tal declaração é (OMG, 2001):

<except_dcl> ::= "exception" <identifier> "{" <member> "}"*

Quando uma exceção é recebida como resultado de uma invocação remota, a aplicação cliente pode determinar exatamente que exceção foi gerada a partir do identificador previamente declarado. Similarmente, todos os membros que tenham sido definidos poderão ser livremente acessados, proporcionando informações complementares que caracterizem o contexto no qual a exceção ocorreu. As exceções de usuário são relacionadas ao domínio da aplicação desenvolvida, e representam ocorrências ligadas à funcionalidade do *software* originadas nos componentes (ou objetos) que o constituem. Como todo elemento que transita entre as aplicações cliente e servidora, as exceções devem ser descritas via IDL e mapeadas para a linguagem escolhida para implementação. Finalmente, tais exceções devem ser vinculadas aos métodos que podem gerá-las através da cláusula *raise* presente na declaração dos métodos, de maneira similar à declaração *throws* em Java (SUN, 2002b).

Exceções de sistema, por sua vez, definem ocorrências não vinculadas ao domínio da aplicação e descrevem erros ocorridos durante a execução de uma solicitação. Em geral, exceções desta categoria são geradas pelas bibliotecas do ORB diante de situações como:

i. Exceções ocorridas no servidor, durante a alocação de recursos ou ativação de um componente.

ii. Exceções de comunicação, como a perda da referência a um objeto remoto, ao servidor ou ao próprio ORB.

iii. Exceções ocorridas no cliente, como o uso de tipos de argumentos inválidos, ou uma definição incorreta de uma invocação dinâmica.

Como são inerentes a qualquer ORB que implemente a especificação CORBA, não é necessário (e sequer possível) utilizar a expressão *raise* com estas exceções. O próprio ORB se encarrega de gerá-las, de forma que toda aplicação cliente deve estar preparada para lidar com suas eventuais ocorrências (OMG, 2001).

Similarmente às exceções de usuário, exceções de sistema também dispõem de informações mais detalhadas a respeito do contexto no qual foram geradas. Tais informações estão disponíveis nos atributos *minor* e *completed* presentes em tais exceções. Os *Minor Codes* são números inteiros que provêem informações adicionais sobre a natureza da falha que ocasionou a exceção, podendo estar definidos na relação padrão da OMG (2002), ou serem específicos da implementação de ORB utilizada. O atributo *completed*, por sua vez, determina o status da operação que causou a exceção, o qual pode ser (OMG, 2001):

i. COMPLETED_YES, quando o objeto invocado conseguiu completar seu processamento antes da ocorrência da exceção.

ii. COMPLETED_NO, quando a exceção ocorreu antes da invocação do método.

iii. COMPLETED_MAYBE, quando o ORB não consegue identificar o status da invocação.

1.2.3 Protocolos de transporte

General Inter-ORB Protocol (GIOP) (OMG, 2001) é um protocolo definido pela especificação CORBA cujo principal objetivo é garantir a interoperabilidade entre implementações de ORBs distintas. GIOP não prescreve o protocolo de transporte a ser utilizado, ao invés disto especifica característica que possibilitem sua utilização sobre qualquer protocolo de transporte orientado a conexão que satisfaça seus requisitos. O padrão GIOP descreve:

i. A definição *Common Data Representation* (CDR), que define a sintaxe de transferência entre os tipos de dados previstos na IDL para o formato a ser

utilizado nas transferência entre ORBs. Todos os tipos de dados previstos na IDL possuem sua representação em CDR.

ii. Os Formatos de Mensagem GIOP são utilizados troca de mensagens entre ORBs, e oferecem suporte a todas as funções requeridas em CORBA, como tratamento de exceções, passagem de contexto e operações sobre referências de objetos remotos.

iii. As Premissas de Transporte GIOP, que descrevem quais características mínimas deve possuir um protocolo de transporte para atender ao envio de mensagens.

Juntamente com o GIOP, a OMG definiu um mapeamento entre este protocolo e o TCP/IP, criando assim o *Internet Inter-ORB Protocol* (IIOP) como o padrão mínimo exigido de qualquer implementação compatível com sua especificação. Contudo tal determinação não é invasiva a ponto de especificar como os ORBs devem proporcionar tal suporte, deixando a cargo de cada fornecedor o uso da abordagem que lhe seja mais conveniente. O único requisito é a possibilidade de comunicação através de mensagens IIOP.

HALTEREN et al. (1999) ressalta em seu artigo relativo ao controle de qualidade de serviço (*QoS*) em ambiente CORBA que, embora especificação proposta pela OMG permitisse o suporte de outros protocolos e tecnologias de rede, isto freqüentemente era realizado através de extensões proprietárias, comprometendo a interoperabilidade. Diante disso a OMG iniciou o desenvolvimento de um novo padrão, denominado *Open Communication Interface* (OCI), cuja abordagem é uma estabelecer uma interface interna ao ORB que abstraia os diferentes protocolos. Desta forma, a interoperabilidade é mantida sem a necessidade de padronização dos protocolos utilizados na comunicação entre os aplicativos cliente e servidor.

1.2.4 Características de implementação

Os elementos presentes em uma aplicação CORBA são: a interface em IDL representando a interface do objeto remoto, a aplicação cliente que utiliza o serviço, o objeto servidor que implementa o serviço e o servidor (ORB) que intermedia a comunicação (OMG, 2001).

Similarmente ao apresentado em RMI, o ponto de partida do desenvolvimento em CORBA é a definição da interface que descreve os serviços implementados por um objeto remoto e que poderão, posteriormente, ser invocados pelas aplicações cliente. O trecho de código abaixo define os mesmos serviços vistos no exemplo de RMI, porém escritos em IDL:

```
module Examples {

   exception RemoteException { };

   exception PedidoNaoEncontrado { };

   interface SimpleService {

   Pedido getPedido(in double numero) raises RemoteException,
                                            PedidoNaoEncontrado;
  };
};
```

A declaração *module* representa em IDL o mesmo conceito de pacotes da linguagem Java, ou seja, trata-se de um *namespace* ou contexto no qual interfaces e declarações relacionadas estão contidas (OMG, 2000). No exemplo acima, o modulo *Examples* declara duas exceções de usuário e uma interface, o *SimpleService*, com um método *getPedido*. Como o retorno deste método não é um tipo de dados previsto em IDL, mas sim um elemento mapeado a partir do domínio da aplicação, a definição em IDL faz-se necessária:

```
module Examples {

   (...)

   valuetype Pedido {
      public double numero;
      public string descricao;
   };

  (...)
};
```

À definição do módulo, segue-se a implementação do software que forneça e consuma os serviços previstos. Como a especificação CORBA determina a interoperabilidade entre linguagens, as aplicações cliente e servidora podem ser implementadas de forma heterogênea, sem nenhum prejuízo em sua funcionalidade (OMG, 2001). No contexto deste exemplo, a implementação será homogênea, baseada na linguagem Java.

O mapeamento de uma definição em IDL para Java é realizado por um compilador denominado *idlj*, especificamente projetado para gerar as classes necessárias à realização de uma determinada IDL. Esta ferramenta utiliza um arquivo com a

extensão *.idl* como argumento de entrada e produz, para o módulo definido neste exemplo, as seguintes classes (SUN, 2002b; OMG, 2000):

i. *SimpleServicePOA.java*: é o *skeleton* utilizado pelo ORB e serve como base para o objeto servidor, que deve estender esta classe. Implementa as interfaces *InvokeHandler* e *SimpleServiceOperations*, além de estender *org.omg.PortableServer.Servant.*

ii. *_SimpleServiceStub.java*: é o *stub* utilizado pela aplicação cliente. Implementa a interface *SimpleService.java* e estende a classe *org.omg.CORBA.portable.ObjectImpl*.

iii. *SimpleService.java*: esta é a interface em Java que representa diretamente a interface IDL. Como se trata de um objeto CORBA, esta classe estende *org.omg.CORBA.Object*, além de implementar a interface *SimpleServiceOperations;*

iv. *SimpleServiceHelper.java*: classe auxiliar cujo principal objetivo é realizar a escrita e leitura da classe *SimpleService* para o fluxo de bytes utilizado pelo ORB em sua comunicação.

v. *SimpleServiceHolder.java*: classe auxiliar utilizada durante a manipulação de parâmetros do tipo *SimpleService* declarados como *in* ou *inout* em IDL.

vi. *SimpleServiceOperations.java*: como os *stubs* e *skeletons* implementam os mesmos métodos, o compilador gera esta interface Java com os métodos previstos na IDL de maneira a reaproveitar o código e garantir do contrato entre ambos.

Além das classes acima relacionadas, o *idlj* também gera, para cada *valuetype*, as classes *Helper*, *Holder* e *Factory*, e, para cada exceção, as classes *Helper*, *Holder* juntamente com as classes que representam tais elementos em Java.

A aplicação cliente utiliza a API provida pela linguagem na qual foi implementada para interagir com o ORB (OMG, 2001). Esta interação envolve os seguintes passos: inicialização de uma referência local ao ORB, obtenção do contexto inicial do servidor de nomes, obtenção de uma referência local a partir do nome do objeto remoto e, finalmente, a invocação dos métodos desejados. O código abaixo demonstra estes passos:

```
(...)
try {
   ORB orb = ORB.init(parametros, null);
```

```
   org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");

   NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

   SimpleService servico = SimpleServiceHelper.
                              narrow(ncRef.resolve_str("SimpleService"));

   servico.getPedido(1);

}
catch (Exception e) {
   e.printStackTrace();
}
(...)
```

A referência ao ORB é inicializada a partir de parâmetros que indicam a localização do servidor como, por exemplo, o endereço IP da máquina e a porta usada para conexão (SUN, 2002b). O objeto representando o ORB é então utilizado para recuperar o contexto inicial do servidor de nomes, o qual pode ser localizado pela *string* *NameService* em qualquer servidor padrão CORBA. A aplicação cliente utiliza o contexto para recuperar as referências dos objetos remotos registrados no servidor, a partir de seu nome. No exemplo acima, foi solicitada uma referência ao objeto registrado sob o nome *SimpleService*. O resultado desta operação pode ser uma exceção, caso o objeto não seja encontrado, por exemplo, ou um objeto genérico do tipo *org.omg.CORBA.Object*. Este objeto é então convertido para seu tipo correto a partir do método *narrow* disponível nas classes *Helper* sendo que este processo também pode resultar em uma exceção caso os tipos sejam incompatíveis ou a referência seja nula. Uma vez de posse da referência, a aplicação cliente passa a invocar seus métodos sem a necessidade de APIs CORBA exceto, evidentemente, pelo tratamento das exceções que possam ocorrer na chamada remota.

O desenvolvimento da parte servidora de uma aplicação distribuída CORBA envolve dois elementos (OMG, 2001; OMG, 2000): o objeto remoto e um programa servidor que registre a instância do objeto no ORB. O objeto remoto, também denominado *servant*, deve implementar a interface que detalha os métodos remotos (*SimpleService*), provendo o código relativo à lógica da aplicação. As instruções relativas à interação entre o *servant* e o ORB que o contém são gerenciadas pelo *skeleton* (*SimpleServicePOA*), que se encarrega de lidar separadamente com as idiossincrasias do ORB devendo, portanto, ser estendido pela classe que define o objeto remoto. A implementação para o exemplo apresentado é a seguinte:

```
class SimpleServiceImpl extends SimpleServicePOA {

(...)
```

```
   public Pedido getPedido(double numero) throws RemoteException,
                                                 PedidoNaoEncontrado;
      Pedido pedido = null;
      (...)
      return pedido;
   }

(...)

}
```

Finalmente, o programa servidor se encarrega de inicializar uma instância do ORB e tornar o objeto remotamente acessível. Para isso os seguintes passos são realizados: criar o ORB, obter uma referência ao gerenciador de adaptadores de objetos (*POAManager*), criar a instância do objeto que será registrado (*SimpleServiceImpl*), obter a referência ao contexto no qual o objeto será vinculado, localizar a raiz deste contexto, registrar a instância do objeto no contexto obtido para, então, instruir o ORB a esperar pelas invocações de métodos. Estes passos estão codificados no exemplo a seguir:

```
(...)
try {

   ORB orb = ORB.init(parametros, null);
   POA raizPOA = POAHelper.narrow(orb.
                                 resolve_initial_references("RootPOA"));
   raizPOA.the_POAManager().activate();

   SimpleServiceImpl simpleServiceImpl = new SimpleServiceImpl();
   simpleServiceImpl.setORB(orb);

   org.omg.CORBA.Object obj = raizPOA.servant_
                                 to_reference(simpleServiceImpl);
   SimpleService simpleServiceRef = HelloHelper.narrow(obj);

   org.omg.CORBA.Object nameServiceRef =
      orb.resolve_initial_references("NameService");
   NamingContextExt namingRef = NamingContextExtHelper.
                                 narrow(nameServiceRef);

   NameComponent path[] = namingRef.to_name("SimpleService");
   namingRef.rebind(path, simpleServiceRef);

   orb.run();
}
catch (Exception e) {
   e.printStackTrace();
}
(...)
```

O desenvolvimento deste exemplo foi baseado na invocação estática a partir dos *stubs* e *skeletons* previamente gerados pelo compilador de IDL. Informações sobre o desenvolvimento de aplicações CORBA baseadas na invocação dinâmica de métodos podem ser obtidas em (OMG, 2000; OMG, 2001).

1.3 RPC baseado em XML

Como forma de otimizar os recursos envolvidos na distribuição de uma aplicação, CORBA e RMI estabelecem protocolos que representam os dados envolvidos em uma chamada remota de método sob a forma binária. Ao mesmo tempo em que torna os protocolos mais eficientes, exigindo menor uso de memória e capacidade de transmissão da rede para serem manipulados, tal abordagem limita o entendimento do protocolo a produtos que sejam aderentes a suas especificações.

Uma alternativa ao emprego dos protocolos binários na distribuição de aplicações é o uso de XML, o qual tem se firmado como um padrão no intercâmbio de dados entre aplicações heterogêneas. A seguir será realizada uma breve abordagem deste padrão para, na seqüência, relacioná-lo com as abordagens de distribuição que o utilizam.

1.3.1 Linguagens de marcação e XML

Diferentemente das linguagens de programação, as linguagens de marcação não processam informações. Ao invés disso, elas são utilizadas para identificar os elementos que constituem um documento de forma a facilitar seu processamento pelos programas que os utilizam. Criado a partir da *Standard Generalized Markup Language* (SGML), o HTML possibilitou o explosivo crescimento da Internet, ao disponibilizar um meio simples de representação de conteúdo sob a forma de hipertexto. Por se tratar de uma linguagem que não prevê extensões de maneira padrão e trata a apresentação juntamente com os dados, o HTML não se mostrou adequado para a transmissão de informações complexas entre aplicações cliente e servidoras (BEDUNAH, 1999). Diante da necessidade de um modelo mais flexível, o *World Wide Web Consortium* (W3C) criou uma nova linguagem de marcação denominada *Extensible Markup Language* (XML) como um subconjunto da SGML (W3C, 2000). Ao contrário do HTML que possuí marcações (*tags*) definidas, o XML caracteriza-se como uma meta-linguagem, a partir da qual os documentos podem ser definidos de maneira extensível, de acordo com a informação que devem ser manipuladas. Os objetivos da especificação, definidos pelo W3C, são:

i. Ser diretamente aplicável na Internet.
ii. Oferecer suporte a uma ampla variedade de aplicações.
iii. Ser compatível com SGML.
iv. Facilitar a implementação de programas que manipulem documentos XML.

v. Minimizar o número de características opcionais da especificação, mantendo-as próxima a zero.
vi. Definir documentos logicamente claros e legíveis por seres humanos.
vii. Oferecer uma forma simples de criar documentos XML.
viii. Definir um modelo formal e conciso.
ix. Minimizar a importância do tamanho das marcações XML.

Um documento XML é um conjunto de dados estruturado lógica e fisicamente. A estrutura lógica é constituída por um ou mais elementos, os quais são delimitados por marcações de início e fim, ou por elementos vazios. Todo documento deve iniciar com um elemento especial, denominado *root* (raiz) no qual todos os demais elementos estão aninhados. A estrutura física é composta por entidades que representam um conjunto de informações internas ou externas, declaradas através de uma referência.

Assim como em SGML, as definições de elementos e entidades em XML são realizadas em um documento separado, denominado *Document Type Definition* (DTD), a partir do qual um *parser*[4] pode verificar sua validade. Um documento é considerado válido quando sua composição respeita a gramática definida pelo DTD quanto aos elementos e entidades permitidos bem como a ordem e o local onde devem estar representados ao longo do documento.

Além do conceito de documento válido, XML acrescentou outra classificação, denominada documento bem-formado (w*ell-formed document*) (W3C, 2000; BEDUNAH, 1999). Um documento bem-formado não está vinculado a nenhum DTD e deve respeitar duas regras:

i. Deve existir um e somente um elemento *root* ao qual todos os demais elementos deverão estar subordinados.
ii. Cada elemento presente no documento deve ser delimitado por uma *tag* de abertura e outra de finalização.

Uma determinada aplicação pode prescindir do uso de um documento DTD quando o documento XML for suficientemente simples, sendo facilmente interpretado pelas aplicações que o utilizam. O exemplo abaixo apresenta um documento bem-formado que representando o acervo de uma biblioteca:

```
<?xml version="1.0" encoding="UTF-8"?>
<biblioteca>
```

[4] Programa responsável por ler e analisar um documento.

```
  <livro>
    <titulo>Titulo Um</titulo>
    <autor>Dr. Autor</autor>
    <ISBN>33354415</ISBN>
  </livro>
  <livro>
    <titulo>Titulo Três</titulo>
    <autor>Dr. Fulano</autor>
    <ISBN>73354415</ISBN>
  </livro>
</biblioteca>
```

Com a inclusão da instrução *<!DOCTYPE Biblioteca SYSTEM "Biblioteca.dtd">* o documento acima passaria a ser validado (caso o programa esteja preparado para isso) de acordo com o conteúdo do seguinte DTD:

```
<!—Projetado para representar o acervo de uma biblioteca-->
<!ELEMENT Biblioteca (Livro+)>
<!ELEMENT Livro  (Titulo, Autor, ISBN)>
<!ELEMENT Titulo (#PCDATA)>
<!ELEMENT Autor  (#PCDATA)>
<!ELEMENT ISBN   (#PCDATA)>
<!-- End DTD-->
```

É importante ressaltar que um documento XML sempre representa qualquer tipo de dados como um texto puro. Isto pode ser observado no DTD acima, onde os elementos estão definidos como #PCDATA, o que indica que quaisquer caracteres suportados pela codificação (*encoding*) escolhida podem ser livremente utilizados. Todavia esta liberdade de representação dos dados dificulta o processamento do documento, pois é impossível identificar se o elemento *ISBN* é um inteiro ou uma *string*.

Como forma de estabelecer um padrão de tipos de dados, o W3C definiu o *XML Schema Languages*, o qual descreve 44 tipos de dados simples que podem ser utilizados tanto nos em documentos XML como nos DTDs. Além dos tipos inicialmente previstos, é possível ainda a definição de novos tipos que sejam combinações ou restrições dos tipos básicos (HAROLD, 2002).

1.3.2 Padrões de RPC com XML

XML não prescreve os cenários nos quais deveria ser aplicado, mas sim uma forma de representação consistente e portável de documentos. Desta maneira, qualquer situação onde seja necessário o intercâmbio de informações caracteriza-se como uma aplicação potencial. Chamadas remotas de procedimento, conforme anteriormente citado, realizam exatamente este tipo de comunicação, enviando, basicamente, a representação de uma chamada (com o nome do método/procedimento e seus parâmetros) a um serviço remoto e recebendo a representação do retorno da execução. Este é o princípio das especificações XML-RPC (USERLAND, 2003) e SOAP (W3C, 2001a).

Embora tenham princípios semelhantes, a abordagem e os objetivos definidos por cada protocolo são muito distintos. XML-RPC é um padrão proposto pela Userland Software (http://www.userland.com) cujo princípio é simplificar o processo de distribuição de aplicações heterogêneas. Devido a esta premissa, a especificação propositadamente ignora características presentes nos mecanismos discutidos anteriormente (em CORBA e RMI), como o gerenciamento de memória (*garbage collection*), *stubs* e *skeletons*, serialização de objetos, entre outros. Sua simplicidade é exibida também nos tipos de dados aos quais oferece suporte, restringido-os a *int, boolean, double, dateTime.iso8601* e *base64*, além de *arrays* e estruturas (HAROLD, 2002; USERLAND, 2003) . Também não são utilizados recursos mais avançados de XML, como os DTDs e *namespaces*. Apesar de sua simplicidade, trata-se de um padrão para o qual até a data desta pesquisa foram relacionadas 74 implementações para diversas plataformas, entre as quais C, C++, Java, Delphi, ASP, Apache, Lisp, Perl, PHP e Tcl.

O *Simple Object Access Protocol* (SOAP) é um protocolo idealizado para realizar a troca de informações estruturadas em um ambiente de computação descentralizado e distribuído (W3C, 2003). Sua especificação é definida por um grupo de especialistas em XML de várias empresas, o que o torna mais elaborado quanto ao uso dos recursos disponibilizados por XML. SOAP faz uso de características como atributos, codificação *Unicode, namespaces* que são ignorados por XML-RPC, além de permitir uma representação mais rica dos dados contidos em uma mensagem, utilizando-se para isso

de linguagens de definição de *schemas*, como a *XML Schema Language*. Tais características o tornam apto a lidar com tarefas mais complexas do que aquelas abordadas pelo XML-RPC (HAROLD, 2002).

Fundamentalmente, SOAP é um paradigma de troca de mensagens que não mantém o estado entre o processo de envio/resposta. Este modelo pode, todavia, ser estendido para suportar interações mais complexas como requisição/resposta ou requisição/múltiplas respostas através da combinação das características de comunicação em uma via de SOAP, juntamente com as possibilidades de interação proporcionadas pelo protocolo ou pela aplicação que se encarregam do transporte. As mensagens SOAP são completamente independentes do mecanismo de distribuição utilizado, não se preocupando com questões relacionadas à confiabilidade, segurança, garantia de entrega, roteamento, etc (W3C, 2001a). Os principais pontos definidos na especificação são: o modelo de processamento de mensagens, o modelo de extensibilidade, as regras para ligação (*binding*) com protocolos de transporte e a estrutura da mensagem SOAP (W3C, 2001b).

SOAP pressupõe um modelo distribuído, onde uma mensagem é originada em um ponto inicial (*sender*) e encaminhado ao ponto final (*receiver*) através de um ou mais intermediários. Estes elementos que originam, encaminham e recebem as mensagens, denominados SOAP *Nodes*, devem agir de acordo com o modelo de processamento de mensagens que determina, entre outras ações, a interpretação do cabeçalho e corpo das mensagens, a identificação do papel representado por determinado *node* durante o processamento de uma mensagem específica, a forma de repasse de mensagens e o controle de versões das mensagens.

Uma mensagem SOAP consiste basicamente dos seguintes elementos (W3C, 2001b):

i. *Envelope*: Toda mensagem SOAP deve contê-lo. É o elemento raiz do documento XML. O Envelope pode conter declarações de *namespaces* e também atributos adicionais como a definição da codificação de caracteres (*encoding*).

ii. *Header*: É um cabeçalho opcional. Ele carrega informações adicionais, como por exemplo, se a mensagem deve ser processada por um determinado

nó intermediário. Quando utilizado, o *Header* deve ser o primeiro elemento do *Envelope*.

iii. *Body*: Este elemento é obrigatório e contém o *payload*, ou a informação a ser transportada para o seu destino final. O elemento *Body* pode conter um elemento opcional *Fault*, usado para carregar mensagens de status e erros retornadas pelos *nodes* durante o processamento da mensagem.

Com base nas características do protocolo, a especificação SOAP também define uma representação de chamada de procedimento remoto, facilitando a troca de mensagens que sejam mapeadas de acordo com a forma de invocação de métodos ou procedimentos normalmente encontrados em linguagens de programação. Todavia tal definição se restringe à representação das solicitações (*requests*) e respostas (*responses*), não detalhando o mapeamento para linguagens de programação específicas (W3C, 2003). Para realizar uma invocação remota, são exigidas as seguintes informações:

i. O endereço do serviço que será acessado.
ii. O nome do procedimento ou do método que será invocado.
iii. Os valores e argumentos que serão passados na invocação.
iv. Propriedades requeridas pelo protocolo utilizado na comunicação, como um GET ou POST no caso do protocolo HTTP.
v. Um cabeçalho, opcional.

Tanto as solicitações como as respostas nada mais são do que mensagens SOAP contendo uma representação específica em seu corpo. As solicitações são representadas por uma estrutura que contém cada parâmetro a ser utilizado na chamada, definindo os modificadores *IN ou IN/OUT*. O nome da marcação que constituí esta estrutura deve ser o mesmo que o procedimento ou método a ser chamado, seguindo as convenções de mapeamento de nomes de aplicação para XML.

De forma similar, uma resposta é encaminhada no corpo da mensagem SOAP como uma estrutura única na qual fica aninhado o valor de retorno da chamada e cada parâmetro definido como saída ou entrada e saída (OUT e IN/OUT). Similarmente à definição da solicitação, cada marcação correspondente a um parâmetro deve ser identificada por um nome.

1.3.3 Implementações de SOAP: JAX-RPC

A *Java API for XML-Based RPC* (JAX-RPC) (SUN, 2002c) é uma especificação que habilita aplicações Java e interoperar com serviços baseados em SOAP. Sua interface de programação abstrai as idiossincrasias de SOAP, de forma a proporcionar uma maior produtividade no desenvolvimento fornecendo, além da realização de chamadas remotas, o ferramental necessário para criar e descrever os serviços distribuídos utilizando tecnologias de apoio como a *Web Services Description Language* (WSDL) (CHRISTENSEN et al., 2001).

Uma característica importante desta especificação em relação a CORBA e RMI é a forma de implementação dos serviços remotos (SUN, 2002c). Enquanto nos dois últimos qualquer programa ou classe pode ser disponibilizado remotamente, JAX-RPC vincula a programação de tais serviços a dois modelos de componentes especificados na plataforma *Java 2 Enterprise Edition* (J2EE), os *Servlets* e *Enterprise Java Beans* (EJB), o que exige a presença de um servidor de aplicações para execução de tais componentes. Apesar desta exigência, a especificação garante a interoperabilidade entre diferentes plataformas e implementações de serviços Web (ARMSTRONG et al., 2003; SUN, 2002c) ao basear suas características na versão 1.1 de SOAP e no formato de descrição de serviços WSDL. Isto possibilita o acesso aos serviços por uma aplicação cliente que utilize os mesmos padrões, independentemente da API utilizada em seu desenvolvimento. Da mesma forma, uma aplicação cliente JAX-RPC pode acessar normalmente qualquer serviço que tenha sido criado com SOAP 1.1 e WSDL 1.1. As seções posteriores tratam das características específicas da utilização de SOAP em aplicações através da API JAX-RPC.

1.3.4 Tratamento de parâmetros e referências

JAX-RPC realiza um mapeamento entre os tipos de dados Java para suas definições em XML e WSDL, de forma a compor uma mensagem SOAP. Uma grande gama de mapeamentos é disponibilizada, entre eles (SUN, 2002c):

i. Os tipos de dados primitivos *boolean, byte, short, int, long, float* e *double,* bem como suas classes de cobertura *Boolean, Byte, Short, Integer, Long, Float e Double*.

ii. *Arrays* cujos membros sejam tipos de dados suportados por JAX-RPC.

iii. Classes Java padrão, como *java.lang.String, java.util.Date, java.util.Calendar, java.math.BigInteger, java.math.BigDecimal*.

Cada tipo de dados suportado pela especificação é representado pela definição equivalente na no XML *Schema Definition Language*. Por exemplo, inteiros são mapeados para *xsd:int*, valores lógicos para *xsd:boolean* e assim por diante. Caso um determinado tipo de dados não possa ser diretamente mapeado, JAX-RPC possibilita a definição de procedimentos de *serialização* e *deserialização* customizados, que se encarregam de prover uma representação adequada.

Complementarmente, a API suporta também a definição de *value types*, similares àqueles discutidos previamente na especificação CORBA. Um *value type* JAX-RPC é uma classe Java que atenda a determinadas características, como possuir um construtor padrão público, não implementar direta ou indiretamente a interface *java.rmi.RemoteInterface* e seguir o padrão de codificação dos componentes *JavaBeans*, o qual determina que os atributos da classe devem possuir métodos que realizem a atribuição e obtenção (*setters* e *getters*) de seu conteúdo. Todavia o suporte a estas classes não é abstrato pois, para cada *value type* presente na aplicação, o compilador do ambiente gera classes de apoio ao processo de serialização/deserialização dos objetos.

Diferentemente do procedimento adotado por CORBA e RMI e de acordo com as características do protocolo SOAP (W3C, 2001b), JAX-RPC não exige suporte à passagem de parâmetros por referência, embora ressalte que tal característica possa ser adicionada em implementações específicas, sob pena de perda de portabilidade(SUN, 2002c). Devido a isto, uma classe que caracterize um serviço não deve estender *java.rmi.Remote*. Quanto à passagem de parâmetros por cópia, o mecanismo é similar àqueles vistos anteriormente, onde o valor do objeto sendo passado é primeiramente copiado antes da invocação do método remoto e enviado para a outro ponto, onde uma nova instância é criada e inicializada com os valores dos atributos previamente lidos.

1.3.5 Exceções

SOAP proporciona um mecanismo de tratamento de exceções baseado em dois princípios: a habilidade de identificar as condições que resultem em uma falha e a na capacidade de enviar o erro correspondente ao ponto que enviou a mensagem. Como

SOAP não tem como objetivo prescrever o mecanismo de comunicação a ser utilizado, a especificação se restringe a definir como as faltas são identificadas (W3C, 2001a). Desta forma, o retorno do erro fica a cargo, no caso aqui apresentado, dos mecanismos detalhados pela especificação JAX-RPC.

Na nomenclatura SOAP, uma exceção recebe a denominação de falta (*fault*). Uma vez originadas, as faltas são representadas normalmente através de uma resposta SOAP que contém um único elemento *env:Fault* dentro de seu *env:Body*. Este elemento contém os seguintes sub-elementos (W3C, 2001a):

i. *env:Code* é um nome XML qualificado que proporcionam a identificação da falta, sendo restrito àqueles definidos no tipo `env:faultCodeEnum`. Toda mensagem de falta incorpora uma lista hierárquica de códigos e informações associadas que identificam a categoria da falta em níveis mais detalhados (W3C, 2001b), através do elemento *env:Subcode*. Este elemento pode ser utilizado para armazenar códigos específicos de cada aplicação. É um item obrigatório no corpo da mensagem que representa a falta.

ii. *env:Reason* contém informações voltadas ao entendimento da falha pelos desenvolvedores das aplicações. Embora seja um elemento mandatório, seu conteúdo não é padronizado, de forma que o que se exige é uma descrição inteligível do erro ocorrido. Este elemento deve possuir um ou mais elementos *env:Text*, marcados com a língua na qual estão representados, de forma a possibilitar a internacionalização das mensagens.

iii. *env:Detail* contém informações específicas da aplicação que complementam as informações da falta SOAP ocorrida. É um item não-obrigatório.

iv. *env:Node* é um outro elemento não-obrigatório. Sua função é identificar o local (*node*) onde a falta foi originada.

v. *env:Role* indica o papel desempenhado pelo *node* no momento que a falta ocorreu. Não é obrigatório.

Caso algum elemento obrigatório do cabeçalho de uma mensagem SOAP não seja compreendido ou seu conteúdo não possa ser processado, uma falta também é gerada e a marcação *env:NotUnderstood* é utilizada para delimitar o conteúdo inválido (W3C, 2001a).

A especificação SOAP 1.2 define os seguintes códigos de falta conforme relacionados na tabela a seguir:

Nome	Significado
VersionMismatch	O *node* que processou a mensagem encontrou um elemento inválido ao invés do elemento esperado.
MustUnderstand	Um elemento contendo o atributo *mustUnderstand* não foi compreendido durante o processamento da mensagem.
DataEncodingUnknown	As informações constantes no cabeçalho ou nos itens da mensagem utilizaram uma codificação de caracteres não reconhecida pelo *node* que recebeu a mensagem.
Sender	A mensagem foi gerada de maneira incorreta ou não contém a informação necessária para ser processada.
Receiver	A mensagem não pode ser manipulada por problemas no processamento, e não por algum problema com o conteúdo da mensagem.

A partir do modelo de faltas definidos na especificação SOAP, as implementações deste protocolo podem realizar um mapeamento que caracterize os erros de maneira familiar ao seu modelo de programação.

No caso da JAX-RPC, o modelo adotado é similar ao visto anteriormente em RMI, exigindo que todo método remoto de um ponto de serviço declare a exceção *java.rmi.RemoteException* em sua cláusula *throws*. A partir disto, as implementações baseadas em JAX-RPC, disponibilizam um mapeamento entre as faltas SOAP e exceções derivadas de *RemoteException* a partir das informações disponibilizadas na mensagem correspondente à falta (SUN, 2002c). Os mapeamentos previstos pela especificação JAX-RPC estão relacionados na tabela abaixo:

Código da falta SOAP	Descrição do erro	Exceção mapeada
Soap-env:Server	O servidor foi impossibilitado de tratar a mensagem devido a algum problema temporário	java.rmi.ServerException
Soap-env:DataEncodingUnknown	A codificação utilizada para representar os parâmetros não foi reconhecida pelo server	java.rmi.MarshallException
rpc:ProcedureNotPresent	O procedimento solicitado pela chamada não foi encontrado no ponto de serviço especificado.	java.rmi.RemoteException
rpc:BadArguments	O servidor não reconheceu os argumentos durante o *parse*, ou o parâmetro esperado pelo servidor foi diferente daquele enviado pelo cliente.	java.rmi.RemoteException

Devido a este mapeamento, as aplicações cliente não precisam conhecer os códigos de erro específicos das faltas SOAP, se preocupando somente com exceções padrão a aplicações Java distribuídas.

1.3.6 Protocolos de transporte

Conforme mencionado anteriormente, SOAP não determina qual protocolo deve ser utilizado na transmissão de suas mensagens. Ao invés disso, a especificação detalha um conjunto formal de regras sobre como uma mensagem deve ser carregada, caracterizando o mecanismo de ligação (*binding*) entre o protocolo e as mensagens SOAP (W3C, 2001b). O SOAP *Protocol Binding Framework* detalha os requisitos e conceitos comuns a qualquer definição de *binding*, possibilita o reuso de características comuns a vários *bindings* e facilita a definição de características opcionais consistentes. É importante ressaltar que um *binding* não se caracteriza como um novo elemento (*node*) SOAP, mas sim como parte integral de qualquer *node.*

A partir das regras descritas no *framework*, uma especificação de *binding* para um determinado protocolo é criada de modo a determinar a funcionalidade proporcionada, a maneira como os serviços do protocolo serão utilizados para transmitir as mensagens, a forma pela qual a funcionalidade prometida será garantida pelo protocolo, o modelo de tratamento das possíveis falhas juntamente com os requisitos a serem atendidos pelas implementações deste *binding* (W3C, 2001b).

A única especificação de *binding* definda por SOAP é voltada a utilização do protocolo HTTP (FIELDING et al., 1999) para troca de mensagens. Seu objetivo não é explorar todas as características do HTTP, mas sim permitir que diversos *nodes* SOAP utilizando diferentes implementações deste mesmo *binding* possam interagir (W3C, 2003). Para cada possível estado de um *node* SOAP, esta especificação determina como o protocolo deve ser utilizado para propiciar o comportamento desejado. Um *node* SOAP realizando uma solicitação pode estar em um dos seguintes estados: inicialização, requisição, enviando e recebendo, recebendo e sucesso ou falha. Um *node* que esteja respondendo a uma solicitação pode estar nos mesmos estados, exceto “recebendo”, que é substituído por “enviando”.

Complementarmente, ressalta-se que enquanto SOAP define o *binding* HTTP como opcional, a especificação JAX-RPC exige que suas implementações

proporcionem o mecanismo de transporte baseado no HTTP 1.1, o qual será utilizado nas considerações posteriores deste trabalho.

1.3.7 Características de implementação

Por se tratar de uma especificação do mesmo fabricante, RMI e JAX-RPC possuem grandes semelhanças quanto ao modelo de programação das aplicações servidora e cliente. No decorrer desta seção serão destacados os pontos divergentes entre as duas abordagens.

Similarmente a CORBA e RMI, o desenvolvimento de aplicações baseados na API JAX-RPC inicia com a definição da interface do serviço a ser implementado na qual os métodos disponíveis remotamente são descritos (ARMSTRONG et al., 2003). O código abaixo representa a interface prevista no serviço remoto:

```
package org.test.examples;

import java.rmi.Remote;
import java.rmi.RemoteException;
import org.test.examples.PedidoNaoEncontrado;
import org.test.examples.Pedido;

public interface SimpleService extends Remote {
   Pedido getPedido(int numero) throws RemoteException,
                                       PedidoNaoEncontrado;
}
```

Como pode ser observado, o programa acima é idêntico àquele visto no exemplo relativo a RMI devido, principalmente, à semelhança dos requisitos exigidos:

i. A obrigatoriedade de estender a classe *Remote*.

ii. A declaração de constantes não é permitida.

iii. Os métodos devem declarar a exceção *RemoteException* ou uma de suas subclasses em sua cláusula de *trows*, bem como exceções específicas da aplicação.

A diferença mais significativa entre as duas especificações é quanto aos tipos de dados passíveis de utilização: enquanto RMI permite qualquer tipo primário ou classe serializável, os tipos utilizados em JAX-RPC são aqueles para os quais existe um mapeamento definido.

As semelhanças com RMI são visíveis também na implementação da classe que concretiza os serviços previstos na interface, na qual as principais divergências são a definição do construtor, que não é obrigado a declarar uma exceção remota, e a

declaração da própria classe, que não precisa declarar nenhuma superclasse específica. A classe listada a seguir foi elaborada de acordo com a especificação JAX-RPC:

```
public class SimpleServiceImpl implements SimpleService {

   public Pedido getPedido(int numero) throws RemoteException,
                                              PedidoNaoEncontrado {
      Pedido pedido = null;
      (...)
      return pedido;
  }

}
```

A partir dos arquivos de configuração que descrevem o serviço (*config.xml*), a ferramenta *wscompile* é utilizada para gerar os componentes necessários às aplicações cliente e servidora. Conforme os parâmetros utilizados, são gerados os *stubs*, *ties*[5], *serializers* e os arquivos WSDL que descrevem o serviço (ARMSTRONG et al., 2003). Como a implementação de um serviço remoto em JAX-RPC deve ser realizada com base nos modelos de componentes servidores previstos na especificação J2EE outra ferramenta, denominada *wsdeploy,* se encarrega de gerar as classes necessárias usando, para isso, as informações previamente detalhadas no arquivo *jaxrpc-ri.xml*. Uma vez implementado, o serviço é instalado em um servidor de aplicação que disponha das APIs servidoras JAX-RPC e vinculado a uma ou mais ligações (*bindings*) de protocolos (SUN, 2002c).

As aplicações cliente podem acessar os serviços através de três mecanismos distintos: *stubs*, uma interface de invocação dinâmica ou *proxies dinâmicos*. Os *stubs* são classes estáticas geradas pelas ferramentas JAX-RPC que implementam a interface prevista pelo serviço. Eles armazenam toda a lógica necessária para realizar a comunicação, podendo ser configurado para utilizar determinados protocolos. *Proxies* dinâmicos são similares aos *stubs*, porém são criados em tempo de execução a partir da interface do serviço. Finalmente, a invocação dinâmica utiliza a representação da chamada e de seus parâmetros, a partir da interface *javax.xml.rpc.Call*, para construir dinamicamente uma chamada e executá-la. Considerando a utilização de *stubs*, uma aplicação cliente seria similar ao código abaixo:

```
public void runTest() {
   SimpleService remoteObject = null;
   try {

      remoteObject = (SimpleService)
SimpleService_Impl().getHelloIFPort();

      Pedido resultado = remoteObject.getPedido();
```

[5] *Ties* são as classes *proxy,* servidoras equivalentes aos *skeletons* CORBA (SUN, 2002c)

```
    } catch (Exception e) {
       e.printStackTrace();
    }
}
```

Finalmente, deve-se ressaltar que, diferentemente dos mecanismos anteriormente abordados, JAX-RPC não disponibiliza um servidor de nomes (*naming*) através do qual as aplicações cliente possam localizar um serviço remoto (SUN, 2002c), delegando este processo às configurações envolvidas na chamada remota, no caso, aos parâmetros vinculados ao *stub* utilizado.

1.4 Conclusões

Observando as características de cada mecanismo apresentado neste capítulo, observa-se que, apesar da semelhança na forma de estruturar o problema de distribuição, cada qual possuí uma maneira proprietária de realizar tal tarefa. Implementar uma aplicação que contemple procedimentos como registrar um objeto remoto ao serviço de nomes, obter referências remotas, tratar exceções ocorridas durante a chamada de métodos, entre outras, implica na utilização de instruções específicas de cada mecanismo juntamente com o modelo de programação imposto pelos mesmos.

Tal constatação evidencia a amplitude das intervenções necessárias em ambos os lados da aplicação, cliente e servidor, diante da exigência de troca do mecanismo de distribuição, devido ao espalhamento das chamadas às APIs específicas nos pontos da aplicação que constituem ou referenciam um elemento remoto.

Capítulo 2

2 Reflexão computacional

Este capítulo apresenta os conceitos de reflexão computacional de forma a fundamentar sua utilização na solução no contexto deste trabalho. Iniciando com a definição apresentada na seção 3.1, o capítulo aborda as principais estratégias utilizadas para adicionar características reflexivas a linguagens de programação, seguindo-se da apresentação dos tipos de reflexão. A partir deste embasamento são apresentadas as soluções adotadas pela linguagem Java em sua especificação padrão e algumas iniciativas cujo objetivo é adicionar novas facilidades reflexivas a esta linguagem.

Complementarmente a seção 3.6 versa sobre a preocupação com a separação de *concerns* presente na engenharia de *software* e insere o uso da reflexão computacional em tal contexto.

2.1 Definição

Reflexão computacional é o comportamento apresentado por sistemas reflexivos quando estes realizam operações sobre si mesmos. Um sistema reflexivo possui, além dos dados que representam o domínio da aplicação, estruturas cujo objetivo é prover sua própria representação. O princípio de reflexão determina que um sistema deve manter uma representação causalmente conectada de seu próprio comportamento, a qual pode ser examinada e modificada pelo próprio sistema (MAES,1987). A conexão causal (*causal connection*) estabelece um relacionamento em duas direções entre a representação e o comportamento por ela descrito, de tal forma que o comportamento é controlado através da manipulação da representação e esta é constantemente atualizada de acordo com o comportamento do sistema (SMITH, 1984). Ou seja, sistemas reflexivos podem inspecionar sua representação (*introspection*) de maneira a avaliar seu próprio comportamento e, a partir desta avaliação, realizar modificações em sua representação (*intercession*) visando um comportamento diferenciado (TATSUBORI, 1999).

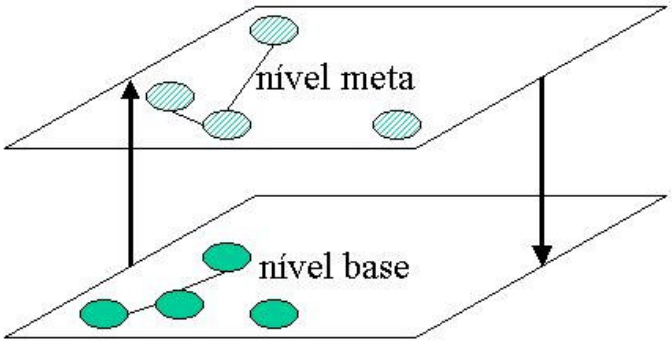
2.2 Arquitetura reflexiva

Uma arquitetura reflexiva é representada pelos mecanismos e ferramentas disponibilizados por um sistema computacional com o objetivo de manusear suas características reflexivas explicitamente. Em uma linguagem de programação, isto implica que os dados utilizados na representação do sistema devem estar acessíveis a qualquer programa que esteja sendo executado. Simultaneamente, os mecanismos da linguagem devem ser capazes de manter a conexão causal entre os dados e os aspectos do sistema que eles representam (MAES,1987).

Sistemas que apresentam tal arquitetura são decompostos em pelo menos dois níveis. O primeiro nível é o **nível base** (*base level*), onde se encontram as instruções e estruturas de dados relativas à manipulação e processamento das funcionalidades relacionadas à atividade fim, ou domínio, da aplicação. Por sua vez o **nível reflexivo**, também denominado de **nível meta** (DOURISH, 1996), é constituído dos dados e código voltados à representação da própria computação e das entidades utilizadas pelo nível base. Tais dados são gerados através do processo de **reificação** (*reification*), que extrai as informações implícitas do nível base de forma a torná-las disponíveis a nível meta.

A separação entre os níveis permite que alterações no nível meta sejam realizadas sem a necessidade de manipular diretamente o nível base. Desta forma, comportamentos não previstos inicialmente no código de nível base podem ser adicionados. A figura 2 demonstra a separação dos níveis em uma arquitetura reflexiva.

Figura 2 Modelo de uma arquitetura reflexiva



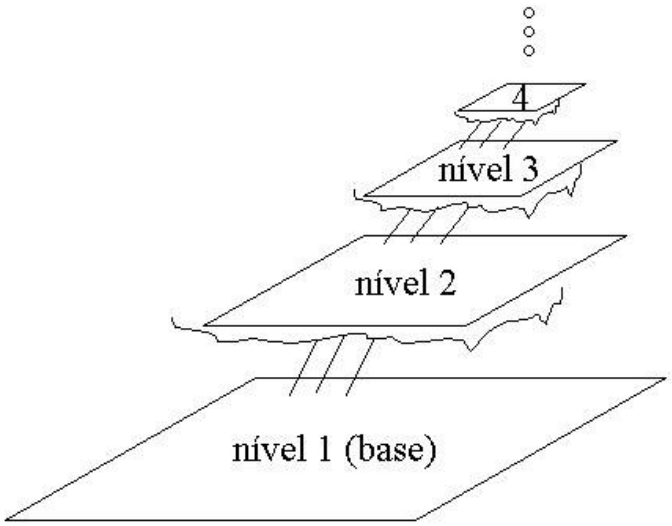
Fonte: Elaboração própria

Embora arquiteturas reflexivas possam ser projetadas e implementadas de maneiras distintas alguns requisitos básicos devem ser atendidos. FERBER (1989) menciona que toda arquitetura reflexiva deve definir quais entidades do nível base serão reificadas, os mecanismos através do qual o nível meta pode ser acessado e manipulado, e os meios através dos quais a conexão causal será garantida. Cabe, portanto, à arquitetura reflexiva determinar a abrangência das características reflexivas que estarão disponíveis. A seguir serão apresentadas duas abordagens utilizadas para implementar arquiteturas reflexivas.

2.2.1 Interpretadores meta-circulares

Um interpretador meta-circular é um programa escrito na mesma linguagem que ele interpreta, com a adição de características reflexivas que possibilitam a alteração do estado do interpretador durante sua execução (SOBEL; FRIENDMAN, 1996). Tais características, quanto utilizadas, não são executadas no mesmo nível da aplicação que realizou a chamada, mas sim um nível acima, ou seja, no nível do interpretador que está executando o programa que realizou a invocação. Desta forma, um programa de usuário (nível 0) é interpretado por um programa no nível acima (nível 1) que, por sua vez, é interpretado por outro programa no nível 2, e assim sucessivamente, constituindo uma torre de interpretadores (SMITH, 1983). A figura 3 representa este conceito:

Figura 3 Torre de interpretadores meta-circulares



Fonte: (SMITH, 1983)

O uso de interpretadores meta-circulares garante automaticamente a consistência entre os níveis da arquitetura, uma vez que a representação contida no nível meta é utilizada pra implementar o próprio sistema, não havendo possibilidade de ocorrer uma falha na sincronização entre ambas. Contudo, como a representação deve servir a dois propósitos simultaneamente, torna-se difícil equacionar a necessidade de uma estrutura que represente de maneira adequada as informações necessárias ao nível meta e, ao mesmo tempo, garanta uma execução eficiente da aplicação. Este inconveniente é acentuado devido à necessidade de existir uma representação unificada de programas e dados, permitindo que os programas possam ser manipulados da mesma maneira que estruturas de dados em uma linguagem de programação (MAES,1987). Uma técnica utilizada pela linguagem 3-Lisp para minimizar a necessidade de reificar todas as entidades de um determinado nível de programa é a utilização da reificação tardia, através do qual as representações são criadas no momento imediatamente anterior à execução da chamada reflexiva que realizará a mudança para o nível superior (DOURISH, 1996).

MAES (1987) denomina *procedural reflection* as características reflexivas implementadas a partir de uma torre de interpretadores meta-circulares. Isto não implica, contudo, que tal técnica esteja vinculada a linguagens procedurais. A seguir estão relacionadas outras linguagens baseadas em interpretadores meta-circulares:

i. Linguagens baseadas em regras: TEIRESIAS E SOAR;
ii. Linguagens baseadas em lógica: FOL, META-PROLOG;
iii. Linguagens funcionais: 3-LISP, BROWN.

2.2.2 Protocolos de meta-objetos

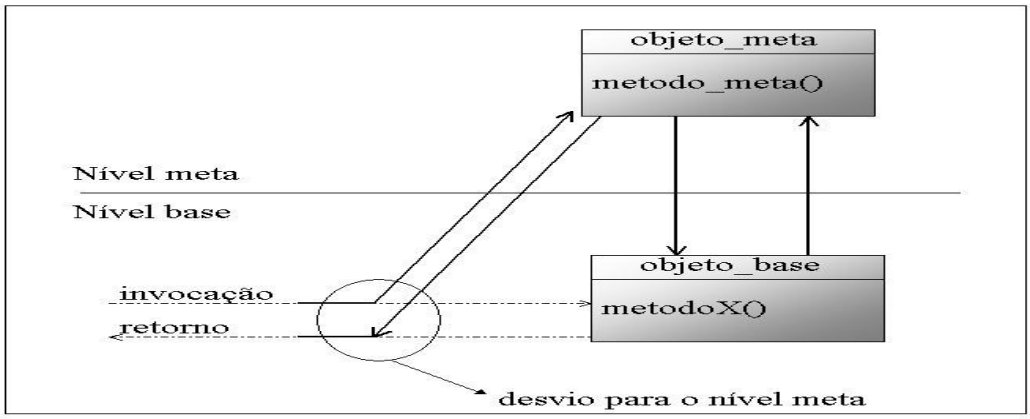
Protocolo de meta-objetos (*Metaobjects Protocols*-MOP) é uma técnica alternativa ao interpretadores meta-circulares para implementação de sistemas reflexivos que utiliza uma abordagem baseada na orientação a objetos (DOURISH, 1996). 3-KRS (MAES,1987) foi uma das primeiras linguagens a adotar uma arquitetura reflexiva baseada em objetos. Como um de seus objetivos era implementar a reflexão com funcionalidades semelhantes àquelas apresentadas pelos interpretadores meta-circulares, sua definição do nível meta tornou-se bastante completa:

i. Para cada objeto de nível base, existe um objeto de nível meta que o representa, contendo informações a respeito da herança, dos construtores disponíveis, etc.
ii. Todo elemento da linguagem faz parte da auto-representação definida. Possuem uma meta-representação todas as instâncias, classes, métodos, mensagens além dos próprios meta-objetos.
iii. Cada meta-objeto contém todas as informações relativas ao objeto por ele representado, inclusive aquelas utilizadas pelo próprio interpretador da linguagem.
iv. Como forma de garantir a conexão causal, a auto-representação do interpretador definida por 3-KRS é utilizada na implementação do sistema, de forma que toda operação efetuada sobre um objeto é repassada ao meta-objeto correspondente.
v. A auto-representação da linguagem é passível de modificações em tempo de execução, de forma a permitir a execução do comportamento modificado imediatamente.

É importante ressaltar que como um meta-objeto também é um objeto, ele também pode possuir outro meta-objeto que o descreva possibilitando, desta maneira, uma regressão virtualmente infinita que transpõe o conceito da torre de interpretadores meta-circulares para linguagens orientadas a objetos (FERBER, 1989; CHIBA, 1995; MENDHEKAR; FRIENDMAN, 1993) .

As características reflexivas disponibilizadas por uma linguagem estão diretamente relacionadas ao nível de aspectos representados pelos meta-objetos de sua arquitetura. FRIENDMAN afirma que, geralmente, os protocolos de meta-objetos limitam sua representação aos objetos existentes em um sistema. Desta forma, as facilidades reflexivas disponíveis se restringem à possibilidade de realizar a manipulação dos dados das instâncias existentes no domínio da aplicação (SOBEL; FRIENDMAN, 1996). Uma arquitetura de meta objetos mais ampla, como a apresentada em 3-KRS, permite que conceitos presentes somente no nível meta, como o fluxo de controle da aplicação e outras propriedades computacionais. A figura 4 apresenta um modelo de reflexão no qual meta-objetos são utilizados para representar os objetos-base:

Figura 4 Reflexão via meta-objetos



Fonte: Elaboração própria

Em sistemas orientados a objetos, um programa pode especializar o comportamento dos objetos de determinada classe através do mecanismo de herança. O programa que utiliza tais objetos passa a executar seus métodos sem o conhecimento de qual classe os implementa, deixando a cargo dos mecanismos da linguagem a definição de qual classe na hierarquia implementa efetivamente a operação (SOBEL; FRIENDMAN, 1996). Similarmente, um MOP estabelece os mecanismos orientados a objetos através dos quais as classes-meta podem ser acessadas e estendidas de forma a implementar um novo comportamento desejado para determinados tipos de entidades, possibilitando assim a adição de funcionalidades não proporcionadas pela linguagem original, mas exigidas por alguns usuários (KICZALES et al., 1993).

2.3 Tipos de Reflexão

Reflexão computacional pode ser classificada quanto ao tipo de características reflexivas oferecidas assim como quanto ao momento em que tais características podem ser acessadas. A seguir os tipos de reflexão serão brevemente apresentados.

2.3.1 Reflexão estrutural

A Reflexão estrutural (*structural reflection*) baseia-se na representação (reificação) dos elementos que constituem a aplicação, tais como a estrutura de classes, tipos de dados e herança, os quais podem ser consultados e utilizados pela própria aplicação(FERBER, 1989; TATSUBORI, 1999). Embora os termos reflexão estrutural e introspecção (*introspection*) sejam freqüentemente utilizados como sinônimos, CHIBA

(2000) os diferencia, apresentando o conceito de reflexão estrutural como sendo a possibilidade de alterar as estruturas de dados e o próprio código de um programa, possibilitando a modificação de estruturas estáticas quando necessário. Introspecção, portanto, refere-se somente à capacidade de uma aplicação inferir sobre sua estrutura, sem a possibilidade de realizar modificações.

2.3.2 Reflexão comportamental

Reflexão comportamental (*behavioral reflection*) (CHIBA, 2000) baseia-se no princípio de interceptação da execução dos programas. Isto é realizado através da adição de ganchos (*hooks*) em operações como chamadas de método, acesso a atributos e construção de novas instâncias nos programas de nível base. Quando uma destas operações é interceptada, o fluxo de controle da aplicação é desviado para um programa de nível meta que esteja associado através de um gancho. Desta maneira um programa de nível meta pode ser definido (ou modificado) para implementar o novo comportamento. Em seu estudo sobre reflexão em linguagens orientadas a objeto, FERBER (1989) aborda a reflexão comportamental (a qual denomina *computational reflection*) citando o modelo introduzido por MAES (1987) no qual para cada objeto existe um meta-objeto correspondente representando sua estrutura e a forma de tratamento dos métodos. Tais meta-objetos representam a "computação" do programa sendo executado e podem ser utilizados para modificar o comportamento do nível base.

2.3.3 Reflexão em tempo de execução (*runtime*)

São arquiteturas nas quais os elementos do nível meta estão presentes durante o tempo de execução da aplicação, concorrendo com o código de nível base pela capacidade de processamento do equipamento. Os interpretadores meta-circulares (SMITH, 1984) e a arquitetura reflexiva da linguagem 3-KRS (MAES, 1987) são exemplos deste tipo de reflexão.

2.3.4 Reflexão em tempo de compilação

Reflexão em tempo de compilação é uma abordagem fundamentada principalmente na necessidade de impor a menor sobrecarga possível às aplicações, na

qual as estruturas que representam as características reflexivas existem somente em tempo de compilação. Tais estruturas são utilizadas para controlar a compilação do programa de modo a traduzir definições em alto nível do programa, adicionando, se necessário, novos elementos como tipos de dados e funções que constituirão o programa final. Além de propiciar uma performance superior a arquiteturas que mantenham um nível meta em tempo de execução, este modelo de reflexão, quando baseado na modificação do código fonte, permite que o compilador da linguagem seja utilizado sem modificações, garantindo assim que as otimizações previstas sejam normalmente realizadas (CHIBA, 1995).

2.3.5 Reflexão em tempo de carregamento (*loadtime*)

Loadtime reflection é uma abordagem que possibilita a reflexão no momento em que o código compilado é carregado pelo mecanismo de execução da linguagem. Tal implementação realiza, na realidade, uma modificação do código compilado antes que este seja efetivamente executado. Desta forma não se faz necessário realizar modificações no compilador e no mecanismo de execução, permitindo que características reflexivas sejam adicionadas sem implicações nos padrões da linguagem. Esta abordagem possui, contudo, algumas limitações, como a impossibilidade de realizar modificações estruturais em uma aplicação após o seu carregamento (CHIBA, 2000) bem como o acréscimo de tempo necessário para realizar as modificações durante o carregamento (TATSUBORI, 1999).

2.4 Reflexão na linguagem Java

2.4.1 Reflection API

A API de Reflexão Java reúne um conjunto de classes que possibilitam a representação de elementos da linguagem, como classes e objetos, bem como o acesso a tais elementos via introspecção. Sua introdução ocorreu a partir da versão 1.1 do JDK, como forma de facilitar o desenvolvimento de outras características da plataforma, como o modelo de componentes JavaBeans e as características de serialização (SUN, 1998). Esta API implementa um modelo de reflexão em tempo de execução, que disponibiliza somente propriedades de introspecção sobre os elementos estruturais

representados (ver seção 2.3). CHIBA (2000) ressalta que esta API não oferece capacidades reflexivas completas, por não disponibilizar mecanismos para intervir no comportamento e na estrutura das aplicações.

A utilização da API de reflexão está sujeita, como toda operação Java, às validações de segurança previstas na JVM. Caso tais premissas de segurança sejam atendidas, a API pode ser utilizada para (SUN, 1998):

i. Construir novas instâncias e novos *arrays*;
ii. Acessar e modificar campos de objetos e classes;
iii. Realizar invocações de métodos em objetos e classes;
iv. Acessar e modificar elementos de *arrays*.

Tais operações são realizadas através de classes especificamente projetadas para representar elementos da linguagem. Dentre estas, destacam-se:

i. *java.reflect.Method*: representa um determinado método recuperado a partir de uma classe. Pode ser utilizado para realizar a invocação dinamicamente incluindo a utilização de parâmetros;
ii. *java.reflect.Field*: representa um atributo de uma determinada classe e oferece os meios para obter e atribuir valores;
iii. *java.reflect.Constructor*: similar a *Method*, porém representando um construtor da classe;

O modelo de programação imposto pela API impede que tais classes sejam diretamente criadas pelo programa que deseja utilizar a reflexão. Ao invés disto, as instâncias destas classes são inicializadas a partir de outra classe, a *java.lang.Class* que atua como um *Factory* para instanciar os demais mecanismos reflexivos, seja a partir de uma classe carregada dinamicamente ou de um objeto previamente alocado. É importante ressaltar que toda classe Java é uma especialização implícita da classe *java.lang.Object,* a qual implementa o método denominado *getClass*(). Isto garante que qualquer classe prevista na especificação da linguagem bem como as classes definidas pelos programadores possuem a funcionalidade necessária para recuperar sua representação reflexiva, facilitando assim o processo de introspecção.

Apesar destas facilidades, a API proporciona uma capacidade de extensão bastante limitada. Além de não oferecer meios para intervir no comportamento da aplicação, a API de Reflexão também limita a modificação de seu modelo ao definir

suas principais classes (*Field, Method* e *Constructor*) como classes finais, ou seja, classes que não podem ser especializadas através do mecanismo de herança (SUN, 1998).

2.4.2 Dynamic Proxy API

O termo *proxy* é usado em diversas áreas da computação. Os exemplos mais típicos de uso de *proxies* são aqueles utilizados como parte dos mecanismos de segurança baseados em *firewalls* (LODIN; SCHUBA, 1998) e os *proxies* usados como suporte à *World Wide Web* (WWW) (LUOTONEN; ALTIS, 1994). Em geral, um *proxy* pode ser considerado como um elemento intermediário que atua no lugar de um componente, interage com outros componentes (um servidor, por exemplo), processa os dados envolvidos na interação e, posteriormente, devolve os dados processados à aplicação cliente (SEITZ et al., 1998). Desta forma, um *proxy* aplicado em uma arquitetura de *firewall* pode determinar se os dados estão em conformidade com os padrões de segurança previamente definidos, e decidir se o acesso será permitido ou não. No exemplo da WWW, o *proxy* pode filtrar ou comprimir os dados de forma a economizar a capacidade de transmissão de rede, ou ainda manter as páginas acessadas em um *cache* de forma a reduzir seu tempo de acesso.

Desde sua versão 1.3, a linguagem Java disponibiliza um conjunto de classes no pacote de reflexão que permite a geração de classes *proxy* dinâmicas. Tais classes constituem a *Dynamic Proxy* API (SUN, 1999). Um *proxy* dinâmico é uma instância de uma classe que implementa um conjunto de interfaces especificado em tempo de execução. Sua principal função é interceptar as invocações aos métodos previstos nestas interfaces, representá-los de acordo com as características reflexivas da linguagem, e repassa-los para outro objeto através de uma interface definida (HARPIN, 2001). Este objeto é uma implementação da interface *java.lang.reflect.InvocationHandler* e os *proxies* são vinculados a tais objetos no momento que seu construtor é executado. Todo o processo é gerenciado pela classe *java.lang.Proxy*, que disponibiliza métodos estáticos como *newInstance*, cuja função é recuperar uma instância da classe *proxy* criada a partir da lista de interfaces.

As classes *proxy* são definidas no contexto do *classloader* informado no momento de sua geração. A partir disso, o método *Proxy.getProxyClass* retorna uma nova classe ou uma classe existente, caso exista uma classe previamente gerada com tais interfaces.

LOPES (1997) ressalta que uma das principais restrições que um *proxy* deve atender é a verificação de tipos, ou seja, um *proxy* deveria ser do mesmo tipo (ou classe) que o objeto por ele representado. Desta forma a aplicação cliente não precisaria ter consciência de sua utilização, pois as verificações de tipo previamente codificadas seriam aceitas normalmente. Em Java (GOSLING et al., 2000), as verificações de tipo são executadas de duas maneiras: através do mecanismo de coerção (*cast*) e do operador *instanceof*. A coerção é um procedimento utilizado para forçar a conversão de um determinado objeto para uma referência de uma classe determinada. Caso a conversão seja inválida, a exceção *ClassCastException* é gerada. O exemplo abaixo demonstra a tentativa de converter o objeto *obj* em uma referência da classe *Cliente*:

```
Cliente cliente_01 = (Cliente) obj;
```

Por sua vez, a instrução *instanceof* realiza a verificação do objeto em face de uma determinada classe. O exemplo abaixo avalia se o objeto *obj* é uma instância da classe cliente:

```
if (obj instanceof Cliente) { ... }
```

Ambas as verificações de tipo executadas sobre um *proxy* gerado dinamicamente são válidas, desde que a classe utilizada no procedimento seja uma das interfaces informadas no momento da inicialização da instância.

2.5 Estendendo a reflexão de Java

As seções anteriores demonstraram que as características reflexivas da linguagem Java se limitam à possibilidade de introspecção dos elementos reificados pela API de Reflexão (classes, objetos, atributos, construtores e métodos) e a uma limitada reflexão comportamental obtida através do uso de classes *proxy* dinâmicas. Conforme ressalta CHIBA (2000), as limitadas possibilidades disponíveis na especificação padrão de Java levou ao surgimento de uma série de propostas visando reduzir tais limitações. O autor ressalta ainda que muitas destas iniciativas se concentram em prover uma melhor reflexão comportamental, através de um protocolo de meta objetos que ofereça maiores possibilidades de customização pelos programadores.

Esta seção visa apresentar algumas destas iniciativas, especialmente para ressaltar as diferentes abordagens utilizadas: compilação, *runtime* e *loadtime*, com ou sem modificação da linguagem e da máquina virtual.

2.5.1 OpenJava

OpenJava é uma extensão da linguagem Java projetada para oferecer características de reflexão comportamental e estrutural. Um de seus objetivos é aplicar as características reflexivas, sempre que possível, em tempo de compilação, evitando assim a perda de performance inerente às abordagens em tempo de execução. Sua principal característica é a utilização de programas meta escritos em uma extensão da linguagem Java, que contém os metaobjetos que representam os elementos textuais e semânticos da linguagem original (TATSUBORI et al., 2000).

O processo de compilação de OpenJava é efetuado por dois componentes principais: um tradutor e um compilador Java regular. Em um primeiro momento, o programa OpenJava é submetido ao tradutor que realiza a geração do código fonte Java padrão acrescido das características especificadas pelos meta-objetos presentes na linguagem estendida. Este código é, então, repassado ao compilador Java, que gera *bytecodes* padrões, os quais podem ser executados em qualquer JVM. Complementarmente, o compilador OpenJava se utiliza das bibliotecas de nível meta, além das bibliotecas padrão (TATSUBORI, 1999).

Uma restrição da abordagem baseada em compilação adotada por OpenJava é a necessidade do código fonte de todos os programas sobre os quais se deseja aplicar as características de reflexão, impedindo que programas distribuídos em formato compilado sejam manipulados. Isto não obriga, todavia, que o código fonte esteja presente em tempo de execução, pois as informações relativas ao meta nível são mantidas em uma classe especial, cujo propósito é permitir o acesso a tais informações pela aplicação (TATSUBORI et al., 2000).

2.5.2 Guaraná

Guaraná é uma arquitetura reflexiva baseada em um núcleo independente de linguagem de programação, elaborada com o objetivo de proporcionar reflexão estrutural e comportamental de uma maneira simples, flexível e segura. Seu núcleo é

responsável por implementar os mecanismos de reificação, interceptação das mensagens, manutenção da meta-informação estrutural e a invocação dinâmica aos meta-objetos e atua sobre os objetos base e meta em tempo de execução (OLIVA; GARCIA; BUZATO, 1998).

A arquitetura de meta-objetos de Guaraná foi concebida de forma a associar um objeto de nível-base com zero ou um meta-objeto. Este meta-objeto se encarrega de observar as operações realizadas sobre o objeto de nível base correspondente através dos serviços de reificação e inteceptação providos pelo núcleo da arquitetura. Múltiplos meta-objetos podem ser combinados através de meta-objetos especiais, denominados *composers*, constituindo grupos de meta-objetos co-relacionados. Este mecanismo possibilita um maior reuso da lógica implementada no nível meta, bem como a associação indireta de um maior número de meta-objetos a um objeto de aplicação (OLIVA; GARCIA; BUZATO, 1998).

Uma implementação de Guaraná foi realizada a partir da modificação de uma máquina virtual Java aberta sem recorrer, todavia, à alterações na linguagem (OLIVA; GARCIA; BUZATO, 1998).

2.5.3 Javassist

Javassist é uma biblioteca de classes Java desenvolvida com o objetivo de possibilitar a reflexão estrutural em tempo de carregamento. Sua principal diferença em relação à API de Reflexão Java é a capacidade de realizar, além do processo normal de introspecção, alterações nas definições de classe, permitindo a inclusão de novos métodos, atributos, alterações na hierarquia e nos modificadores presentes na declaração de classes (*public, private, etc*). Os principais objetivos da ferramenta são: abstrair do programador o código fonte da aplicação, implementar a reflexão estrutural com a melhor performance possível e assegurar a compatibilidade de tipos (CHIBA, 2000).

Ao contrário de OpenJava, Javassist não se utiliza de extensões da linguagem em sua implementação. Sua abordagem é baseada na modificação de *bytecodes* Java. Neste modelo o programador determina, através da API provida pela ferramenta, qual classe deve ser modificada e, a partir destas definições, as alterações necessárias são aplicadas no momento de carregamento da classe. Desta forma, cria-se uma abstração em relação

ao código fonte, uma vez que este não é necessário durante a aplicação da reflexão estrutural (CHIBA, 2000).

O processo de adição de métodos a classes existentes ressalta a preocupação com a performance. Ao invés de permitir a definição e compilação do código fonte durante o carregamento da classe, a ferramenta restringe a constituição do corpo dos novos métodos a uma cópia de métodos existentes em outras classes previamente compiladas. Embora a performance deste mecanismo seja superior à compilação, este procedimento traz uma certa limitação na definição de novos métodos, os quais não podem ser livremente definidos (CHIBA, 2000).

Por se tratar de uma biblioteca, Javassist pode ser utilizado a partir de qualquer classe, possibilitando sua aplicação em outros cenários além do *Classloader* como servidores Web e de aplicação (CHIBA, 2000).

2.6 Separação de concerns

Separação de *concerns* é a denominação para uma prática amplamente utilizada na engenharia de software e nas linguagens de programação: o processo de adicionar níveis de abstração cada vez maiores conforme o crescimento da complexidade do software (LOPES; HÜRSCH, 1995; CHU-CARROL, 2000). De uma forma geral, separação de *concerns* refere-se ao processo de analisar, encapsular e manipular determinadas partes de um software que sejam responsáveis pela execução de uma atividade ou tarefa específica (TARR et al., 2000). Analisando esta última afirmação, nota-se uma grande similaridade com estudos realizados sobre modularização de aplicações, especialmente quando o critério adotado para sua identificação segue os pressupostos de PARNAS (1972), segundo o qual um módulo não deve ser encarado como uma sub-rotina, mas sim como um elemento que assume determinadas responsabilidades.

Durante os últimos anos novas técnicas foram desenvolvidas para proporcionar uma maior nível de abstração, tais como a programação estruturada e o desenvolvimento orientado a objetos. O problema é que a complexidade inerente às aplicações cresceu de tal forma que as características disponibilizadas por tais técnicas não se mostraram suficientes para atingir o grau e o tipo de modularidade exigidos. Os atuais paradigmas de desenvolvimento lidam de maneira adequada com certos tipos de

concerns, como o encapsulamento dos dados na orientação a objetos e as funções em abordagens funcionais, levando à "*tirania da decomposição dominante*". Este problema ocorre quando uma determinada estrutura (de classes, por exemplo) é utilizada para decompor as características funcionais de uma aplicação que, por sua vez, compartilham uma determinada característica não-funcional (um método de impressão). Embora todos os componentes funcionem adequadamente, não existe uma estrutura que centralize a característica não-funcional, de forma que qualquer alteração deve ser realizada em diversos pontos da aplicação (TARR et al., 2000).

A situação descreve perfeitamente os conceitos detalhados por LOPES & HÜRSCH (1995), que ressalta a necessidade de tratar corretamente os *concerns* de determinado software a nível conceitual e de implementação. O nível conceitual, que engloba as definições e a identificação de cada *concern*, é relativamente bem abordado pelas atuais metodologias de engenharia de software. O mesmo não ocorre com a implementação, onde as abstrações relacionadas no nível conceitual são mapeadas para os elementos de uma linguagem de programação: devido à falta de elementos de linguagem adequados para separar corretamente os *concerns*, ocorre uma tentativa de tratar diversos *concerns* em um mesmo local, gerando um código monolítico. Esta incapacidade de representar os *concerns* em artefatos de software apropriados faz com que o código da nova característica fique espalhado (*scattered*) por diversos lugares e misturado (*tangled*) com instruções relativas a outros *concerns* e, principalmente, com o código funcional da aplicação (LOPES; HÜRSCH, 1995).

A dificuldade em tratar eficazmente a separação de *concerns* compromente os principais objetivos desejados com a modularização de uma aplicação, pois:

i. Desenvolver uma aplicação com código espalhado é difícil, pois todos os *concerns* envolvidos estão sendo manipulados no mesmo nível e ao mesmo tempo;
ii. O código misturado é difícil de ser mantido e modificado, pois existe um acoplamento forte entre os *concerns* por ele representado;
iii. O código espalhado desencadeia uma anomalia de herança, pois torna-se difícil redefinir um método em uma subclasse quando a superclasse possui tal código em seu método.

2.6.1 Reflexão e separação de concerns

A separação entre os níveis meta e base proporcionada pelas arquiteturas reflexivas constitui um modelo no qual a separação de *concerns* se encaixa perfeitamente. Segundo MAES (1987), enquanto os elementos de nível base se encarregam das tarefas ligadas à computação do domínio da aplicação, os elementos meta podem ser livremente utilizados para implementar as funções de apoio (ou características não-funcionais). A autora ressalta ainda que as linguagens de programação deveriam propiciar características reflexivas que possibilitassem aos desenvolvedores adicionar e remover determinados comportamentos que sejam necessários somente em algum momento como, por exemplo, depuração e log.

LOPES & HÜRSCH (1995) reafirmam esta adequação, ressaltando que ao controlar o envio e recebimento de mensagens aos objetos de nível base, os meta objetos podem realizar operações voltadas à implementação de um determinado *concern*. Todavia como os objetos-meta são mecanismos genéricos, a questão de como compor diferentes concerns de uma forma modular ainda persiste. Ou seja, embora os elementos de nível meta não tenham mais referências ao código que implementa os *concerns*, este mesmo código passa a existir no nível meta com os mesmos problemas anteriormente destacados (código espalhado e misturado).

2.7 Conclusões

Ao promover a separação entre os níveis meta e base de uma aplicação, a reflexão computacional propicia uma maior flexibilidade para as aplicações. Com base neste princípio, procedimentos voltados à execução de tarefas não-funcionais podem ser adequadamente isoladas do código funcional da aplicação, constituindo um caminho eficaz para a introdução de novos elementos que não tenham sido inicialmente previstos na concepção do *software*.

Todavia a falta de um padrão para as facilidades reflexivas providas pelas linguagens, como pode ser observado nas diferentes estratégias de implementação das arquiteturas e facilidades reflexivas, delega ao projetista da aplicação a decisão sobre qual abordagem se aplica adequadamente para o problema (e linguagem) em questão.

Finalmente constata-se que embora as características reflexivas de Java sejam limitadas, fato que fomenta a pesquisa de extensões ao padrão, a possibilidade de

realizar a interceptação de métodos através de *proxies* dinâmicos e de utilizar a introspecção para manipular objetos mostra-se promissora para o problema apresentado nesta dissertação.

Capítulo 3

3 Framework para distribuição

3.1 Introdução

O principal objetivo do *framework* aqui apresentado é separar os aspectos relativos ao mecanismo de distribuição utilizados na aplicação cliente. Tais mecanismos, quando utilizados da maneira tradicional, acabam entremeando suas instruções ao código fonte relativo à lógica de negócio, tornando a aplicação mais confusa e difícil de manter. Esta situação pode ser constatada nos modelos de programação apresentados no capítulo sobre mecanismos de distribuição.

Cada mecanismo disponibiliza o seu próprio meio de realizar as tarefas necessárias para acessar os componentes remotos. Em todas elas é necessário, em primeiro lugar, obter uma referência ao serviço de nomes no qual foi registrada a instância do serviço a ser acessado. Previamente, cada classe da aplicação que precisar interagir com um serviço remoto deve, explicitamente, realizar a importação das bibliotecas de classes que representam o mecanismo. Uma vez que a referência ao serviço de nomes tenha sido obtida, faz-se necessário realizar a localização do serviço específico, em um processo denominado *lookup.*

Pretende-se, com a utilização do *framework* aqui proposto, permitir a abstração dos mecanismos de distribuição de forma que o desenvolvedor da aplicação cliente não tenha a necessidade de lidar diretamente com tais APIs. Adicionalmente este encapsulamento deve possibilitar a troca do mecanismo de distribuição com impacto nulo nas aplicações previamente elaboradas.

Para atingir tais objetivos, o projeto e implementação deste *framework* foram baseados nas características de reflexão padrão da linguagem Java, ou seja, a capacidade de introspecção provida pela API de Reflexão, e as características comportamentais da API de Proxies Dinâmicos. O objetivo desta escolha é demonstrar, a princípio, que tais funcionalidades são suficientes para solucionar o problema de distribuição aqui apresentado, bem como avaliar se o impacto gerado pela reflexão em tempo de execução inviabilizaria sua utilização em tal contexto.

3.2 Resumo dos design patterns adotados

O objetivo desta seção é apresentar os padrões de projeto utilizados no projeto do *framework* proposto. Padrões de projeto (*Design Patterns*) (GAMMA et al., 1995) são descrições textuais de uma solução comprovadamente aplicável a determinado problema de projeto. O documento que descreve um padrão de projeto enfatiza o contexto no qual a solução deve ser aplicada, caracterizando o problema e descrevendo as conseqüências e o impacto da solução. BOOCH et al. (2000) enfatiza a relação entre padrões e *frameworks*:

> *"A pattern is a common solution to a common problem in a given context. A mechanism is a design pattern that applies to a society of classes. A framework is an architectural pattern that provides an extensible template for applications within a domain."*

3.2.1 Proxy

Proxy (GAMMA et al., 1995) é um padrão de projeto estrutural que define objetos cuja função é representar determinado serviço. Objetos *proxy* disponibilizam uma interface idêntica àquela existente no serviço por ele representado, adicionando um nível de "indireção" que é utilizado de forma a possibilitar a execução de outras operações antes ou depois da execução do serviço efetivo, controlando, desta maneira, a forma de acesso ao serviço.

3.2.2 Factory Method

Factory Method (GAMMA et al., 1995) é um padrão que proporciona uma interface a partir da qual os objetos podem ser criados, e delega às subclasses a decisão sobre qual classe deve ser instanciada. É um utilizado especialmente quando uma classe não pode determinar qual subclasse deve ser utilizada na obtenção da instância, podendo tomar esta decisão tardiamente através de uma parametrização.

3.2.3 Singletron

Singletron (GAMMA et al., 1995) é um padrão de criação que visa garantir a existência de somente uma instância de determinada classe, a qual pode ser globalmente acessada. Este padrão é uma alternativa à definição de variáveis globais, que não impedem a criação de múltiplas instâncias. Classes *singletron* geralmente implementam

serviços genéricos que devem ser acessados de qualquer ponto da aplicação, como uma fila de impressão ou um gerenciador de janelas.

3.2.4 Façade

Façade (GAMMA et al., 1995) é um padrão estrutural que visa simplificar a utilização de um subsistema através da definição de uma interface unificada. A aplicação deste padrão reduz o acoplamento entre as camadas da aplicação ao reduzir o número de elementos com os quais o cliente da interface deve lidar. Uma dos motivos que leva à utilização deste padrão é a necessidade de dividir a aplicação em camadas, tornando a implementação de uma camada inferior abstrata para as camadas superiores.

3.2.5 Front Controller

Front Controller (ALUR; CRUPI; MALKS, 2001) é um padrão voltado ao desenvolvimento em aplicações Web nas quais solicitações dos usuários devem ser coordenadas e processadas. Este padrão propõe a definição de um elemento frontal que centralize todas as operações solicitadas pelos clientes, facilitando, desta maneira, a realização de outros procedimentos como a aplicação de políticas de segurança e rastreamento das operações realizadas (*log*). Como poderá ser visto nas seções seguintes, este padrão influenciou de maneira evidente a arquitetura da solução apresentada.

3.3 Características do framework

Esta seção apresenta os principais itens considerados durante a elaboração dos elementos que constituem o *framework*, fornecendo uma visão geral de como as decisões de projeto foram tomadas.

3.3.1 Interceptação de chamadas

Conforme previamente discutido, a interceptação de chamadas a métodos é um dos princípios básicos da reflexão comportamental. No contexto deste framework, a chamada de método representa o gatilho que dispara toda a interação entre o cliente e os componentes responsáveis pelo encapsulamento do mecanismo de distribuição. A adoção desta abordagem possibilita que o modo de programação da aplicação cliente

não seja modificado em função do uso do framework, permitindo ao programador utilizar os objetos que representam os serviços do mesmo modo que objetos Java locais exceto, evidentemente, pelo tratamento das exceções reportadas pelas classes, de acordo com as possibilidades discutidas a seguir.

Esta abordagem incorre, todavia, em um processamento extra a cada chamada de métodos, pois, além da execução da operação desejada também é executado o código adicional para realizar a interceptação e, adicionalmente, a lógica implementada para servir a propósitos específicos, no caso deste estudo, a distribuição da aplicação.

3.3.2 Tratamento de exceções

Um framework que vise encapsular mecanismos de comunicação deve prever, em seu projeto, como as características específicas de tratamento de exceção deverão ser representadas. Durante esta pesquisa algumas possibilidades foram consideradas: a completa abstração da distribuição dos componentes, a criação de classes de exceção específicas do framework.

A abordagem baseada na abstração completa do mecanismo de distribuição visa esconder do programador da aplicação cliente qualquer classe ou biblioteca que possibilite a distinção entre um objeto sendo acessado na mesma máquina virtual ou naqueles acessados em JVMs separadas. Uma primeira tentativa de realizar tal abstração é a utilização de blocos de captura de exceções (*try/catch*) que tratem os eventos ocorridos de maneira silenciosa. Ou seja, uma vez que a exceção ocorra, o framework poderia optar em não reportá-la ao cliente. Esta opção gera, obviamente, um comportamento inadequado da aplicação ao tornar os erros “invisíveis” para a aplicação cliente e, fatalmente, torna a tarefa de identificar erros no aplicativo um trabalho muito difícil. Uma saída interessante para evitar estes problemas ao mesmo tempo em que a noção de uma aplicação local é mantida, pode-se realizar o mapeamento entre as exceções remotas e as exceções normalmente previstas pela especificação Java, como por exemplo, um erro de inicialização do mecanismo de distribuição ou de *lookup* do objeto poderia ser mapeado para a exceção *java.lang.ClassNotFoundException*, uma vez que a referência ao objeto remoto não poderá ser criada. Complementarmente a este mapeamento, um arquivo de contendo o registro (*log*) de todas as exceções capturadas

pelo framework pode ser gerado, de forma a possibilitar uma depuração mais detalhada das condições nas quais determinadas exceções ocorreram.

Outra abordagem possível se baseia na elaboração de uma hierarquia de classes de exceções específicas do framework, projetadas de forma a abranger todas as exceções previstas para cada mecanismo de distribuição para o qual um FRDTransporte tenha sido implementado. Cada exceção definida por esta hierarquia deve mapear as exceções correspondentes entre os mecanismos utilizados. A mesma falha de inicialização descrita acima poderia ser mapeada para uma exceção como *org.test.remote.FRDInitializationException*, e as falhas de localização para *org.test.remote.FRDLookupException*. Desta forma, as exceções poderiam ser normalmente reportadas à aplicação cliente com uma vantagem adicional: a exceção do framework pode ser definida de forma a agregar a exceção original como uma de suas variáveis de instância, permitindo ao usuário da aplicação identificar exatamente qual a origem da falha. De fato, este procedimento foi adotado pela especificação Java para a classe *java.lang.Throwable* a partir da versão 1.4 do JDK, que passou a conter um construtor específico que recebe, além de uma *string* com a mensagem correspondente ao erro, um outro objeto *Throwable* que identifica a causa da exceção. Como este objeto também pode conter uma causa, criou-se um meio de representar uma cadeia de exceções. Apesar desta disponibilidade, a opção por utilizar uma classe específica é fundamentada pela necessidade de utilizar o framework em versões anteriores do JDK, para quais tal implementação não se encontra disponível.

A decisão sobre qual modelo de manipulação de exceções deve ser adotado está diretamente ligada ao tipo de desenvolvimento previsto. Caso o esforço de desenvolvimento se baseie em novas aplicações, a abordagem de exceções de cobertura mostra-se mais interessante, uma vez que a forma de reportar os erros integralmente facilita a identificação e, conseqüentemente, a correção. Em aplicações existentes, onde o esforço de desenvolvimento se baseia principalmente no *refactoring* visando a separação das aplicações cliente e servidora, a abordagem de abstração de exceções pode poupar os desenvolvedores da codificação de blocos *try/catch* inexistentes na aplicação original.

3.3.3 Aplicação de interfaces

A programação baseada no uso de interfaces proporciona uma maior abstração e uma maior segurança com relação aos tipos de dados envolvidos em uma chamada (SZYPERSKI, 1999). A abstração decorre da garantia que o objeto sendo invocado implementa, obrigatoriamente, os métodos previstos na interface de modo que a aplicação que o utiliza não precisa conhecer a classe que implementa efetivamente os métodos. Isto facilita a criação de classes abstratas, mecanismos de cobertura e, no caso deste estudo, de classes *proxy*. De modo análogo, os tipos de dados definidos como parâmetros ou retorno de um método, bem como as exceções que tenham sido definidas na cláusula de *throws,* podem ser verificados em tempo de compilação, evitando problemas de coerção de tipos em tempo de execução.

Como forma de reforçar o contrato entre as aplicações cliente e servidora, a mesma interface utilizada pela classe que implementa o serviço remoto é utilizada pelo framework para gerar o *proxy* dinâmico que representa o serviço remoto adicionando, desta forma, uma garantia quanto à compatibilidade dos tipos de dados envolvidos na operação.

3.3.4 Abstração do mecanismo de distribuição

A arquitetura aqui proposta delega um papel central aos mecanismos de comunicação tanto pela sua importância no processo de localização, conexão e comunicação dos métodos remotos, quanto pelo seu efetivo posicionamento nas camadas do *framework*. Nesta arquitetura, camadas de abstração foram criadas de maneira a evitar que a aplicação cliente precise interagir com as APIs dos mecanismos de distribuição. Toda interação com estas interfaces é gerenciada internamente pelo *framework*, o que torna possível escolher o mecanismo que melhor se adapte ao problema em questão, sem maiores inconvenientes para o código cliente previamente desenvolvido.

3.3.5 Utilização da reflexão

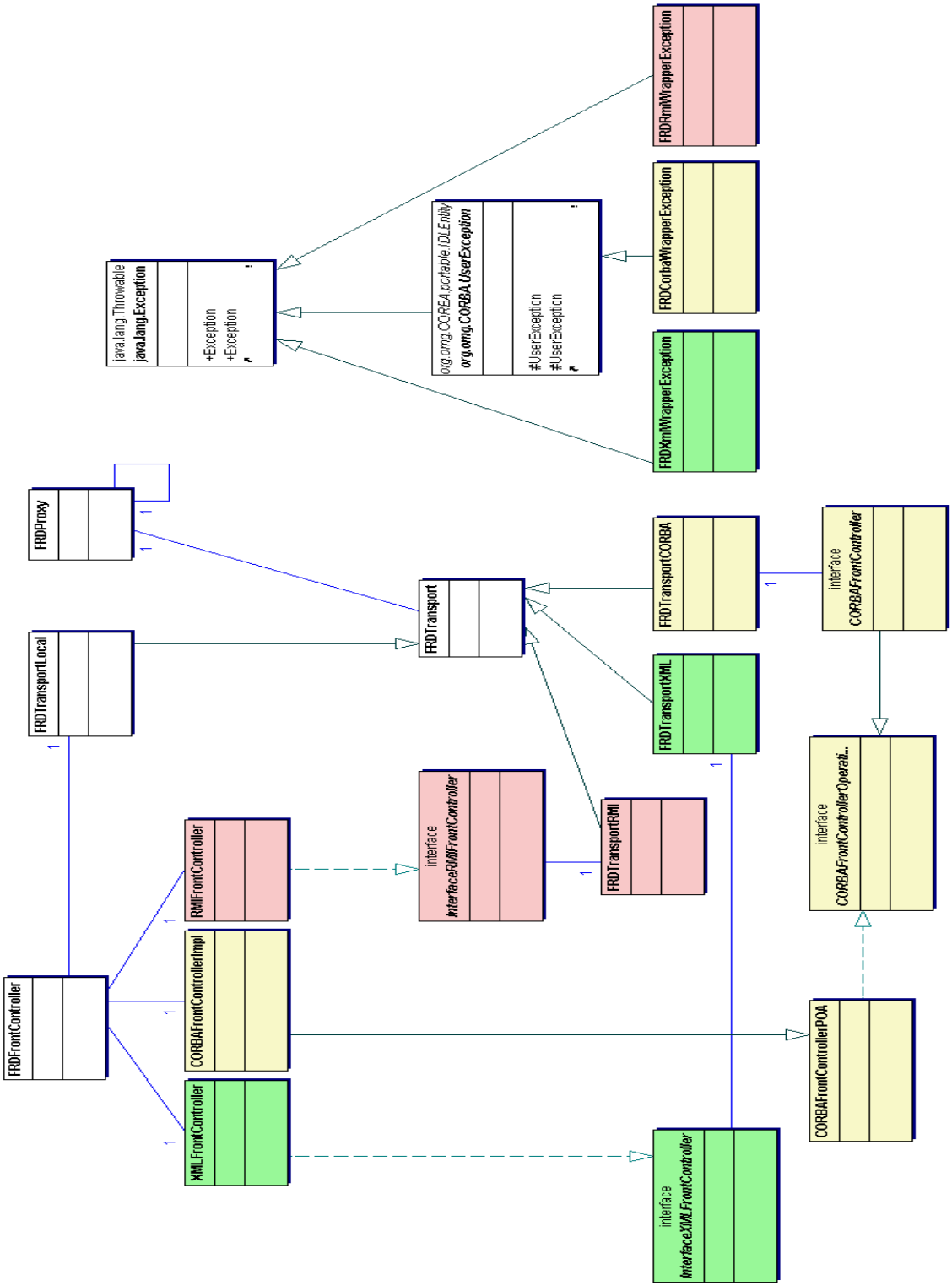
Embora o problema aqui apresentado seja relativamente simples de ser solucionado mediante aplicação do padrão de projeto *Proxy* (GAMMA et al., 1995), as

implementações deste padrão normalmente levam à codificação de uma classe extra (ou duas, quando combinado com o padrão *Adapter*) para cada classe remotamente acessada. Tendo em vista a necessidade de simplificar o desenvolvimento de aplicações distribuídas, buscou-se um meio de eliminar esta codificação. Conforme apresentado na seção 2.6, reflexão computacional possuí características que possibilitam a separação de *concerns* em um nível mais abstrato facilitando, desta forma, a concepção de uma arquitetura na qual certas codificações repetitivas possam ser eliminadas através do uso de componentes genéricos que manipulem as informações de nível meta. Esta abordagem foi utilizada na concepção de todos os componentes do *framework* que intermedeiam a chamada de um método pela aplicação cliente.

3.4 Arquitetura e implementação do framework

Com base nos elementos apresentados na seção 3.3 e nos padrões de projeto vistos em 3.2 o *framework* foi elaborado. As classes criadas para satisfazer os requisitos são apresentadas na figura 5:

Figura 5 Diagrama de classes do *framework*



Fonte: Elaboração própria

Este diagrama de classes apresenta todos os componentes da solução e os divide com base no mecanismo de transporte ao qual cada classe está relacionada. As classes verdes compõem a implementação voltada ao encapsulamento das chamadas JAX-RPC/SOAP. As classes rosa são aquelas que tratam a API RMI e as amarelas foram criadas para abstrair a API CORBA. As próximas seções apresentarão as classes relevantes deste modelo, juntamente com a estratégia de implementação adotada.

3.4.1 FRDProxy

A classe FRDProxy é o único elemento do framework, além das exceções, que pode ser utilizado diretamente pelo desenvolvedor da aplicação cliente. Esta classe foi projetada de maneira a servir a dois propósitos: inicializar o *proxy* a partir da interface determinada e atuar como um elemento centralizador de todas as chamadas efetuadas pelos proxies dinâmicos.

De maneira a garantir que todas as chamadas de método nos *proxies* sejam manipuladas em um único local, a classe FRDProxy foi projetada de forma a implementar a interface *InvocationHandler* juntamente com o *design pattern Singletron*. Ao implementar o método *invoke* previsto na interface, FRDProxy torna-se uma classe apta a ser vinculada em qualquer *proxy* gerado dinamicamente. Já as premissas de uma classe *singletron* garantem que uma determinada máquina virtual poderá possuir, no máximo, uma instância desta classe. Em conjunto com os métodos que representam o *pattern Factory*, FRDProxy se torna o único elemento a efetuar a criação de *proxies* e a direcionar quaisquer chamadas de método dirigidas a tais objetos.

Implementar a classe FRDProxy como um *Singletron* envolveu os seguintes procedimentos:

i. Declarar o construtor da classe como privado, de forma e não permitir qualquer invocação por outras classes;
ii. Declarar um atributo privado e estático do mesmo tipo da classe. Este atributo é a instância *singletron*.

Ao implementar este *pattern*, o comportamento da classe se torna diferente daquele visto em classes normais. Uma das diferenças é a impossibilidade de reportar as exceções que possam ocorrer na chamada do construtor, pois o processo de criação do *singletron* é realizado automaticamente pela máquina virtual a partir da primeira

referência a um dos métodos estáticos[6] da classe. Isto influenciou na definição do ponto de inicialização da referência a FRDTransport, que poderia, em situações normais, ser realizado no construtor de FRDProxy. Considerando que uma única referência a FRDTransport é utilizada em FRDProxy, tornou-se necessário realizar a verificação de um atributo lógico que indica se o transporte foi corretamente instanciado no método que inicializa os *proxies.*

A utilização de objetos proxy dinämicos, nos quais esta proposta é baseada, exige que as instâncias sejam criadas a partir do método estático *newProxyInstance* da classe java.lang.reflect.Proxy. Sendo um dos principais objetivos deste *framework* proporcionar uma abstração de mecanismos, optou-se por adicionar à classe FRDProxy características de um Factory, de forma que esta classe represente o ponto de interação entre o desenvolvedor da aplicação e o framework. Na realidade o que se obteve foi uma implementação do pattern Factory que encapsula outro Factory, pois este também é o padrão adotado pela classe java.lang.reflect.Proxy. Tal característica é representada pelo método newInstance cuja função é retornar um objeto proxy que implemente a interface informada:

```
public static Object newInstance(Class classe) throws Exception
{
   Object retorno = proxy.initialize(new Class[]{classe});
   return retorno;
}
```

É importante ressaltar que a assinatura deste método revela uma decisão de projeto: embora a *Proxy* API permita que uma instância p*roxy* implemente diversas interfaces, optou-se por limitar este número a apenas uma. Esta decisão apóia-se no cenário de uso definido para este framework, o qual será posteriormente esclarecido na seção que aborda o FRDFrontController. Uma vez executado, o método delega a construção efetiva dos *proxies* ao método *initialize* que executa dois procedimentos principais:

i. Verifica se o status do FRDProxy, representado pelo atributo lógico *logged*, foi previamente atribuído, determinando, desta forma, a necessidade de instanciar o FRDTransport a ser utilizado no envio do método. Após a inicialização de um novo FRDTransport, o método *connect* desta classe é utilizado para estabelecer a conexão através do mecanismo encapsulado;

[6] Métodos estáticos podem ser executados a partir da classe, não exigindo a instanciação prévia de um objeto (GOSLING et al., 2000).

ii. Realiza a chamada ao método estático *newProxyInstance* da classe *java.lang.reflect.Proxy*, informando o *Classloader* a ser utilizado (no caso, o seu próprio), a interface recebida e a instância do tipo *InvocationHandler* que receberá as chamadas de método reificadas.

Conforme visto anteriormente, os *proxies* dinâmicos delegam a execução das mensagens por eles recebida a um objeto que implemente a interface *InvocationHandler*. Neste framework, o próprio FRDProxy implementa tal interface e, no momento da construção do objeto *proxy*, informa uma referência a si mesmo (*this*) de forma a vincular **todos** os *proxies* da aplicação à **única instância** (*singletron*) de FRDProxy existente na máquina virtual:

```
private Object initialize(Class[] interfaces) throws Exception {
   if (!proxy.transportOk) {
      transport = FRDTransport.getTransport();
      transport.connect();
   }
   return(Proxy.newProxyInstance(this.getClass().getClassLoader(),
                                 interfaces,
                                 this));
}
```

Complementarmente, a interface *InvocationHandler* exige que o método *invoke* seja implementado. Este método utiliza a meta classe *Method* da API de Reflexão para reificar a chamada de método recebida por um *proxy*. A partir deste meta objeto e dos parâmetros da chamada relacionados em um *array* de objetos, o framework pode delegar a execução do método à uma instância da classe FRDTransporte que, por sua vez, encaminhará o método através do mecanismo por ela representado. Uma das características mais interessantes deste método é a sua forma de repasse de exceções: por se tratar de um método que atua sobre a representação de qualquer operação, todas as exceções são lançadas como *Throwable*, ou seja, são representadas pela classe mais geral na hierarquia de exceções Java. Cabe aos mecanismos da API *Proxy* realizar a coerção para os tipos de exceção previstos na interface de cada *proxy* dinâmico que tenha sido gerado. O mesmo vale para o valor de retorno do método que é sempre representado como *Object* e, posteriormente, convertido para a classe definida na interface:

```
public Object invoke(Object proxy, Method method, Object[] args)
                                                      throws Throwable {
   Object retorno = new Object();
   try {

      retorno = dcTransport.invoke(proxy, method, args);

   } catch (UndeclaredThrowableException e) {
      e.printStackTrace();
   }
```

```
        return retorno;
    }
```

Embora o método *invoke* de FRDProxy seja simples, ele representa um dos possíveis gargalos do framework pois neste ponto ocorre a interceptação e reificação de **todos os métodos** interceptados através dos mecanismos de reflexão, bem como da geração dinâmica das classes *proxy*.

3.4.2 FRDTransport

FRDTransport e suas subclasses são responsáveis por abstrair os mecanismos de comunicação do restante do *framework*. Assim como FRDProxy, esta classe assume dois papéis distintos: prover os mecanismos para recuperação dos objetos de transporte e definir os métodos e o comportamento base de suas subclasses.

FRDTransport abstrai a maneira como os mecanismos de transporte são construídos através da implementação do pattern *Factory*. A decisão de utilizar tal *pattern* decorre da necessidade de proporcionar um meio padrão para inicialização dos objetos, evitando assim que a classe FRDProxy fique responsável por tal operação. Todavia a implementação deste comportamento é diferente daquele visto para as classes *proxy*, pois naquela situação as classes *proxy* são geradas via reflexão em tempo de execução, não exigindo uma codificação prévia. No caso das classes de transporte, o programador do framework deve prover uma classe que especialize FRDTransport para cada mecanismo de distribuição utilizado. Tendo, potencialmente, diversos mecanismos disponíveis, torna-se necessária a elaboração de um meio que identifique a classe desejada. Diante disso, definiu-se um arquivo de configurações no qual o parâmetro *FRDTransport.class* representa o nome completo da classe a ser utilizada:

```
FRDTransport.class=org.test.remote.client.transport.rmi.FRDTransportRMI
#FRDTransport.class=org.test.remote.client.transport.local.FRDTransportLocal
#FRDTransport.class=org.test.remote.client.transport.corba.FRDTransportCORBA
```

O arquivo de configurações, denominado *frdremote.properties*, se localiza junto à aplicação cliente, permitindo que diferentes configurações sejam utilizadas em cada máquina que esteja executando uma aplicação baseada no framework. Ou seja, é possível que cada instância da aplicação utilize um mecanismo de distribuição diferente.

A recuperação do mecanismo de transporte é realizada a partir das características reflexivas de Java: de posse do nome da classe representado em uma *string*, o *ClassLoader* é utilizado para obter uma referência à classe dinamicamente. O objeto retornado pelo carregamento dinâmico não é uma instância da classe solicitada, mas sim

um meta objeto do tipo *Class*, que representa a classe recém lida. Outro possível resultado desta operação é a exceção *ClassNotFoundException*, gerada quando o *ClassLoader* não encontra a classe solicitada no *classpath* da máquina virtual. A partir da referência à classe de transporte, novos procedimentos reflexivos são utilizados de modo a obter uma referência a um dos construtores implementados. O construtor recuperado é representado pela meta classe *Constructor*, a partir do qual as instâncias são, finalmente, criadas:

```
{...}

   FRDTransport retorno = null;

   defaultProps = new Properties();
   InputStream in = ClassLoader.
                        getSystemResourceAsStream("frdremote.properties");
   defaultProps.load(in);
   in.close();

   String classname = defaultProps.getProperty("FRDTransport.class");

   ClassLoader classLoader = ClassLoader.getSystemClassLoader();
   Class classe = classLoader.loadClass(classname);

   Constructor constructor = classe.getConstructor(new Class[]{});
   retorno = (FRDTransport) constructor.newInstance(new Object[]{});


{...}
```

Assim como no retorno dos métodos dos *proxies* dinâmicos, a chamada a *newInstance* a partir da referência ao construtor retorna uma instância representada pela classe *Object* que precisa ser convertida para o tipo desejado. Como todos os tipos de transporte são especializações de FRDTransport o retorno pode ser convertido para esta classe, aproveitando as características de orientação a objetos da linguagem.

Complementarmente, a classe FRDTransport define métodos abstratos, que deverão ser implementados por suas subclasses. O mais importante destes é o método *invoke* que é utilizado pela classe FRDProxy para delegar a execução de um método reificado:

```
{...}

abstract public boolean disconnect()throws Exception;
abstract public boolean connect() throws Exception;
abstract public Object invoke(Object proxy, Method method, Object[] args)
                                                        throws Throwable;

{...}
```

3.4.3 FRDFrontController

FRDFrontController é a classe que atua como um representante da aplicação cliente no computador remoto. Sua função é gerenciar a execução de todos os métodos

interceptados por FRDProxy e enviados através do par FRDTransport/FrontController específico de cada protocolo, utilizando características reflexivas para recompor os métodos recebidos da aplicação cliente.

Um dos requisitos exigidos nesta implementação é a capacidade de lidar com invocações de métodos em diferentes objetos. Esta situação ocorre, por exemplo, quando um usuário utiliza determinado processo da aplicação que possua uma chamada a outro processo. Nesta situação dois objetos remotos estarão simultaneamente em uso, exigindo de FRDFrontController a capacidade de direcionar a invocação à instância correspondente.

A recuperação das instâncias responsáveis por executar os métodos (*facades*) é realizada no método *criaFacade* a partir do nome da interface recebido como uma *string*. De forma a tornar este processo transparente para o desenvolvedor, algumas regras simples foram definidas para determinar a nomenclatura de classes e pacotes:

i. Toda interface que representa um serviço remoto deve possuir o prefixo **Remote**;
ii. Toda classe que representa um *facade* deve possuir o prefixo **Facade**, seguido do mesmo nome definido na interface;
iii. Tanto os *facades* quanto as interfaces devem ser definidas dentro da mesma estrutura de pacotes;
iv. Os *facades* devem, obrigatoriamente, implementar a interface.

Por exemplo, uma aplicação que defina a interface:

```
package com.corp.vendas;

public interface RemoteVendaExterna {
   {...}
}
```

Deve definir seu *facade* da seguinte maneira:

```
package com.corp.vendas;

public class FacadeVendaExterna implements RemoteVendaExterna {
   {...}
}
```

Embora tais convenções restrinjam a liberdade do programador de definir livremente os nomes de classes e a estrutura de pacotes, elas eliminam a necessidade de um processo de configuração para mapear cada interface a seu *facade*, ao mesmo tempo em que leva a uma padronização dos elementos que constituem o sistema. A mesma preocupação com a simplicidade do *framework* embasou a decisão de limitar a

associação entre interfaces e *proxies* a uma relação de um para um. Isto garante que as convenções acima sejam exeqüíveis, pois caso um *proxy* implementasse mais de uma interface não haveria a possibilidade de recuperar as instâncias da maneira descrita. Após a validação destas regras o método *criaFacade* passa a dispor do nome da classe que deverá ser instanciada, e procede da mesma maneira que o processo visto em FRDTransport: carregamento dinâmico da classe através do *ClassLoader*, recuperação do construtor e inicialização da instância via reflexão.

As características definidas até o momento são, na realidade, funções de apoio ao método principal desta classe, denominado *executaMetodo*. Este procedimento recebe os parâmetros coletados na aplicação cliente e verificar se existe uma instância previamente criada do *facade* indicado pelo parâmetro *classe*. Tal verificação é realizada em uma lista (*Hashtable*) que armazena instâncias desta natureza. Caso nenhuma instância seja encontrada, o método *criaFacade* é executado e o objeto retornado é adicionado à lista. Este comportamento possibilita a execução de múltiplos *facades* para uma mesma aplicação cliente. Após isto o método constrói, através de uma chamada reflexiva a *getClass* em cada argumento presente no array de *Objects*, um *array* de *Classes* representando os tipos de dados a serem utilizado na execução do método. Este *array* de *Classes* é utilizado, juntamente com o nome do método, em uma chamada reflexiva efetuada sobre o objeto *facade*, de modo a recuperar uma referência ao o método que, finalmente, é executado.

As exceções ocorridas durante as chamadas reflexivas são automaticamente representadas sob a forma de uma exceção genérica do tipo *java.lang.reflect.InvocationTargetException*. Esta classe de exceção possui um atributo representando a falta que realmente ocorreu no objeto alvo da invocação, a qual é recuperada através do método *getTargetException* e repassada como um *Throwable*. O código a seguir demonstra estas operações:

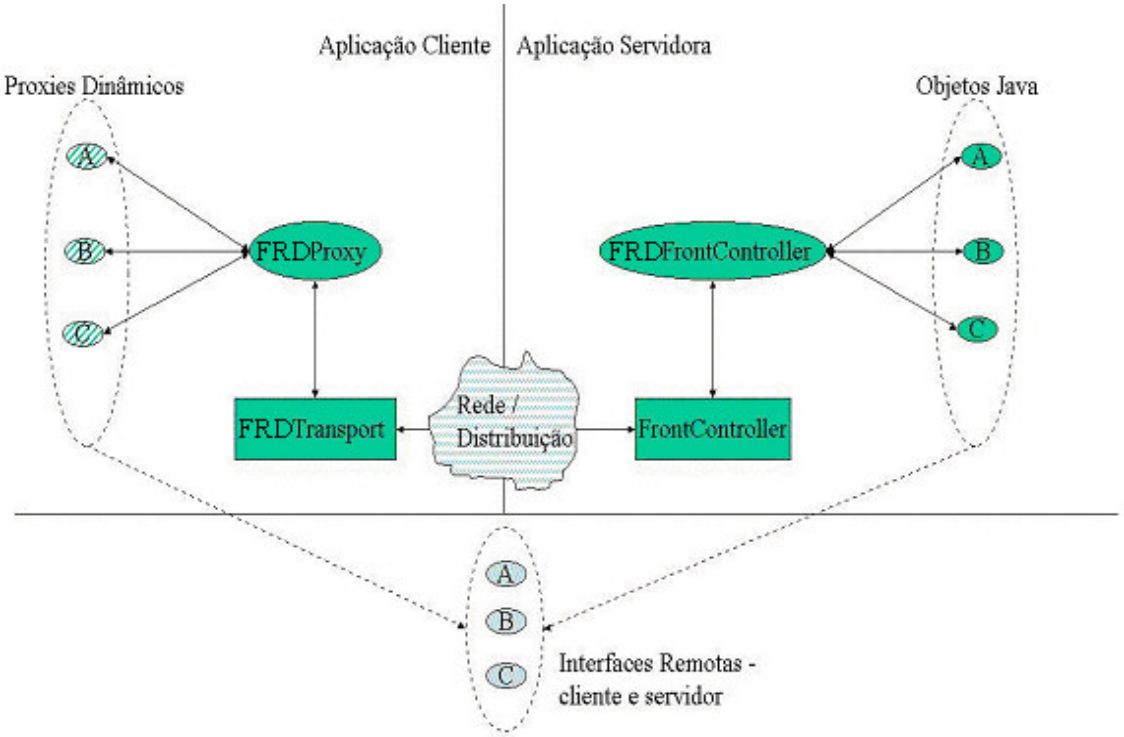
```
public Object executaMetodo(String classe,
                            String method, Object[] args)
                                              throws Throwable, Exception {
   String usecase = classe;
   Object sessionFacade = (Object) sessionFacades.get(usecase);
   if (sessionFacade == null) {
      sessionFacade = this.criaFacade(usecase);
      sessionFacades.put(usecase, sessionFacade);
   }
   Object retorno = new Object();
   Class[] paramTypes = new Class[]{};
   if ((args != null) && (args.length > 0)) {
      paramTypes = new Class[args.length];
      for (int i = 0; i < args.length; i++) {
```

```
            paramTypes[i] = args[i].getClass();
         }
      }
      Method m = sessionFacade.getClass().getMethod(method, paramTypes);
      try {
         retorno = m.invoke(sessionFacade, args);
      } catch (java.lang.reflect.InvocationTargetException ex) {
         throw ex.getTargetException();
      } catch (Exception ex) {
         ex.printStackTrace();
      }
      return retorno;
   }
```

3.4.4 Apresentação da arquitetura

A utilização do *framework* apresentado leva a uma arquitetura onde as aplicações cliente e servidora utilizam mecanismos de distribuição sobre os quais nenhum dos participantes toma conhecimento. Complementarmente a desejada simplicidade de utilização é obtida pois em nenhum momento exige-se do desenvolvedor a implementação de classes adaptadoras ou *proxies* para cada novo objeto que possa ser remotamente acessado. A figura 6 demonstra como os elementos interagem de forma possibilitar a comunicação cliente/servidor:

Figura 6 Arquitetura das aplicações desenvolvidas com o *framework*



Fonte: Elaboração própria

Como pode ser observado, as idiossincrasias relativas à distribuição foram abstraídas de ambos os lados da aplicação, cabendo à camada de transporte lidar com cada API de distribuição desejada. O contrato entre as aplicações cliente e servidora torna-se evidente através da utilização das mesmas interfaces na implementação tanto dos objetos localizados no servidor quanto nos *proxies* dinamicamente gerados. A seção seguinte apresenta a implementação dos mecanismos discutidos no capítulo 1.

FRD

3.5 Implementação da camada de transporte

A camada de transporte, constituída pelo par FRDTransport/FrontController, é o ponto do framework que trata as especificidades de cada mecanismo de distribuição. Sua principal função é realizar o repasse da chamada de método ocorrida no *proxy* para a classe encarregada das execuções remotas (*FRDFrontController*). Durante este processo as questões relativas a tratamento de parâmetros, tipos de dados e exceções devem ser convenientemente tratadas, de maneira a propiciar um comportamento consistente da aplicação cliente.

Uma decisão de projeto comum a todos os transportes aqui demonstrados é a necessidade de modificar a representação dos métodos: como a classe *Method* não é serializável, optou-se por repassar somente o nome do método, delegando às camadas posteriores do framework sua recuperação através das APIs de reflexão. Solução semelhante foi adotada com relação aos tipos dos parâmetros da chamada, exigidos no momento de uma invocação reflexiva a um método: devido à existência de um *array* contendo cada parâmetro representado como um *Object* evitou-se o envio de outro array de *Classes* (necessário para recuperar o método via reflexão), pois tal informação pode ser obtida, sempre que necessário, através do método *getClass* de qualquer objeto Java.

Cabe ressaltar que, dos dois elementos envolvidos na camada de transporte, o *framework* somente restringe o modelo de codificação das classes derivadas de FRDTransport. A implementação dos FrontControllers é totalmente livre, uma vez que estes podem, ser implementados de forma a tirar proveito das características específicas do mecanismo de transporte encapsulado. Apesar disso, todos os FrontControllers aqui implementados utilizaram a mesma assinatura para o método encarregado de receber e repassar as informações da chamada remota.

O restante desta seção apresentará exemplos de implementação da camada de transporte para os mecanismos de distribuição previamente apresentados.

3.5.1 RMI

Por se tratar do mecanismo padrão de distribuição da linguagem Java, a implementação do transporte em RMI torna-se extremamente simples. Não existe a necessidade de manipulação dos parâmetros do método remoto, dado que o mecanismo de serialização da linguagem é totalmente compatível com RMI.

Duas classes são responsáveis pelo transporte RMI: a RMITransport e RMIFrontController. RMITransport realiza as chamadas às interfaces de programação RMI para localizar e recuperar a instância de RMIFrontController a partir do serviço de nomes. RMIFrontController, por sua vez, é o único objeto RMI existente na aplicação e se encarrega de ativar uma instância de si mesmo e realizar sua vinculação no serviço de nomes, de maneira similar ao exemplo apresentado na seção 1.1.4.

Um cuidado especial foi dispensado ao tratamento de exceções, pois a classe *FRDProxy* espera que toda instância de exceção seja mapeada para a classe *Throwable*, conforme requerido pela Proxy API. Contudo RMI não permite que exceções do tipo *Throwable* sejam declaradas na cláusula de *throws* em métodos de objetos remotos, impedido um repasse direto através dos métodos implementados em RMIFrontController. Para evitar que toda exceção da aplicação tenha de especializar a classe *RemoteException*, definiu-se uma classe denominada *FRDRmiWrapperException*, cuja função é atuar como uma cobertura para as exceções ocorridas no servidor. Durante sua execução, a classe *RMIFrontController* captura qualquer exceção originada na invocação do método e a utiliza para construir uma nova instância de *FRDRmiWrapperException*. Quando a classe *FRDTransportRMI* intercepta exceções deste tipo o procedimento é inverso: a falta original encapsulada dentro da exceção de cobertura é recuperada e repassada através da instrução *throw*, garantindo que *FRDProxy* receba a exceção originalmente gerada:

```
public Object invoke(String classe, String method, Object[] args)
                                                    throws RemoteException,

FRDRmiWrapperException {
   try {

      return localFrontController.executaMetodo(classe, method, args);

   } catch (Throwable e) {
```

```
            throw new FRDRmiWrapperException(e);
        }
    }
```

A interface remota definida para a classe de transporte em RMI é bastante simples, uma vez que somente o método *invoke* é declarado:

```
public interface InterfaceRMIFrontController extends Remote{
  public Object invoke(String classe, String method, Object[] args)
                                                    throws RemoteException,
                                                           Exception;
}
```

3.5.2 CORBA

Ao contrário da facilidade quanto ao tratamento de parâmetros encontrada na implementação via RMI, as classes escritas para lidar com as APIs CORBA necessitam de um processo específico para realizar tal tarefa. Conforme visto na seção 1.2.1, CORBA possuí um mapeamento dos tipos de dados primários da linguagem IDL para as linguagens de implementação, no caso Java. Classes especificamente elaboradas para mapear conceitos do domínio da aplicação devem estar definidas em uma interface IDL como um **valuetype**, o que permite seu uso em chamadas remotas. Como esta exigência fere um dos princípios do *framework*, o encapsulamento das especificidades do mecanismo de distribuição, foi necessário definir uma forma de transportar os parâmetros para evitar este procedimento.

A solução encontrada baseou-se na utilização do mecanismo de serialização em conjunto com uma classe existente na API CORBA da linguagem Java, a classe *org.omg.CORBA. Any*. Esta classe é utilizada para armazenar os tipos de dados para os quais existe um mapeamento definido e, mais importante, dispõe de um meio para carregar qualquer objeto serializável. A partir disso definiu-se a operação *executaMetodo* da classe CORBAFrontControllerImpl com uma assinatura semelhante àquela apresentada em RMI utilizando, contudo, a classe *Any* no lugar de *Object* para representar os argumentos da chamada e seu valor de retorno. Como FRDProxy sempre envia os argumentos reificados a partir dos *proxies* dinâmicos como um array de *Objects*, a classe FRDTransportCORBA realiza a conversão para *Any* de cada argumento recebido. O código abaixo é responsável pela conversão no lado cliente, antes do envio para o servidor, bem como da recuperação do valor de retorno do método:

```
public java.lang.Object invoke(Object proxy, Method method, Object[]
args)
                                                                throws
Throwable {
```

```
      try {
         Any[] anyArgs = new Any[args.length];
         for (int i = 0; i < anyArgs.length; i++) {
            anyArgs[i] = orb.create_any();
            anyArgs[i].insert_Value((Serializable)args[i]);
         }
       Any resposta = frontController.executaMetodo(proxy.getClass().
                                                        getInterfaces()[0].
                                                           getName(),
                                                    method.getName(),
                                                    anyArgs);
         return resposta.extract_Value();
      } catch (FRDCorbaWrapperException e) {
         throw e.getCausa();
      }
   }
```

No servidor, a classe CORBAFrontControllerImpl realiza o processo inverso, recuperando os objetos originais a partir do conteúdo de cada elemento do *array* de *Any*, conforme exigido pela classe *FRDFrontController* e encapsulando o retorno da chamada em um novo *Any*:

```
public Any executaMetodo(String classe, String metodo, Any[] argumentos)
                                              throws FRDCorbaWrapperException
{
   java.lang.Object[] objArgs = new
   java.lang.Object[argumentos.length];
   for (int i = 0; i < objArgs.length; i++) {
      objArgs[i] = argumentos[i].extract_Value();
   }
   Any resultado = orb.create_any();
   try {
   resultado.insert_Value( (Serializable) localFrontController.
                                          executaMetodo(classe,
                                                        metodo,
                                                        objArgs));
   } catch (Exception e) {
      Any exception = orb.create_any();
      exception.insert_Value(e);
      FRDCorbaWrapperException ex = new
                                FRDCorbaWrapperException(exception);
      throw ex;
   }
   return resultado;
}
```

O processo de tratamento de parâmetros e valores de retorno torna desnecessário definir um **valuetype** para cada tipo de objeto do domínio da aplicação utilizado na invocação de métodos, uma vez que, ao utilizar a classe *Any* para transportar objetos serializados, o ORB não toma conhecimento de sua existência, sendo usado somente para o efetivo transporte da representação. Esta representação, ao ser recebida pelo cliente, pode ser normalmente reconstituída pelo mecanismo de serialização.

Esta mesma abordagem foi utilizada no tratamento de exceções pois, conforme destacado na seção 1.2.2, a especificação CORBA também exige que todas as classes de exceção de usuário sejam declaradas através da IDL. Como isto leva ao mesmo problema dos **valuetypes**, optou-se por implementar uma exceção de cobertura, similar

àquela vista em RMI, contendo um atributo do tipo *Any* para armazenar a exceção original em um formato passível de ser transportado pelo *Object Request Broker*:

```
public final class FRDCorbaWrapperException extends
org.omg.CORBA.UserException {
   public org.omg.CORBA.Any causa = null;

   public Exception getCausa() {
      return (Exception)(Serializable)this.causa.extract_Value();
   }

  { ... }

}
```

Como toda aplicação CORBA, a camada de transporte aqui definida foi elaborada a partir de uma definição IDL na qual estão declaradas a assinatura do método *invoke* juntamente com a exceção de cobertura previamente citados:

```
module org {
   module frd {
      module remote {
         module server {
            module corba {
              exception FRDCorbaWrapperException {
                 any causa;
              };
              typedef sequence<any> Args;
              interface CORBAFrontController {
                 any invoke(in string classe,
                            in string metodo,
                            in Args argumentos)
                                 raises (FRDCorbaWrapperException);
              };
            };
         };
      };
   };
};
```

3.5.3 JAX-RPC/SOAP

O modelo de programação previsto na especificação JAX-RPC exige que cada classe do domínio da aplicação seja serializada por uma classe específica (ver seção 1.3.4). Em uma aplicação desenvolvida com o uso explícito deste mecanismo de distribuição, este procedimento é automatizado pelo compilador disponível no ambiente de desenvolvimento. Contudo a utilização deste compilador pelo desenvolvedor da aplicação não é uma opção válida no contexto deste *framework*, assim como não o é a definição de interfaces IDL no caso da implementação via CORBA, visto que tais ações expõem características específicas de cada API.

Similarmente à solução adotada em CORBA, utilizou-se a serialização de forma a obter uma representação que permitisse o envio de tais classes de maneira genérica. Como JAX-RPC não disponibiliza uma classe auxiliar como a classe *Any* da API

CORBA, foi necessário recorrer a uma representação de dados suportada pela API. Optou-se por realizar a conversão dos objetos para *arrays* de *bytes* através das classes de serialização de objetos Java (*ObjectInputStream* e *ObjectOutputStream*). Conseqüentemente, e novamente de maneira similar ao transporte CORBA, a classe FRDTransportXML se encarrega de converter cada argumento envolvido na invocação de método. Abaixo observa-se a interface definida para o serviço e a implementação do método *invoke*:

```
public interface InterfaceXMLFrontController extends Remote{
   public byte[] invoke(String classe, String method, byte[][] args)
                                                    throws RemoteException,

FRDXmlWrapperException;
}


{ ... }
public Object invoke(Object proxy, Method method, Object[] args)
                                                        throws Throwable {
   try {
      ByteArrayOutputStream byteArrayOS = new ByteArrayOutputStream();
      ObjectOutputStream objectOS = new ObjectOutputStream(byteArrayOS);
      ByteArrayInputStream byteArrayIS = null;
      ObjectInputStream objectIS = null;
      byte[][] byteArgs = new byte[args.length][];
      for (int i = 0; i < byteArgs.length; i++) {
         objectOS.writeObject(args[i]);
         byteArgs[i] = byteArrayOS.toByteArray();
         objectOS.reset();
      }
      byte[] resultado = frontController.invoke(

 proxy.getClass().getInterfaces()[0].getName(),
                           method.getName(),
                           byteArgs);
      if (resultado == null)
         return null;
      byteArrayIS = new ByteArrayInputStream(resultado);
      objectIS    = new ObjectInputStream(byteArrayIS);
      return objectIS.readObject();
   } catch (FRDXmlWrapperException e) {
      throw e.getCause();
   }
}
{ ... }
```

Embora as classes utilizadas para realizar a serialização também ofereçam suporte à representação dos objetos sob o formato de *strings*, optou-se pela utilização de *bytes* por questões de performance. Algumas considerações sobre esta decisão serão realizadas na seção relativa às medições obtidas nas invocações de método.

Na outra ponta da comunicação, o processo servidor implementado pela classe XMLFrontController se encarrega de obter os objetos originais a partir conteúdo de cada *array* de *bytes*, possibilitando a chamada da operação *executaMetodo* de FRDFrontController:

```
public byte[] invoke(String classe, String method, byte[][] argumentos)
                                        throws RemoteException,
                                               FRDXmlWrapperException
```

```
{
   try {
      Object[] objArgs = new java.lang.Object[argumentos.length];
      if (argumentos.length > 0) {
         byteArrayIS = new ByteArrayInputStream(argumentos[0]);
         objectIS    = new ObjectInputStream(byteArrayIS);
         objArgs[0] = objectIS.readObject();
         for (int i = 1; i < objArgs.length; i++) {
            byteArrayIS.reset();
            byteArrayIS.read(argumentos[i]);
            objArgs[i] = objectIS.readObject();
      }
   }
   Object resultado = localFrontController.executaMetodo(classe,
                                                          method,
                                                          objArgs);
   if (resultado == null)
      return null;
   objectOS.writeObject(resultado);

   return byteArrayOS.toByteArray();

   } catch (Throwable e) {
      throw new FRDXmlWrapperException(e);
   }
}
```

Como pode ser observado nos trechos de código acima, o procedimento definido para reportar as exceções ocorridas nas classes da aplicação foi o encapsulamento em uma exceção do *framework* (FRDXmlWrapperException), exatamente como no transporte RMI, permitindo a representação de qualquer exceção de maneira padronizada. Esta classe de exceção é idêntica à FRDRmiWrapperException e a opção de criar esta nova classe ao invés de reaproveitar aquela existente em RMI deveu-se à perspectiva de futuras especializações da camada de transporte de cada mecanismo.

3.5.4 Local

O desenvolvimento de uma aplicação distribuída implica em um ciclo de desenvolvimento mais longo que um software não-distribuído. Sempre que o programador da codifica elementos servidores, existe a necessidade de refazer o processo de *deploy*[7] antes de realizar os testes na implementação realizada. Além disso, o processo de depuração de uma aplicação distribuída é mais complexo, envolvendo a execução do processo de depuração em ambos os lados da aplicação (cliente e servidor). Complementarmente deve-se considerar o maior tempo de execução devido à sobrecarga imposta pelo mecanismo de distribuição e pela própria latência da rede. Frente a estes fatores, e contando com a flexibilidade disponibilizada pela arquitetura proposta, optou-se por implementar uma camada de transporte que, na realidade, não

[7] Instalação dos componentes ou classes, juntamente com suas configurações, em um servidor capaz de executá-los.

abstrai nenhum mecanismo de distribuição. Sua função é propiciar a execução das classes na mesma máquina virtual (local) que a parte cliente da aplicação, ao mesmo tempo em que o modelo de programação proposto pelo *framework* é mantido.

A classe FRDTransporteLocal, assim como os demais componentes de transportes, realiza o repasse dos argumentos recebidos em seu método *invoke* mas, devido à não existência de APIs para distribuição envolvidas no processo, nenhum tratamento é necessário nos parâmetros da chamada "remota" ou nas exceções eventualmente geradas durante a execução do método. O código a seguir apresenta esta abordagem:

```
public Object invoke(Object proxy, Method method, Object[] args)
                                                    throws Throwable {
   Object retorno = null;
   retorno = localFrontController.
        executaMetodo(proxy.getClass().getInterfaces()[0].getName(),
                      method.getName(),
                      args);
   return retorno;

}
```

3.5.5 Comentários

A implementação da camada de transporte do *framework* para RMI, CORBA e JAX-RPC/SOAP evidencia atividades que podem ser consideradas como padrão para implementações através de outros mecanismos. Tais atividades devem responder às seguintes questões:

i. Qual representação dos argumentos da chamada de método é suportada pelo mecanismo em questão e como realizar a conversão dos objetos Java para tal formato?

ii. Diante de várias representações possíveis para os argumentos, qual delas é mais eficiente em volume e facilidade de processamento?

iii. De que maneira as exceções são reportadas pelo mecanismo utilizado? É possível utilizar exceções do domínio da aplicação diretamente?

Além destas questões ressalta-se que as classes derivadas de FRDTransport são responsáveis por localizar e obter uma referência ao serviço remoto. Como este processo requer APIs específicas de cada mecanismo de distribuição, a definição de um arquivo de configurações que armazene os parâmetros específicos deve ser considerada.

Deve-se ressaltar que a forma de serialização escolhida na implementação do transporte CORBA e JAX-RPC foi baseada na premissa de que **todos** os argumentos da chamada reificada pelo *framework* são objetos. Desta forma, qualquer tentativa de utilizar tipos de dados primários resultará em um erro de serialização.

Finalmente constata-se que o projeto da camada de transporte do *framework* foi efetivo ao propiciar a abstração de três mecanismos de distribuição sem exigir alterações nas classes que realizam a interação com o cliente, especialmente FRDProxy, e na classe responsável pela invocação no lado servidor (FRDFrontController).

3.6 Aplicando o framework

Como forma de demonstrar a aplicabilidade do *framework* e avaliar a que nível os requisitos foram atendidos, definiu-se uma aplicação composta de um único caso de uso: a manutenção de um cadastro de produtos. A intenção deste aplicativo não é realizar as atividades normalmente encontradas em aplicações similares (conexão com banco de dados, controle de transações, controle de acesso, etc), mas sim definir as interfaces e classes do domínio da aplicação que sejam remotamente executadas segundo o modelo definido neste estudo. Esta implementação foi realizada a partir das seguintes regras impostas pelo *framework*:

i. Todos os parâmetros e retorno de métodos devem ser serializáveis, e não devem incluir tipos primários de dados (int, float, etc), mas sim as classes equivalentes (Integer, Float, etc);
ii. Deve ser definida uma interface para toda classe que servirá como um *façade*, onde estarão os métodos passíveis de serem acessados remotamente. O nome desta classe deverá ser definido com o prefixo *Remote*;
iii. Deve ser definida uma classe *façade* que implemente os métodos da interface anteriormente citada. O nome desta classe deverá ser definido com o prefixo *Facade*;
iv. Tanto a interface quanto o *façade* devem estar contidos dentro do mesmo pacote, de forma a possibilitar o carregamento dinâmico da classe;
v. A interface deverá ser distribuída nas duas camadas da aplicação – cliente e servidora – uma vez que a obtenção da referência remota será realizada através desta classe.

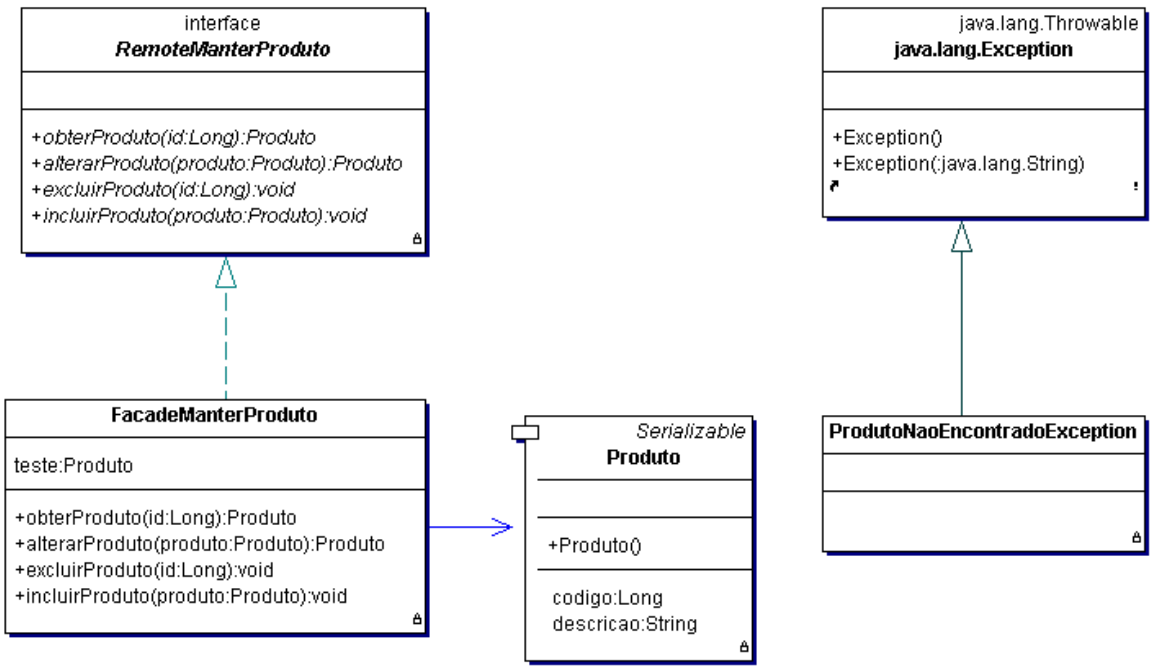
Diante disso foram elaboradas quatro classes que implementam a estrutura desejada para este exemplo: RemoteManterProduto, FacadeManterProduto, Produto, que representa a entidade do domínio da aplicação, e a exceção ProdutoNaoEncontradoException. Para confirmar a capacidade do *framework* de lidar com chamadas remotas que envolvam classes de aplicação dois métodos foram implementados no *façade*:

i. *obterProduto*, que recebe como parâmetro um número identificador do produto desejado e retorna uma instância da classe Produto;

ii. *alterarProduto*, que recebe como parâmetro uma instância de Produto e cuja função é somente lançar uma exceção do tipo ProdutoNaoEncontradoException.

O diagrama de classes apresentado na figura 7 representa os elementos da aplicação:

Figura 7 Diagrama de classes da aplicação-exemplo



Fonte: Elaboração própria

Com base neste modelo de classes voltadas ao servidor, elaborou-se a da aplicação cliente responsável pela chamada dos métodos previstos na interface RemoteManterProduto. A forma de interação desta classe com os demais objetos da aplicação é idêntica ao realizado caso o *framework* não fosse utilizado, exceto pelas

instruções necessárias à obtenção da instância do *facade* que, em Java é obtida da seguinte maneira:

```
RemoteManterProduto facade = new FacadeManterProduto();
```

Todavia, para permitir a criação dos *proxies* responsáveis pela representação do serviço remoto, a obtenção da instância deve ser realizada através do método *factory* previsto na classe FRDProxy. Este método retorna, a partir da interface informada em seu parâmetro, o novo objeto *proxy* que redirecionará, através das demais classes do *framework*„ toda chamada de método realizada. Diante disso, o código apresentado acima deve ser modificado para:

```
RemoteManterProduto facade = (RemoteManterProduto) FRDProxy.
                                newInstance(RemoteManterProduto.class);
```

Além desta alteração, deve-se prever também o tratamento das exceções específicas do *framework* que eventualmente possam ocorrer. A implementação aqui apresentada elaborou tais exceções como especializações de *java.lang.RuntimeException* que, por se tratar de uma exceção não checada (*unchecked*) (GOSLING et al., 2000), dispensa a utilização do bloco *try/catch*[8]. Desta forma o código da aplicação cliente preocupa-se em manipular somente as exceções definidas na assinatura dos métodos do *facade*, ou seja, exceções do domínio da aplicação.

Com base nesta aplicação foram realizadas medições de performance que permitiram, por um lado, evidenciar a adequação do *framework* aos requisitos apresentados no começo deste capítulo e, de outro, averiguar o comportamento diante da alteração do mecanismo de transporte. Estas avaliações estão detalhadas na seção 3.7.

3.7 Medições

As seções 3.4 e 3.4.4 apresentaram os procedimentos adotados pelo *framework* de maneira a obter a funcionalidade esperada. Observa-se que toda chamada de método realizada a partir de uma aplicação cliente é manipulada de maneira a abstrair o mecanismo de distribuição adotado. Dentre as operações utilizadas neste processo, destacam-se os procedimentos reflexivos voltados à geração de *proxies*, interceptação de métodos, introspecção de propriedades dos parâmetros como o nome de sua classe e

[8] Embora a opção de utilizar exceções não-checadas permita que o código seja escrito sem o tratamento de tais exceções, deve-se ressaltar que quando tais exceções ocorrerem, a aplicação será imediatamente finalizada.

obtenção de valores, criação de instâncias a partir dos construtores padrão entre outros. Complementarmente atividades voltadas à serialização dos parâmetros e valores de retorno também são executadas no contexto da camada de transporte. Este volume de operações incorre, claramente, em uma sobrecarga na execução da aplicação. Diante disto, esta seção avalia a que ponto a utilização do *framework* proposto afeta a performance, fornecendo subsídios para definir as situações nas quais sua utilização é viável.

As medições foram realizadas a partir de uma aplicação principal realizando 10.000 invocações do método *obterProduto* apresentado na seção anterior. O processo foi executado cinco vezes, sendo que as duas primeiras execuções foram descartadas de maneira a não considerar o tempo de otimização necessário ao interpretador Java nas situações em que o processo servidor estava ativo. Neste processo foram utilizadas duas máquinas AMD Duron 950MHz com 256 Mb de memória RAM, rodando o Windows 2000 Professional atualizado com o *service pack* 3 e interligadas diretamente através de um cabo de rede *crossover* a 100Mbps. A JVM utilizada foi a versão 1.4.1_02, *build* 1.4.1_02-b06, utilizando o HotSpot Client em modo *mixed*. Nenhum parâmetro adicional foi utilizado na JVM. Os dados da execução da aplicação foram coletados pela ferramenta JProfiler v2.2.1 da EJ Technologies, disponível no site http://www.jprofiler.com .

A abordagem escolhida para realizar as medições buscou evidenciar o impacto de cada técnica utilizada nos componentes do *framework*. Inicialmente são apresentados os números que demonstram o tempo das chamadas de método local confrontada com a execução do mesmo método via reflexão e através do *framework* configurado para utilizar a camada de transporte em modo local. Em seguida verificou-se a execução do mesmo procedimento através de uma implementação direta das três opções de distribuição apresentadas (CORBA, RMI e JAX-RPC/SOAP) juntamente com o uso do *framework* configurado para cada uma destas opções. Ressalta-se que todas as medições desconsideraram o tempo necessário à obtenção da instância remota, de forma a mensurar somente o tempo total de execução da chamada.

Tabela 1 Medição de 10.000 invocações de métodos locais – em milisegundos

	M1	M2	M3	média	por chamada	acréscimo 1	acréscimo 2
Direta	196	220	204	207	0,020667		
Reflexiva	754	722	735	737	0,073700	3,5661	

Framework	1325	1608	1452	1462	0,146167	7,0726	1,9833

Conforme pode ser observado na tabela 1, o tempo total de execução do método é extremamente reduzido devido à simplicidade da lógica implementada. Na realidade este método simplesmente instancia um objeto da classe Produto, atribui valores a suas variáveis de instância (código e descrição) e utiliza esta instância como retorno do método. A segunda medição demonstra a sobrecarga imposta pela utilização das facilidades reflexivas, através das quais o objeto *Method* correspondente ao procedimento a ser executado é recuperado e a chamada é realizada de maneira indireta através do método *invoke*. O tempo de execução, nesta situação, é elevado em 3,5661 vezes em relação à chamada direta, conforme demonstrado na coluna "acréscimo 1". Em seguida verifica-se que o uso do *framework* em modo local incorreu em um tempo superior a sete vezes a execução direta. Todavia, ao comparar sua performance com aquela verificada na simulação reflexiva, constata-se que houve um acréscimo de 98,33% ("acréscimo 2"), valor que pode ser considerado reduzido em função do processamento realizado pela classe FRDFrontController. Complementarmente, deve-se considerar que obter a representação reflexiva do método a cada invocação não é uma abordagem eficiente: este processo pode ser mais veloz se combinado com um mecanismo que mantenha em *cache* os métodos previamente recuperados, de maneira análoga ao realizado com as instâncias das classes *facade*.

Tabela 2 Medição de 10.000 invocações de métodos remotos – em milisegundos

		M1	M2	M3	Média p/ chamada		acréscimo 1	acréscimo 2
RMI	Direto	35665	35883	35993	35847	3,584700	173,4532	
	framework	44030	44483	42267	43593	4,359333		1,2161
CORBA	Direto	35234	34983	35699	35305	3,530533	170,8323	
	framework	71480	71979	71728	71729	7,172900		2,0317
SOAP	Direto	181261	182199	182994	182151	18,215133	881,3774	
	framework	685012	684025	691302	686780	68,677967		3,7704

A tabela 2 apresenta as medições obtidas com o uso dos mecanismos de distribuição, em um primeiro momento em uma utilização direta e, em seguida, através do *framework*. Independentemente do mecanismo que seja analisado a sobrecarga decorrente da invocação remota torna-se evidente, o que pode ser observado na coluna "acréscimo 1", que demonstra quanto uma chamada efetuada através de cada

mecanismo é mais lenta que a mesma chamada local, apresentada na tabela 1. A coluna "acréscimo 2" determina o impacto sobre a performance de cada padrão de distribuição, exibindo a razão entre a performance obtida de cada mecanismo antes e depois da aplicação do *framework*.

As medições com RMI mostram um tempo 173,45 vezes ("acréscimo 1") superior à invocação direta do mesmo método. Curiosamente, a implementação CORBA disponibilizada pela versão 1.4.1_02 do ambiente Java apresentou uma performance ligeiramente superior a RMI neste teste. Os resultados da comunicação via JAX-RPC/SOAP evidenciaram a penalidade resultante de uma representação menos eficiente (XML) que os formatos binários adotados por CORBA e RMI, que resultou em um acréscimo da ordem de 881,37 vezes sobre o método original sendo, aproximadamente, cinco vezes mais lento que os demais mecanismos.

Considerando os resultados obtidos com o uso do *framework*, evidencia-se a vantagem da implementação mais próxima da linguagem adotada por RMI: como seu processo de serialização é o padrão da plataforma, a sobrecarga é menor e mostra-se bastante aceitável: 21,61% ("acréscimo 2"). O transporte CORBA, por sua vez, exige o encapsulamento dos dados nas classes *Any* complementarmente ao processo de serialização. Isto explica a grande diferença frente a RMI, frente às boas marcas obtidas nas invocações CORBA nativas, onde a serialização é tratada diretamente pelas classes geradas durante a compilação da IDL. Esta mesma justificativa se aplica à excessiva sobrecarga do *framework* nas chamadas via JAX-RPC, pois o processo de serialização não é executado pelas classes utilitárias geradas para cada classe Java mapeada para SOAP, mas sim através mapeamento padrão da especificação que trata os *arrays* de *bytes* utilizados na interface do XMLFrontController. Este tipo de dados é representado, em SOAP como um *xsd:base64Binary*, exigindo uma grande alocação de memória (BOSWORTH et al., 2003) para tratar o pequeno volume de informações da chamada de método apresentada. Existem, todavia, estudos voltados à definição das representações de dados em SOAP nos quais são sugeridas abordagens voltadas ao acréscimo de a performance (ENGELEN, 2003) que poderiam ser utilizadas para incrementar o desempenho da camada de transporte implementada com este mecanismo.

Esta avaliação de performance demonstra que, em todas as situações apresentadas, a utilização das camadas de abstração implementadas pelo *framework* implica em uma perda de performance quando comparado à invocação do mesmo método sem a sua intermediação. Contudo deve-se observar que o cenário no qual esta proposta se aplica consiste em aplicações distribuídas que se valem do padrão de projeto *facade*, ou seja, aplicações explicitamente projetadas de maneira a concentrar diversas invocações de método curtas em uma classe que disponibilize um serviço menos granular. Esta característica é um dos objetivos deste padrão de projeto: reduzir o número de interações necessárias entre a camada cliente e servidora da aplicação, tendo em vista o custo elevado decorrente das invocações remotas de método. Ou seja, os métodos implementados em uma classe que represente um *facade* são, normalmente, processos mais elaborados que realizam acesso a outros recursos disponibilizados no servidor (sistema de arquivos, sistemas de gerenciamento de banco de dados, etc), o que acaba por tornar o tempo necessário à distribuição apenas uma fração do processamento total realizado.

Conclusão

O objetivo principal desta dissertação, o isolamento dos mecanismos de distribuição em componentes de um *framework* foi plenamente atingido. Através dos elementos propostos no Capítulo 3, tanto as classes cliente como as servidoras de um software podem ser desenvolvidas sem qualquer vínculo a APIs de um padrão de distribuição. Desta maneira a flexibilidade é incrementada, pois o desenvolvimento da aplicação pode ser realizado independentemente da escolha do mecanismo de distribuição. Ao postergar esta decisão para estágios mais avançados de um projeto, propicia-se ao arquiteto do *software* o tempo necessário para que avaliações mais aprofundadas sejam realizadas, visando selecionar a melhor solução para o problema a ser abordado.

Complementarmente a reflexão computacional, discutida no Capítulo 2, mostrou-se adequada para solucionar o problema de distribuição ao permitir a elaboração de mecanismos genéricos e reutilizáveis para todos os padrões de distribuição apresentados no Capítulo 1. Em nenhum momento foi necessária a utilização de extensões reflexivas voltadas a complementar o padrão da linguagem Java, demonstrando que as características inerentes à plataforma foram suficientes para o problema deste trabalho e, por outro lado, garantindo que a portabilidade da implementação realizada em qualquer máquina virtual padrão.

Adicionalmente a abordagem adotada reduziu sobremaneira a complexidade envolvida na implementação de aplicações distribuídas. O exemplo de aplicação demonstrado no Capítulo 3 pôde ser desenvolvido com classes “normais”, sem a necessidade de implementar interfaces específicas, gerar descritores em IDL, *stubs*, *skeletons*, realizar o *binding* em serviços de nomes entre outros passos necessários à distribuição vistos no Capítulo 1. Simplesmente implementou-se classes em conformidade com as pequenas restrições impostas pelo *framework*. Outra característica que contribuiu para esta simplificação é a possibilidade de realizar o desenvolvimento a partir de uma configuração que permita a execução local, na qual nenhum servidor de aplicações esteja presente. Isto permite uma maior desenvoltura no ciclo codificação/compilação/depuração, uma vez que a instalação das classes geradas em um servidor de aplicações pode ser efetivada depois que todos os testes tenham sido

corretamente executados em modo local, evitando a presença de um fator complicador: os mecanismos de distribuição.

Finalmente deve-se ressaltar que, embora exista uma penalidade imposta à performance, conforme demonstrado nas medições do Capítulo 3, o *framework* tem se mostrado totalmente funcional em situações reais de uso, onde a premissas de métodos menos granulares na interface dos *facades* é respeitada. Exemplos de métodos atualmente utilizados incluem validação de campos da interface com o usuário, recuperação de informações em bancos de dados, invocação de métodos responsáveis por transações com registros (inclusões, alterações e exclusões) entre outros. Tal utilização prática tem demonstrado que outros fatores, como a sobrecarga da rede, do servidor de aplicações e de banco de dados, tendem a eliminar qualquer percepção por parte dos usuários das aplicações cliente, permitindo a manutenção da flexibilidade e simplicidade obtidas com o uso do *framework*.

Trabalhos relacionados

SOARES et al (2002) propõe a utilização do AspectJ, uma extensão da linguagem Java como forma de abstrair a distribuição da aplicação. Os autores propõem o redirecionamento das chamadas de método ocorrida na aplicação cliente através de *aspectos*. Todavia esta abordagem ainda exige a definição de uma codificação específica para cada método do *facade*, embora os autores a considerem mais simples que o necessário nas abordagens tradicionais, baseadas em um par de adaptadores, opinião reforçada pelos argumentos desta dissertação.

ALVES & BORBA (2001) definem um padrão de projeto denominado *Distributed Adapters Pattern* (DAP), cuja implementação exige o uso de um par de adaptadores para cada objeto remoto. O trabalho dedica especial atenção a um método de implementação progressiva (PIM) permitida pelo uso deste padrão. Embora o presente trabalho não tenha sido elaborado no contexto de um método similar, as características de isolamento e substituição de mecanismos de distribuição se encaixam perfeitamente nas premissas do citado método.

RINE et al (1999) apresenta uma proposta de utilização de adaptadores para reduzir a complexidade nas interações e aumentar o reuso de uma aplicação baseada em componentes. Os adaptadores são automaticamente gerados a partir de informações

definidas externamente à cada componente, evitando a codificação manual destes elementos exigida nos trabalhos de SILVA et al (1997) e ALVES & BORBA (2001).

Trabalhos futuros

A utilização das características reflexivas de Java tornou os elementos do *framework* bastante discretos para o desenvolvedor. Todavia a sua utilização pela aplicação cliente ainda possui um ponto que evidencia sua presença: a recuperação das instâncias de objetos remotos através do *factory* implementado na classe FRDProxy. Embora quando comparada às instruções necessárias a qualquer padrão de distribuição esta intervenção no modelo de programação seja mínimo, seria interessante eliminar tal evidência. Uma das alternativas promissoras para esta implementação é a utilização da programação orientada a aspectos (KICZALES et al., 1997) como forma de obter uma abordagem mais abstrata que o padrão *factory* adotado neste trabalho.

Além disso, a implementação da classe FRDFrontController foi extremamente simplificada, com o intuito de demonstrar a viabilidade da abordagem deste estudo. À implementação atual devem ser adicionados recursos mais sofisticados de gerenciamento de instâncias, *cache* de métodos reificados, utilização de *threads* para o processamento concorrente, entre outros. Tais características, juntamente com uma implementação mais elaborada das camadas de transporte, tendem a reduzir a sobrecarga observada nas medições do Capítulo 3.

Bibliografia

ALUR, Deepak; CRUPI, John; MALKS, Dan. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall & Sun Microsystems Press. 2001.

ALVES, Vander; BORBA, Paulo. *An Implementation Method for Distributed Object-Oriented Applications*. In: XV Simpósio Brasileiro de Engenharia de Software (SBES), (out. 2001, Rio de Janeiro, Brasil).

ARMSTRONG, Eric; BODOF, Stephanie; CARSON, Debbie et al. *The Java Web Services Tutorial*. 2003. [acessado em 10 abr. 2003]. Disponível em: http://java.sun.com/webservices/docs/1.1/tutorial/doc/index.html.

BEDUNAH, J.B. *XML: The Future Of The Web*. ACM Crossroads. 1999. [acessado em 10 abr. 2003]. Disponível em: http://www.acm.org/crossroads/xrds6-2/future.html.

BOOCH, Grady; JACOBSON, Ivar; RUMBAUGH, James et al. *The Unified Modeling Language User Guide*. Addison Wesley. 2000.

BOSWORTH, Adam; BOX, Don; GUDGIN, Margin et al. *XML, SOAP and Binary Data version 1.0*. 2003. [acessado em 18 mai. 2003]. Disponível em: http://msdn.microsoft.com/webservices/understanding/webservicebasics/default.aspx?pull=/library/en-us/dnwebsrv/html/infoset_whitepaper.asp

CHIBA, Shigeru. *A Metaobject Protocol for C++*. In: Proceedings of the 10th Annual Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'95), (out. 1995, Austin, EUA), p. 285-299.

CHIBA, Shigeru. *Load-time Structural Reflection in Java*. In: 14th European Conference on Object-Oriented Programming (ECOOP 2000), (jun. 2000, Sophia Antipolis & Cannes, France), p. 313–336.

CHRISTENSEN, Eric; CURBERA, Francisco; MEREDITH, Greg et al. *W3C. Web Services Description Language (WSDL) 1.1*. 2001. [acessado em 10 jan. 2003]. Disponível em: http://www.w3.org/TR/wsdl.

CHU-CARROL, Mark C. *Separation of Concerns: An Organizational Approach*. 2000. [acessado em 10 jan. 2003]. Disponível em: http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/chucarroll.pdf

DASHOFY, Eric M.; MEDVIDOVIC, Nenad; TAYLOR, Richard N. *Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures*. In: Proceedings of the 1999 International Conference on Software Engineering, (mai. 1999, Los Angeles, USA), p. 3-12.

DOURISH, Paul. *Open Implementation and Flexibility in CSCW Toolkits*. 1996. Tese de Doutorado, University College, Londres, Inglaterra. Disponível em: http://www.parc.xerox.com/csl/members/dourish/thesis.html

ENGELEN, Robert A. *Putting the SOAP Envelope With Web Services for Scientific Computing*. 2003. [acessado em 10 mai. 2003]. Disponível em: http://websrv.cs.fsu.edu/~engelen/icws03.pdf

FERBER, Jacques. *Computational Reflection in Class based Object Oriented Languages*. In: Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'89), (out. 1989, New Orleans, EUA), p. 317-326.

FIELDING, R.; GETTYS, J.; MOGUL, J. et al. *Hypertext Transfer Protocol -- HTTP/1.1*. 1999. [acessado em 10 abr. 2003]. Disponível em: http://www.ietf.org/rfc/rfc2616.txt.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph et al. *Design Pattern: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

GOMAA, Hassan; MENASCÉ, Daniel A. *Design and performance modeling of component interconnection patterns*. In: Proceedings of Second International Workshop on Software and Performance, (set. 2000, Ottawa, Canada), p. 117-126.

GOSLING, James; JOY, Bill; STEELE, Guy et al.. *The Java Language Specification*. 2 ed., Addison-Wesley, 2000.

HALTEREN, A.T.; NOUTASH, A.; NIEUWENHUIS, L.J.M et al. *Extending CORBA with specialised protocols for QoS provisioning*. In: Proceedings of International Symposium on Distributed Objects and Applications (DOA'99), (set. 1999, Edinburgh, Scotland).

HAROLD, Elliotte R. *Processing XML with Java*. 2002. Livro Eletrônico. [acessado em 10 jan. 2003]. Disponível em: http://www.ibiblio.org/xml/books/xmljava/.

HARPIN, Tom. *Using java.lang.reflect.Proxy to Interpose on Java Class Methods*. 2001. [acessado em 10 dez. 2002]. Disponível em: http://developer.java.sun.com/developer/technicalArticles/JavaLP/Interposing/.

KICZALES, Gregor; ASHLEY, J. Michael et al. *Metaobject Protocols: Why we want them and what else they can do*. In: *Object-Oriented Programming: The CLOS Prospective*, p. 101-118, Andreas Paepcke, Ed., MIT Press, Cambridge, MA, EUA, 1993. Disponível em: http://www.parc.xerox.com/spl/groups/eca/pubs/papers/Kiczales-Andreas-MOP/for-web.pdf

KICZALES, Gregor; LAINPING, J.; MENDHEKAR, A. et al. *Aspect-Oriented Programming*. In: 11th European Conference on Object-Oriented Programming (ECOOP 97), (jun. 1997, Jyväskylä, Finland), p. 220–242.

LODIN, Steve W.; SCHUBA, Christoph L. *Firewalls fend off invasions from the Net*. IEEE Spectrum, v.35, n.2, p. 26-34, 1998.

LOPES, Cristina V.; HÜRSCH, Walter L. *Separation of Concerns*. 1995. [acessado em 10 mar. 2003]. Disponível em: http://www.ccs.neu.edu/research/demeter/papers/publications-abstracts.html.

LOPES, Cristina V. *D: A Language Framework for Distributed Programming*. 1997. Tese de Doutorado, Northeastern University, Boston, EUA. Disponível em: http://www2.parc.com/csl/groups/sda/publications/papers/PARC-AOP-D97/for-web.pdf.

LUOTONEN, A., ALTIS K.. *World-Wide Web Proxies*. In: First International Conference on the World-Wide Web (mai. 1994, Genebra, Suíça).

MAES, Pattie. *Concepts and Experiments in Computational Reflection*. In: Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87), (dez. 1987, Orlando, EUA), p. 147-155.

MCPHERSON, Scott. *Java Servlets and Serialization with RMI*. 1999. [acessado em 04 fev. 2002]. Disponível em: http://developer.java.sun.com/developer/technicalArticles/RMI/rmi/.

MENDHEKAR, Anurag; FRIENDMAN, Daniel P. *Towards a Theory of Reflective Programming Languages*. In: Informal Proceedings of the Third Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming at OOPSLA'93, (out. 1993, San Francisco, EUA).

MEHTA, Nikunj R.; MEDVIDOVIC, Nenad; PHADKE, Sandeep. *Towards a Taxonomy of Software Connectors*. In, 3rd International Software Architecture Workshop. (nov. 1998, Orlando, EUA).

MILI, Hafedh; MCHEICK; Hamid, SADOU, Salah. *Distribution and* Aspects. In: 1st International Conference on Aspect-Oriented Software Development (abr. 2002, Ensched, The Netherlands).

OLIVA, Alexandre, GARCIA, Islene C., BUZATO, Luiz E. *The Reflexive Architecture of Guaraná*. Relatório Técnico IC-98-14, Instituto de Computação, Universidade Estadual de Campinas, 1998.

OMG. *OMG IDL to Java Language Mapping*. 2000. [acessado em 04 dez. 2002]. Disponível em: http://www.omg.org/docs/ptc/00-01-08.pdf.

OMG. *CORBA Specification v2.6*. 2001. [acessado em 04 fev. 2002]. Disponível em: http://www.omg.org/cgi-bin/doc?formal/01-12-01.pdf.

OMG. *Standard Minor Exception Codes*. 2002. [acessado em 04 mar. 2003]. Disponível em: http://www.omg.org/docs/omg/02-12-04.txt.

OMG. *Catalog of OMG IDL / Language Mappings Specifications*. 2003. [acessado em 04 fev. 2003] Disponível em: http://www.omg.org/technology/documents/idl2x_spec_catalog.htm..

PARNAS, D. L. *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of ACM, v.15, n.12, p. 1053-1058, 1972. Disponível em: http://www.acm.org/classics/may96/

PRYCE, Nathaniel G. *Component Interaction in Distributed Systems*. 2000. Tese de Doutorado, Universidade de Londres, Londres, Inglaterra. Disponível em: http://www.doc.ic.ac.uk/~np2/phd/thesis.pdf

RINE, David; NADA, Nader; JABER, Khaled. *Using Adapters to Reduce Interaction Complexity in Reusable Component-Based Software Development*. In: Proceedings of SSR'99, ACM Press, 1999.

SEITZ, J.; DAVIES, N.; EBNER, M. et al. *A CORBA-based Proxy Architecture for Mobile Multimedia Applications*. In: Proceedings of the 2nd IFIP/IEEE International Conference on Management of Multimedia Network and Services 98 (MMMS'98), (nov. 1998, Versalhes, França).

SILVA, Antônio R.; ROSA, Francisco A.; GONÇALVES, Teresa et al. *Distributed Proxy: A Design Pattern for the Incremental Development of Distributed Applications*. In: The 4th Conference on Pattern Languages of Programming, PLoP '97, (set. 1997, Monticello, USA).

SHAW, Mary; DELINE, Robert; ZELESNIK, Gregory. *Abstractions and Implementations for Architectural Connections*. In: Proceedings of the 3rd International Conference on Configurable Distributed Systems, (mai.1996, Annapolis, EUA), p. 2-10.

SMITH, B.C. *Reflection and Semantics in Lisp*. In Proceedings of the 11th ACM Symposium on Principles of Programming Languages (POPL), (jan. 1984, Salt Lake City, EUA), p. 23-35.

SOARES, Sérgio; LAUREANO, Eduardo; BORBA, Paulo. *Implementing Distribution and Persistence Aspects with AspectJ*. In: Proceedings of OOPSLA'02, Object Oriented Programming Systems Languages and Applications, (nov. 2002, Washington, EUA).

SOBEL, Jonathan M.; FRIENDMAN, Daniel P. *Reflection Oriented Programming*. In: Proceedings of the Reflection '96 Conference, (abr. 1996, San Francisco, EUA). Disponível em: http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/sobel/sobel.pdf.

SUN Microsystems. *Java Core Reflection*. 1998. [acessado em 10 set. 2002]. Disponível em: http://java.sun.com/j2se/1.4.1/docs/guide/reflection/spec/java-reflectionTOC.doc.html.

SUN Microsystems. *Dynamic Proxy Classes*. 1999. [acessado em 10 nov. 2002]. Disponível em: http://java.sun.com/j2se/1.4.1/docs/guide/reflection/proxy.html.

SUN Microsystems. *Java Object Serialization Specification*. 2001. [acessado em 07 nov. 2002]. Disponível em: http://java.sun.com/j2se/1.4.1/docs/guide/serialization/spec/serialTOC.doc.html.

SUN Microsystems. *Java Remote Method Invocation Specification*. 2002. [acessado em 06 jul. 2002]. Disponível em: ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf.

SUN Microsystems. *Java IDL Technology Documentation*, 2002. [acessado em 06 jul. 2002]. Disponível em: http://java.sun.com/j2se/1.4/docs/guide/idl/.

SUN Microsystems. *Java API for XML-based RPC - JAX-RPC 1.0*. 2002. [acessado em 16 out. 2002]. Disponível em: http://java.sun.com/xml/downloads/jaxrpc.html.

SZYPERSKI, Clemens. *Component Software – Beyond Object-Oriented Programming*. ACM Press, 1999.

TARR, Peri.; D'HONDT, Maja, BERGMANS, Lodewijk et al. *Workshop on Aspects and Dimensions of Concern: Requirements on, and Challenge Problems For, Advanced Separation Of Concerns*. 2000. [acessado em 16 out. 2002]. Disponível em: http://www.research.ibm.com/hyperspace/workshops/icse2001/.

TATSUBORI, Michiaki. *An Extension Mechanism for the Java Language*. 1999. Dissertação de Mestrado, Universidade de Tsukuba, Ibaraki, Japão. Disponível em: http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/papers/mich_thesis99.pdf.

TATSUBORI, Michiaki; CHIBA, Shigeru; KILLIJIAN, Marc-Olivier et al. *OpenJava: A Class-Based Macro System for Java*. 2000. [acessado em 27 jan. 2003]. Disponível em: http://www.csg.is.titech.ac.jp/~mich/pub/200007_lncs1826.pdf.

USERLAND. *XML-RPC Home Page*. [acessado em 27 jan. 2003]. Disponível em: http://www.xmlrpc.com/.

W3C. *Extensible Markup Language (XML) 1.0 (Second Edition)*. 2000. [acessado em 06 jul. 2002]. Disponível em: http://www.w3.org/TR/2000/REC-xml-20001006.

W3C. *SOAP Version 1.2 Part 0: Primer*. 2001. [acessado em 06 jul. 2002]. Disponível em: http://www.w3.org/TR/soap12-part0/.

W3C. *SOAP Version 1.2 Part 1: Messaging Framework – Working Draft*. 2001. [acessado em 12 jul. 2002]. Disponível em: http://www.w3.org/TR/soap12-part1/.

W3C. *SOAP Version 1.2 Part 2: Adjuncts - Working Draft. 2002*. [acessado em 12 jan. 2003]. Disponível em: http://www.w3.org/TR/soap12-part2/.

WALDO, J.; WYANT, G.; WOLLRATH, A. et al. *A Note on Distributed Computing*. 1994. [acessado em 16 out. 2002]. Disponível em: http://research.sun.com/research/techrep/1994/smli_tr-94-29.pdf.