Uma Comparação entre Diversas Tecnologias de Comunicação de Objetos Distribuídos em Java

Carlos Eduardo Calabrez

Trabalho Final de Mestrado Profissional

Instituto de Computação
Universidade Estadual de Campinas

Uma Comparação entre Diversas Tecnologias de Comunicação de Objetos Distribuídos em Java

Carlos Eduardo Calabrez

Fevereiro de 2004

Banca Examinadora:

- **Prof. Dr. Alfredo Goldman vel Lejbman (Orientador)**
Departamento de Ciência da Computação, IME, USP

- **Prof. Dr. Marco Dimas Gubitoso**
Departamento de Ciência da Computação, IME, USP

- **Prof. Dr. Edmundo R. M. Madeira**
Instituto de Computação, UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Calabrez, Carlos Eduardo

C125c Uma comparação entre diversas tecnologias de comunicação de objetos distribuídos em Java / Carlos Eduardo Calabrez -- Campinas, [S.P. :s.n.], 2004

Orientador: Alfredo Goldman vel Lejbman

Dissertação (Mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Sistemas operacionais distribuídos (computadores). 2.Programação orientada a objetos. 3.Java (linguagem de programação de computador). 4.CORBA (linguagem de programação de computador) I. Lejbman, Alfredo Goldman Vel. II. Garcia, Islene Calciolari. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Uma Comparação entre Diversas Tecnologias de Comunicação de Objetos Distribuídos em Java

Este exemplar corresponde à redação final do Trabalho Final devidamente corrigida e defendida por Carlos Eduardo Calabrez e aprovada pela BancaExaminadora.

Campinas, Fevereiro de 2004.

Prof. Dr. Alfredo Goldman vel Lejbman
(Orientador)
Profa. Dra. Islene Calciolari Garcia
(Co-orientadora)

Trabalho Final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Computação na área de Engenharia de Computação

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 19 de fevereiro de 2004, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Marco Dimas Gubitoso
DCC – IME - USP

Prof. Dr. Edmundo Roberto Mauro Madeira
IC - UNICAMP

Prof. Dr. Alfredo Goldman Vel Lejbman
DCC – IME - USP

Resumo

Este trabalho apresenta diversas tecnologias de comunicação entre objetos distribuídos em Java: RMI, RMI-IIOP, CORBA (JavaIDL e JacORB) e JAX-RPC (SOAP). Suas arquiteturas e suas principais características são descritas, assim como exemplos contendo os passos principais para a sua utilização.

Em seguida são apresentadas comparações detalhadas do desempenho dessas tecnologias em cenários distintos. Estudos similares demonstraram que RMI obtém um bom desempenho em muitos casos. Este trabalho confirma essa tendência e verifica a diferença de desempenho ao se utilizar as demais tecnologias.

Abstract

This work presents several communication technologies for distributed objects in Java: RMI, RMI-IIOP (JavaIDL and JacORB) and JAX-RPC (SOAP). We describe their architectures and main characteristics, and we also present practical examples of their usage.

We complete the work with detailed comparisons among these technologies on different scenarios. From previous works we already know that RMI generally outperforms the other technologies, but we also want to verify the performance ratio of the other technologies.

Agradecimentos

À minha esposa e companheira, Cristiane, pela paciência e apoio nesses anos de mestrado.

Ao Professor Alfredo Goldman pela orientação, compreensão e apoio na realização deste trabalho. E à Professora Islene Calciolari Garcia pela co-orientação recebida.

À Área de Pesquisa e Desenvolvimento do setor PCS da Motorola Industrial Ltda pelo patrocínio e apoio na realização deste trabalho.

À minha mãe por toda a sua dedicação na minha formação, à minha irmã por estar sempre ao meu lado, e a meu pai e a Deus, que ouvem minhas orações e sempre me amparam.

Aos pais da minha esposa, que me ajudaram em todos os aspectos desde que vim para a cidade de Campinas.

Conteúdo

Lista de Figuras

Lista de Gráficos

Capítulo 1

Introdução

Sistemas distribuídos consistem em coleções de computadores autônomos conectados por uma rede de comunicação e equipados com software que habilita esses computadores a se integrarem e compartilharem os recursos (hardware, processamento e dados) dos seus sistemas [1].

Sistemas distribuídos têm se desenvolvido muito nos últimos anos, assim como sua utilização tem crescido consideravelmente. Os principais fatores que contribuem para esse crescimento são a evolução da capacidade de processamento dos computadores assim como as crescentes taxas de transmissão das redes de comunicação.

As principais características dos sistemas distribuídos tornam sua utilização muito atraente: compartilhamento de recursos, facilidade de adição de recursos (extensão), concorrência, escalabilidade, tolerância a falhas e transparência.

Para o compartilhamento de recursos, existem basicamente dois modelos: o orientado a objetos e o orientado a serviços. Nos dois modelos os recursos são gerenciados por servidores. Os clientes se comunicam com os servidores para requisitar um determinado serviço. No modelo orientado a objetos, cada recurso é representado por um objeto e a requisição de um serviço se dá pela chamada de um método do respectivo objeto (local ou remoto). No modelo orientado a serviços, um objeto representa os serviços providos por um servidor (ao invés de recursos específicos). O modelo baseado em objetos é mais genérico e vem sendo cada vez mais utilizado devido à disponibilidade de diferentes plataformas de desenvolvimento.

Tanto o modelo orientado a objetos quanto o modelo orientado a serviços evoluíram a partir de um conceito denominado RPC (*Remote Procedure Calling* ou Chamada de Procedimento Remoto [2]), utilizado inicialmente com linguagens de programação estruturada.

Este trabalho apresenta as principais tecnologias de programação distribuída disponíveis para a programação na linguagem Java. A linguagem Java foi escolhida pois é

a linguagem de programação orientada a objetos que mais tem crescido em utilização e importância nos últimos anos, principalmente para as aplicações desenvolvidas para a Web.

O objetivo deste trabalho é descrever as principais tecnologias de programação disponíveis em Java e apresentar uma comparação de desempenho entre as mesmas.

As tecnologias abordadas por este trabalho são: RMI, RMI-IIOP, CORBA (JavaIDL e JacORB) – baseadas no modelo orientado a objetos – e JAX-RPC (SOAP) – baseada no modelo orientado a serviços. Existem várias diferenças entre essas tecnologias em termos de arquitetura e características. Essas diferenças são fatores determinantes do desempenho e interoperabilidade das tecnologias. Ao se optar pela utilização de determinada tecnologia devem ser levados em conta os fatores mais importantes para a aplicação específica, tais como desempenho e interoperabilidade, ou propriedades específicas, tais como grau de dificuldade de utilização e passagem por *firewalls*.

Este trabalho é organizado da seguinte forma: no capítulo 2 são apresentadas as principais tecnologias de programação distribuída disponíveis para a linguagem Java. Em seguida são apresentadas comparações de desempenho dessas tecnologias, iniciando-se com uma descrição da metodologia utilizada nas comparações (capítulo 3) e os resultados propriamente ditos (capítulo 4). O capítulo 5 contém a conclusão.

Capítulo 2

Tecnologias

As tecnologias de programação de objetos distribuídos aliam os conceitos de programação orientada a objetos com os conceitos de chamadas de procedimentos remotos.

Existem diversos mecanismos de chamada de procedimento remoto (RPC – *Remote Procedure Calling* – [2]). O objetivo desses mecanismos é permitir a execução de procedimentos em máquinas remotas de uma forma conveniente para as linguagens de programação estruturada. A chamada pode ser executada pelo programador de uma forma similar à chamada de um procedimento local. Toda a parte relativa à **comunicação** entre os sistemas envolvidos é realizada pelos mecanismos de RPC. Assim sendo, os mecanismos de RPC facilitam muito o desenvolvimento de aplicações distribuídas.

Uma chamada remota de procedimento pode ser detalhada da seguinte forma:

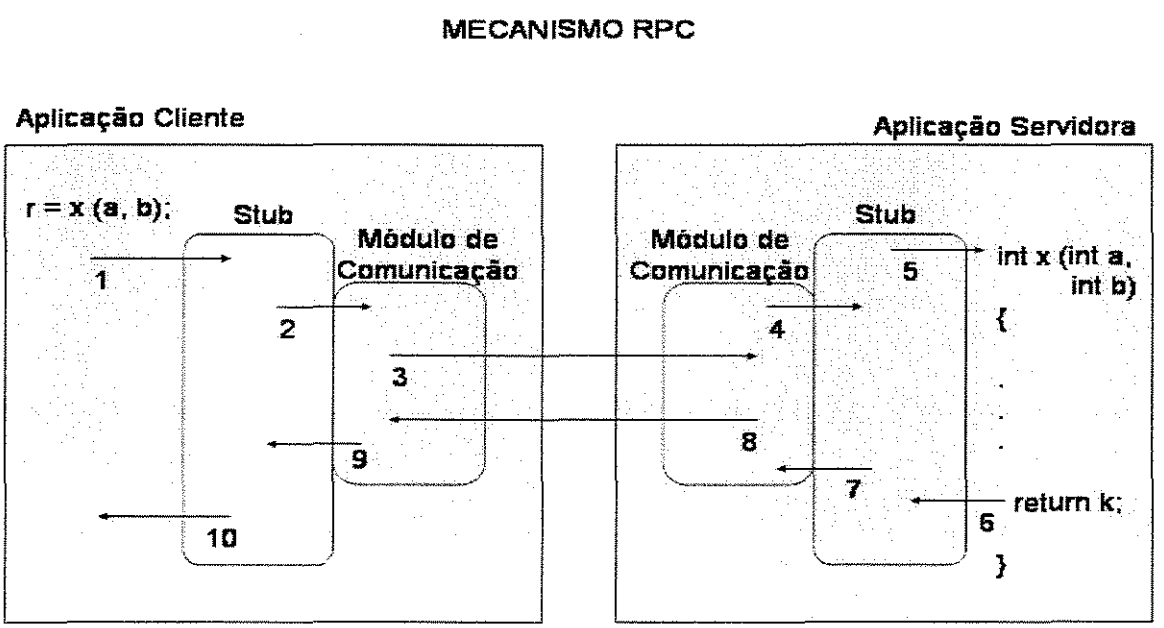

Figura 1 - Mecanismo RPC

1. a aplicação cliente faz a chamada ao procedimento a ser executado remotamente

(como se fosse uma chamada a um procedimento local);

2. um módulo chamado *stub* obtém os dados dessa chamada (identificação do procedimento e respectivos argumentos) e faz o empacotamento desses dados (*marshalling*) em um formato que possibilite sua transmissão através de uma rede de comunicação, independente da plataforma utilizada;
3. o módulo responsável pela comunicação é acionado de forma a fazer o envio de uma mensagem contendo a chamada de procedimento para o servidor apropriado;
4. o módulo de comunicação do lado do servidor recebe a mensagem vinda do cliente e a repassa para o *stub* do servidor (também chamado de *skeleton*);
5. o *stub* do servidor desempacota os dados da chamada (*unmarshalling*) e chama o módulo responsável por identificar o procedimento a ser executado;
6. o procedimento desejado é executado;
7. os resultados da execução do procedimento (código de retorno e argumentos de saída) são empacotados pelo *stub* do servidor e passados para o módulo de comunicação;
8. uma mensagem é então enviada para o cliente, contendo os resultados empacotados;
9. a mensagem é recebida e repassada para o *stub* do cliente;
10. os resultados são desempacotados e devolvidos para a aplicação cliente.

O nível de transparência dos mecanismos de RPC varia de uma implementação para outra. Existem opiniões divergentes quanto ao grau de transparência que deve ser dado ao programador [3]. Como chamadas a procedimentos remotos envolvem a comunicação com uma aplicação em um outro computador na rede, características inerentes a sistemas distribuídos (latência, acesso à memória, falhas parciais e concorrência) devem ser levadas em conta. Algumas implementações fornecem mais recursos para que o programador possa tratar essas características diferentes entre programação local e distribuída, enquanto que outras optam por realizar esse tratamento de forma mais transparente para a aplicação.

Existe um compromisso entre o desempenho e a generalidade de uma implementação de RPC. Por exemplo, em um ambiente homogêneo não seria necessário empacotamento (*marshalling*) e desempacotamento (*unmarshalling*) dos dados das chamadas de procedimento remoto em um formato independente de plataforma e linguagem de programação.

Além da comunicação entre as aplicações, os mecanismos de RPC definem uma forma pela qual a interface dos serviços remotos é descrita. Existem basicamente duas

maneiras para a descrição de interfaces: através de uma linguagem específica para a definição da mesma (IDL - *Interface Definition Language*) ou através da própria linguagem de programação utilizada na implementação do mecanismo de RPC.

As interfaces dos serviços remotos devem conter o nome dos procedimentos disponíveis para execução e os tipos dos respectivos argumentos de entrada, de saída e dos códigos de retorno. A partir da descrição das interfaces dos serviços remotos são gerados os *stubs* do cliente e do servidor, através de um compilador de interfaces, parte integrante dos mecanismos de RPC. No caso dos mecanismos que utilizam uma linguagem própria para a descrição de interfaces, os compiladores de interface podem ser capazes de gerar os *stubs* em mais do que uma linguagem de programação. Dessa forma, além da possibilidade de escolha da linguagem de programação, permite-se também que clientes e servidores escritos em linguagens diferentes se comuniquem através de RPC.

Uma outra funcionalidade importante dos mecanismos de RPC é a de localização de servidor de um determinado serviço, também conhecida como serviço de nomes ou diretório. Essa funcionalidade faz a associação (*binding*) entre o nome de um serviço e a identificação do respectivo servidor (nome da máquina, número do processo, porta de comunicação, etc). Os servidores devem publicar os nomes dos serviços que eles mantém através dessa funcionalidade. Já os clientes utilizam essa funcionalidade para localizar o endereço do servidor de um determinado serviço.

Pode se dizer que plataformas de programação de objetos distribuídos são mecanismos de RPC orientados a objetos. Java [5] é uma tecnologia de programação orientada a objetos que vem se desenvolvendo muito nos últimos anos e é cada vez mais utilizada. Existem diferentes plataformas de programação de objetos distribuídos disponíveis para utilização em Java. A seguir são apresentadas as principais tecnologias existentes e uma comparação entre as suas características principais.

2.1 RMI

RMI – *Remote Method Invocation* [4] – é uma tecnologia de programação de objetos distribuídos que é parte integrante da plataforma Java e está disponível desde a versão 1.1 do Java SDK (*Sun Development Kit*). RMI permite que métodos de objetos remotos sejam chamados da mesma forma que métodos de objetos locais, estendendo o conceito de programação distribuída para a linguagem Java. RMI possibilita a comunicação entre

objetos rodando em máquinas virtuais diferentes, independentemente dessas máquinas virtuais estarem na mesma máquina física ou não.

A arquitetura de RMI é baseada na definição da interface do serviço remoto. Um objeto servidor define o serviço que ele provê através de uma interface, que é usada pelo cliente para fazer uma chamada ao serviço.

Baseando-se na definição de uma interface implementa-se a classe que representa o objeto servidor. A partir da implementação do objeto servidor geram-se as classes do *stub* do cliente e do *skeleton* do servidor, utilizando-se a ferramenta **rmic** (RMI *Compiler*) – compilador de interfaces de Java RMI.

Tanto a implementação do objeto servidor quanto o *stub* do cliente implementam a interface do serviço remoto. Quando o cliente executa uma chamada a um método da interface, o *stub* do cliente é invocado. Em seguida o *stub* do cliente se comunica com o *skeleton* do servidor para que a chamada da implementação real do método seja invocada.

A partir da versão 1.2 do Java SDK, *skeletons* não são mais necessários. Um protocolo adicional de *stubs* foi introduzido do lado do servidor, responsável pela funcionalidade realizada anteriormente pelos *skeletons*. Entretanto, *skeletons* ainda podem ser gerados se for necessária a compatibilidade com versões anteriores de SDK.

2.1.1 Arquitetura

A arquitetura do sistema RMI é dividida em camadas, de forma a facilitar sua expansão e inclusão de novas funcionalidades:

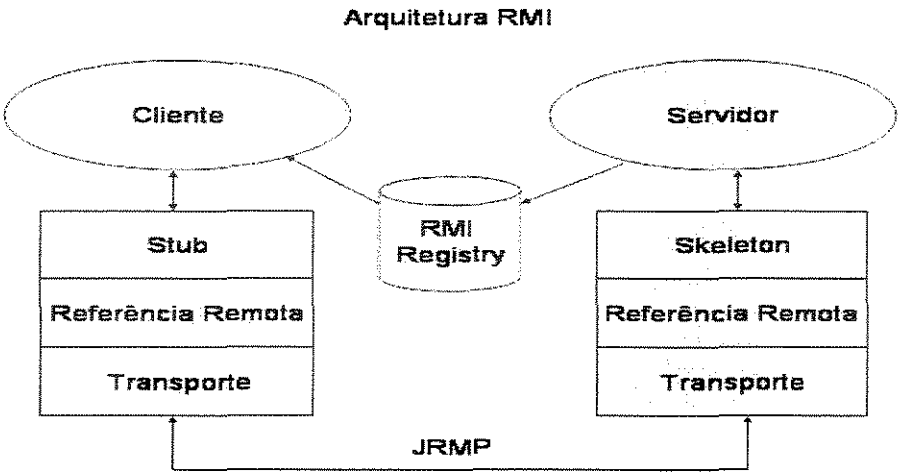

Figura 2 - Arquitetura RMI

- **Camada de *Stubs & Skeletons***: essa camada reside logo abaixo da camada da aplicação. Como seu próprio nome indica, ela inclui o *stub* do cliente e o *skeleton* do servidor. O *stub* do cliente é responsável por empacotar (*marshal*) o nome do método remoto e os argumentos da chamada e repassar a chamada ao objeto servidor, funcionando como um *proxy* do objeto remoto. Os serviços da camada inferior (Referência Remota) são utilizados para realizar essa função. O *skeleton* do servidor é responsável por receber a chamada vinda do cliente, desempacotá-la (*unmarshalling*) e chamar o método remoto propriamente dito. Após a execução do método remoto, o *skeleton* empacota os dados a serem devolvidos (argumentos de saída, código de retorno do método ou exceção levantada durante a execução) e os envia para o cliente.
- **Camada de Referência Remota**: do lado do cliente, essa camada é responsável pela criação de uma referência para o objeto da máquina remota (estabelecendo uma conexão com o objeto) e por disponibilizar os mecanismos para a invocação de métodos remotos – utilizando os serviços da camada inferior (Transporte). Do lado do servidor, essa camada recebe os dados da chamada e os repassa a chamada para o *skeleton* do servidor. A camada de referência remota é responsável por todas as semânticas envolvidas no estabelecimento de uma conexão, tais como ativação de objetos remotos dormentes, gerenciamento de tentativas de conexão (*conection retry*), etc.

- **Camada de Transporte**: responsável pela conexão entre as máquinas virtuais do cliente e do servidor, e pelo transporte das mensagens entre as mesmas. Essa camada utiliza o protocolo JRMP (*Java Remote Method Protocol*) sobre TCP/IP para a comunicação entre as máquinas virtuais. JRMP é um protocolo proprietário da Sun baseado em fluxos de bytes (*streams*). O protocolo HTTP também pode ser utilizado para encapsular as mensagens do protocolo JRMP quando existe alguma máquina *firewall* entre o cliente e o servidor.

2.1.2 Serviço de Localização de Servidor (Serviço de Nomes ou Diretório)

Para invocar um método remoto, é necessária uma referência para um objeto remoto. Uma referência pode ser obtida como código de retorno de uma chamada a outro método remoto ou através do serviço de localização de servidor (serviço de nomes ou diretório). No caso da utilização desse serviço, o cliente precisa apenas executar uma busca passando o nome do serviço remoto em formato URL:

```
[rmi://<máquina>[:porta de comunicação]/]<nome do serviço>
```

O servidor deve publicar o nome do serviço após a criação do objeto que o implementa.

RMI possui um serviço de localização de servidor específico chamado *RMI Registry*, mas outros serviços podem ser usados, tais como JNDI – *Java Naming and Directory Interface*.

2.1.3 Características

A seguir são listadas algumas das principais características de Java RMI:

- **Seriação de objetos (*Object Serialization*)**: RMI usa seriação de objetos [6] para transformar tipos primitivos e objetos em um formato apropriado para transmissão de uma máquina virtual para outra (empacotamento dos dados ou *marshalling*).

- **Passagem de parâmetros:** em Java parâmetros são sempre passados por valor se as chamadas de métodos ocorrem dentro da mesma máquina virtual, tanto para

tipos primitivos quanto para referências a objetos. O mesmo é válido para códigos de retorno de métodos. Em RMI, em uma chamada a um método remoto (ou seja, um método de um objeto de uma outra máquina virtual), os seguintes tipos de passagem de parâmetros (e códigos de retorno de métodos) podem ocorrer:

- tipos primitivos são passados por valor;
- objetos locais são passados por valor, assim como todos os objetos referenciados pelo mesmo - ou seja, todo o grafo de dependência a partir do objeto é seriado, formando uma estrutura de dados auto-contida (sem referências a endereços de memória locais), para ser transmitido de uma máquina virtual para outra;
- objetos remotos são passados por referência – entretanto, uma referência a um objeto remoto é convertida para uma referência para o respectivo *stub*.

- **Polimorfismo distribuído**: RMI não trunca nenhum tipo de dado passado de uma máquina virtual para outra, ou seja, RMI provê polimorfismo distribuído. Por exemplo, se um objeto de uma classe A é esperado como parâmetro de um método e ao invés disso é passado um objeto de uma classe B, subclasse de A, a definição completa do objeto é enviada e não apenas a parte que corresponde à classe A.

- **Carga Dinâmica de classes/*stubs* (*dynamic class/stub download*)**: se uma classe de um objeto passado como parâmetro ou devolvido por um método não estiver carregada na máquina virtual, RMI tenta carregar a implementação da classe localmente. Caso a implementação não esteja disponível na máquina local, RMI tenta obtê-la a partir da localização especificada pela propriedade *codebase* (`java.rmi.server.codebase` – lista de URLs), passada juntamente com o objeto na chamada ou retorno do método [7]. O mesmo procedimento é válido quando um cliente obtém uma referência para um objeto remoto e o respectivo *stub* não está disponível na máquina local.

- **Coleta de lixo distribuída**: RMI estende o conceito de coleta de lixo para programação distribuída. Entretanto, não é possível precisar se um objeto remoto não tem mais nenhuma referência ou se as conexões com os clientes que o referenciam foram interrompidas. Sendo assim, RMI considera que um objeto remoto é candidato para coleta de lixo distribuída quando ele não é acessado após

um determinado período de tempo.

- **Segurança**: RMI utiliza o mesmo mecanismo usado em Java para segurança, baseado na utilização de objetos gerenciadores de segurança (classe `SecurityManager`). Um objeto `SecurityManager` é necessário em qualquer máquina virtual que precise obter a implementação de uma classe remotamente. O `SecurityManager` define quais operações podem ser executadas por objetos de classes obtidas remotamente.

- **Ativação de objetos remotos**: RMI provê mecanismos para que objetos remotos sejam criados sob demanda, ou seja, apenas quando um cliente tentar acessar um de seus métodos. Alguns tutoriais nas páginas de RMI contêm maiores detalhes sobre as formas de ativação de objetos remotos: [11]

2.1.4 Exemplo

O seguinte exemplo apresenta os passos mínimos necessários para a utilização de RMI em uma aplicação. Os tutoriais [8] e [9] contêm maiores detalhes. Nas páginas da Sun também é possível encontrar um mini-curso sobre RMI [10].

Em primeiro lugar é necessário definir a interface do objeto remoto. Para isto, basta implementar uma interface pública que estenda a classe `java.rmi.Remote` e cujos métodos lancem exceções do tipo `java.rmi.RemoteException`. Objetos que sejam passados como parâmetros de métodos ou código de retorno devem implementar a interface `java.io.Serializable`.

```
package exemplo_rmi;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface XInt extends java.rmi.Remote
{
  // Método a ser executado remotamente
  public String metodoRemoto(int i) throws java.rmi.RemoteException;
}
```

Em seguida é necessário implementar o objeto remoto. A forma mais simples é criar uma classe que estenda a classe `java.rmi.UnicastRemoteObject`, defina um construtor público sem parâmetros que lance uma exceção `java.rmi.RemoteException` e

implemente a interface do objeto remoto (arquivo `XImpl.java`):

```
package exemplo_rmi;

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

import exemplo_rmi.XInt;

public class XImpl extends java.rmi.UnicastRemoteObject implements XInt
{
  // Construtor
  public XImpl() throws java.rmi.RemoteException
  {
    ...
  }

  // Método a ser executado remotamente
  public String metodoRemoto(int i) throws java.rmi.RemoteException
  {
    ...
  }
}
```

O próximo passo é implementar o servidor que hospeda o objeto remoto. As três tarefas principais que o servidor precisa executar são: criação e instalação de um gerenciador de segurança, criação do objeto remoto e registro do objeto remoto:

```
package exemplo_rmi;

import java.rmi.Naming;
import java.rmi.RMISecurityManager;

import exemplo_rmi.XImpl;

public class XServer
{
  public static void main(String args[])
  {
    ...

    try {
    // 1 - Criação e instalação de um gerenciador de segurança
    System.setSecurityManager(new RMISecurityManager());

    // 2 - Criação do objeto remoto
    XImpl x;
    x = new Ximpl();

    // 3 - Registro do objeto remoto no RMI Registry
    //   Neste exemplo registryHost é uma variável do tipo String
    //   contendo o nome do servidor que hospeda o RMI Registry e
    //   port é uma variável do tipo String que contém o número da
    //   porta de comunicação utilizada pelo RMI Registry.
```

```
    Naming.rebind("//" + registryHost + ":" +port + "/" + "Xserver", x);

    ...
    }

    catch (RemoteException e) {
      ...
    }
    ...
  }
}
```

Durante a criação e instalação do gerenciador de segurança, informações sobre a política de segurança são necessárias. Tal política é definida através de arquivos específicos, configurados através da propriedade `java.security.policy`, que requer a URL do arquivo de política de segurança.

Para um cliente chamar um método de um objeto remoto, basta obter uma referência para o objeto remoto (neste exemplo o RMI *Registry* é utilizado) e executar a chamada propriamente dita:

```
package exemplo_rmi;

import java.rmi.Naming;
import java.rmi.RemoteException;

import exemplo_rmi.XInt;

public class XClient
{
  public void algumMetodo()
  {
    ...

    try {

    // 1 - Obtenção da referência para o objeto remoto
    //   Neste exemplo registryHost é uma variável do tipo String
    //   contendo o nome do servidor que hospeda o RMI Registry e
    //   port é uma variável do tipo String que contém o número da
    //   porta de comunicação utilizada pelo RMI Registry.
    XInt x;
    x = (Xint)Naming.lookup("//" + registryHost +
                            ":" +port + "/" + "Xserver", x);

    // 2 - Chamada ao método do objeto remoto
    s = x.metodoRemoto(10);

    ...
    }
    catch (RemoteException e) {
      ...
```

```
    }
    ...
  }
}
```

Em seguida é possível gerar o *stub* do cliente e o *skeleton* do servidor a partir da classe que implementa a interface remota (`XImpl.java`):

```
# rmic XImpl
```

O comando acima gera as classes `XImpl_Stub.class` e `XImpl_Skel.class`.

Como mencionado anteriormente, os *skeletons* dos servidores não são mais necessários nas últimas versões de Java SDK. Para gerar código compatível apenas com Java 1.2 ou mais recente (ou seja, apenas a classe `XImpl_Stub.class`, *stub* que é utilizado tanto pelo cliente como pelo servidor), basta executar:

```
# rmic -v1.2 XImpl
```

Após a compilação dos arquivos contendo a interface, sua implementação, o servidor e o cliente, é possível rodar o exemplo, começando pela ativação do RMI *Registry* (através da execução da ferramenta `rmiRegistry`) e então executando-se o servidor seguido do cliente.

2.2 CORBA

CORBA – *Common Object Request Broker Architecture* [13], [14], [15], [16] – é uma arquitetura para a programação de objetos distribuídos que especifica o componente ORB (*Object Request Broker*) da OMA – *Object Management Architecture*. A organização responsável pela criação, definição e evolução da OMA é a OMG – *Object Management Group* [12], um consórcio formado por aproximadamente 800 membros, dentre eles empresas de pequeno e grande porte, universidades e institutos de pesquisa. É importante destacar que a OMG é responsável pela especificação da OMA e de seus componentes, não pela sua implementação.

Existem diversas implementações de CORBA disponíveis no mercado: produtos comerciais – tais como Orbix da IONA [17] e Visibroker da Borland [18] – e implementações abertas – por exemplo, Java IDL [19] e JacORB [20].

Assim como RMI, o modelo adotado por CORBA é baseado no modelo de objetos, onde os serviços disponibilizados por objetos servidores são acessados através de uma interface bem definida pelos objetos clientes. Para a definição de interfaces, CORBA especifica uma linguagem denominada OMG IDL (*Interface Definition Language*) e mapeamentos dessa linguagem para determinadas linguagens de programação, tais como C, C++, Java, Smalltalk e COBOL. A partir da definição da interface de um objeto em IDL, um compilador IDL é utilizado para a geração do *stub* do cliente e do *skeleton* do servidor.

As aplicações cliente e servidor podem ser desenvolvidas utilizando esses *stubs* e *skeletons*. Entretanto, CORBA provê mecanismos para chamadas remotas de procedimento nos quais cliente e servidor não necessitam ter conhecimento da definição das interfaces em tempo de compilação.

Implementações da arquitetura CORBA como as mencionadas acima normalmente incluem implementações do elemento Serviços de Objetos (*Object Services*) da arquitetura OMA tais como Serviço de Nomes (*Name Service*), Serviço de Eventos (*Event Service*), etc.

2.2.1 Arquitetura OMA

A arquitetura OMA é baseada no Modelo de Objetos (Object Model) e no Modelo de Referência (Reference Model). O Modelo de Objetos é um modelo cliente/servidor, no qual o objeto servidor provê serviços aos objetos clientes. Esses serviços são chamados através de operações definidas na interface do objeto servidor.

O Modelo de Referência define os seguintes componentes, assim como suas interfaces e as interações entre os mesmos:

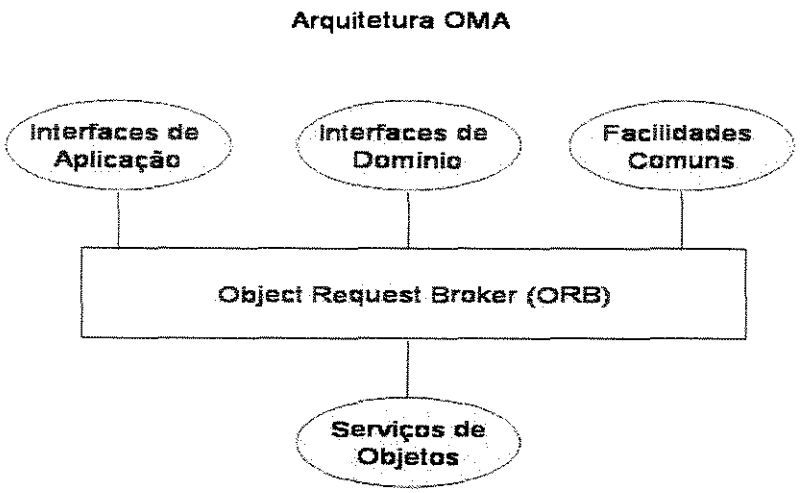

Figura 3 - Arquitetura OMA

- **ORB (*Object Request Broker*):** é o componente central da arquitetura OMA, responsável pela comunicação entre objetos distribuídos. A ORB provê transparência de localização, ativação e independência de linguagem de programação entre cliente e servidor.

- **Serviços de Objetos (*Object Services*):** esse componente define os serviços básicos utilizados por quaisquer aplicações: nomes, eventos, ciclo de vida, persistência, transações, controle de concorrência, relacionamento, externalização, busca, propriedades, segurança, entre outros.

- **Facilidades Comuns (*Common Facilities*):** são interfaces orientadas para aplicações de usuário, que podem ser utilizadas por aplicações de domínios distintos: interface com o usuário, gerenciamento de informação, gerenciamento de sistemas e gerenciamento de tarefas.

- **Inferfaces de Domínio (*Domain Interfaces*):** são interfaces orientadas para domínios específicos, tais como: finanças, medicina, manufatura e telecomunicações.

- **Interfaces de Aplicação (*Application Interfaces*):** são interfaces não padronizadas desenvolvidas para as aplicações específicas. Essas interfaces utilizam os serviços dos demais componentes.

2.2.2 Arquitetura CORBA

A arquitetura CORBA especifica as interfaces e os serviços que o componente ORB da arquitetura OMA deve fornecer. CORBA define os seguintes componentes:

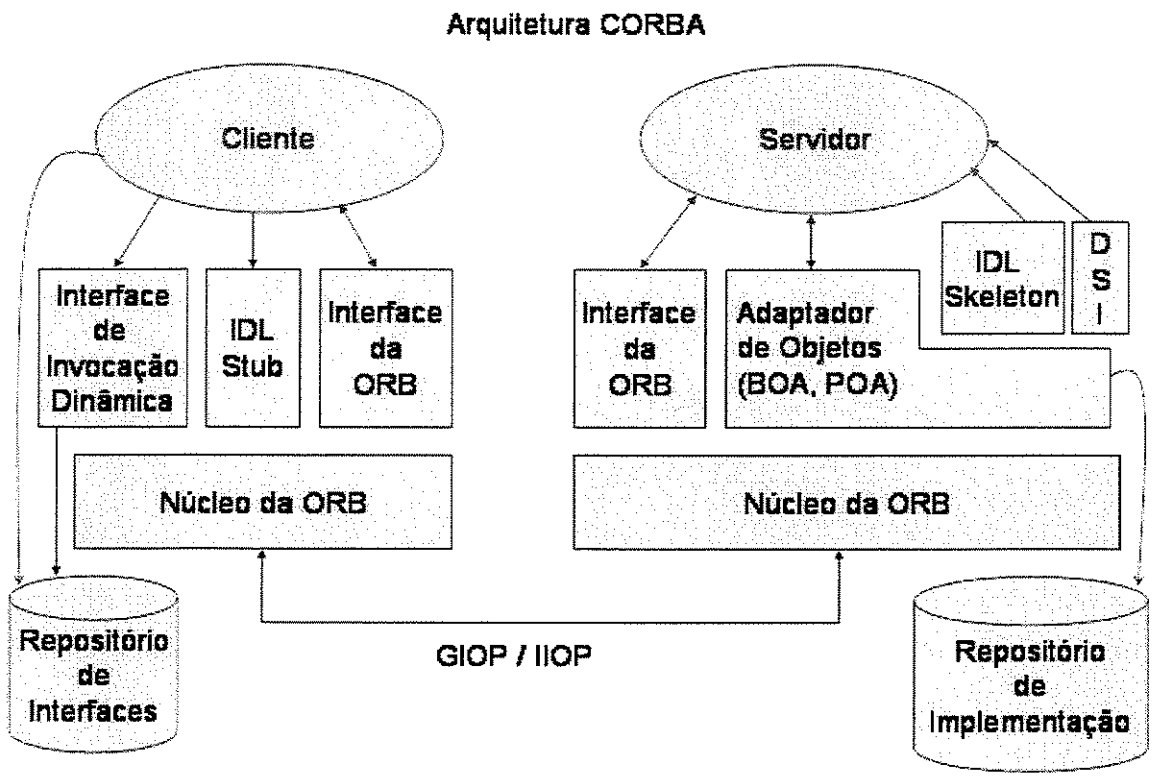

Figura 4 - Arquitetura CORBA

- **Núcleo da ORB (ORB *Core*):** responsável pela comunicação entre clientes e servidores, provê mecanismos para a chamada a métodos de objetos remotos, implementando transparência de localização e de ativação de servidores, assim como independência de linguagem de programação e de plataforma de hardware.

- **OMG IDL (*Interface Definition Language*):** é uma linguagem declarativa utilizada para a especificação das operações e atributos de objetos CORBA. IDL

permite também a definição de módulos, contendo definições de diversos objetos. A OMG especifica mapeamentos de IDL para algumas linguagens de programação, implementados por compiladores IDL.

- **IDL *Stub*:** gerados por compiladores IDL, *stubs* são utilizados por aplicações cliente para chamadas a métodos de servidores remotos. Os *stubs* são responsáveis pela criação de chamadas e pelo envio das mesmas ao servidor, através da ORB. No retorno das chamadas os *stubs* repassam os argumentos de saída e códigos de retorno recebidos para a aplicação cliente.

- **IDL *Skeleton*:** gerados por compiladores IDL, *skeletons* são utilizados por aplicações servidoras no tratamento de chamadas de métodos remotos. Os *skeletons* recebem as chamadas de métodos através da ORB e as repassam para o objeto destino. *Skeletons* também são responsáveis por enviar os argumentos de saída e códigos de retorno dos métodos para a aplicação cliente.

- **Adaptadores de Objetos (*Object Adapters*):** responsáveis por prover os mecanismos e interfaces necessários para que implementações de objetos CORBA utilizem os serviços da ORB. Sua funcionalidade inclui os seguintes serviços: registro de objetos, geração de referências a objetos, ativação e desativação de objetos e servidores, e registro de implementações. A arquitetura CORBA atualmente especifica dois tipos de adaptadores:
    - **BOA (*Basic Object Adapter*)** – primeiro adaptador especificado pela OMG, de definição muito genérica em algumas áreas, foi implementado de formas incompatíveis por diferentes produtos, dificultando a portabilidade das implementações de objetos CORBA;
    - **POA (*Portable Object Adapter*)** – criado para resolver os problemas de portabilidade entre implementações de adaptadores de diferentes produtos. Inclui também algumas funcionalidades adicionais: suporte a objetos transientes e persistentes, ativação de objetos explícita e sob demanda, especificação de políticas de *multithreading*, segurança e gerenciamento de objetos, entre outras.

- **Interface para Invocação Estática (SII - *Static Invocation Interface*):** é a invocação de métodos remotos através da utilização de *stubs* de cliente e *skeletons* de servidor, gerados pelo compilador IDL, ou seja, toda a informação

necessária para a chamada é conhecida em tempo de compilação. É importante notar que a utilização de *stubs* pelas aplicações cliente não implica na utilização de *skeletons* nos servidores e vice-versa, ou seja, interfaces dinâmicas (descritas a seguir) podem ser utilizadas independentemente no cliente e no servidor.

- **Interface para Invocação Dinâmica (DII - *Dynamic Invocation Interface*):** é a invocação de métodos remotos pela qual as informações a respeito dos mesmos (nome, parâmetros e respectivos tipos) é obtida em tempo de execução através da utilização do Repositório de Interfaces. Dessa forma uma aplicação cliente pode invocar os métodos de quaisquer objetos CORBA, sem necessidade de conhecer suas interfaces em tempo de compilação.

- **Interface de Esqueleto Dinâmica (DSI - *Dynamic Skeleton Interface*):** é o equivalente a DII, mas do lado do servidor. Esse mecanismo permite a implementação de servidores capazes de tratar chamadas de métodos sem a necessidade de se utilizar *skeletons* gerados por compiladores IDL.

- **Repositório de Interfaces (*Interface Repository*):** é um repositório que provê informações sobre interfaces de objetos remotos registrados. Permite que aplicações cliente e servidor obtenham definições de interfaces de objetos CORBA em tempo de execução.

- **Repositório de Implementação (*Implementation Repository*):** é um repositório que contém informações utilizadas pela ORB para localizar e ativar objetos servidores.

- **Protocolo GIOP (*Generic Inter ORB Protocol*):** é um protocolo de comunicação que garante a interoperabilidade entre implementações de ORBs distintas. Esse protocolo define as mensagens e o formato de representação dos dados para a comunicação entre ORBs. A implementação de GIOP sobre TCP/IP é chamada de IIOP (*Internet Inter-ORB Protocol*).

2.2.3 Características

CORBA é uma tecnologia que se desenvolveu ao longo de vários anos [21]. Entre as principais características de CORBA pode-se citar:

- **Independência de linguagem de programação:** a utilização de uma linguagem específica para a definição de interfaces (OMG IDL) e a existência de mapeamentos dessa linguagem para diferentes linguagens de programação possibilitam a comunicação entre aplicações cliente e servidor escritas em linguagens distintas.

- **Independência de plataforma de hardware:** proporcionada pela disponibilidade de implementações de CORBA em plataformas distintas.

- **Interoperabilidade entre implementações distintas:** a utilização do protocolo GIOP/IIOP permite que aplicações cliente e servidor desenvolvidas utilizando-se implementações distintas de CORBA se comuniquem transparentemente.

- **IOR (*Interoperable Object Reference*):** é um formato padrão para referências a objetos, necessário para a interoperabilidade entre diferentes implementações de ORBs. Contém informações que permitem que uma ORB localize e se comunique com objetos remotos.

- **Invocação dinâmica de métodos:** realizada através da utilização do repositório de interfaces e da interface para invocação dinâmica.

- **Modos de passagem de parâmetros:** OMG IDL permite que se especifique parâmetros de entrada (`in`), de saída (`out`) ou de entrada e saída (`inout`).

- **Objetos passados por valor (*objects by value*):** possibilidade de passar e devolver objetos por valor em uma chamada remota de procedimento, através da utilização do elemento `valuetype` de OMG IDL.

- **Modos de invocação de métodos remotos:**
    - **síncrono**: a aplicação cliente fica bloqueada esperando a resposta da invocação do método remoto;
    - **síncrono postergado (*deferred synchronous*)**: a aplicação cliente faz a chamada e não é bloqueada. Para obter a resposta a aplicação pode fazer uma chamada específica para verificar se a resposta já está disponível (e então recebê-la) ou bloquear a sua execução enquanto a resposta não é

recebida. Esse modo só pode ser utilizado com invocação dinâmica, pois a especificação para a geração de interfaces de invocação estática através de *stubs* não contempla esse modo de invocação;

- **sem resposta (*one-way*)**: nesse modo não existe resposta. A aplicação cliente invoca o método remoto e a ORB faz o possível para que a chamada seja executada, mas não há garantia (*best effort*);
- **assíncrono com *callback***: a aplicação cliente não é bloqueada ao fazer a chamada ao método remoto. Nesse modo uma referência a um objeto é passada como parâmetro adicional, utilizada pela ORB para a entrega da resposta;
- **assíncrono com *polling***: este modo é similar ao modo síncrono postergado. A aplicação cliente faz a chamada ao método remoto e recebe imediatamente no retorno da chamada um objeto devolvido por valor (`valuetype`). Esse objeto pode ser utilizado posteriormente para verificar se a resposta já está disponível ou para bloquear a execução da aplicação enquanto a resposta não é recebida.

Os modos assíncronos de invocação foram especificados posteriormente [21], juntamente com alterações na especificação da geração de *stubs*, podendo ser utilizados com invocação estática de métodos remotos.

- **Persistência:** CORBA permite que a criação de objetos transientes, cujo tempo de vida é associado ao tempo de vida da aplicação que os hospedam, e objetos persistentes, que não possuem essa limitação. Maiores detalhes sobre o uso de servidores transientes e persistentes em Java IDL podem ser encontrados em [22].

- **Tipo genérico *any*:** é um tipo abstrato de dados de OMG IDL que pode conter valores de qualquer outro tipo OMG IDL.

- **Segurança:** a OMG especifica diferentes mecanismos para a configuração e utilização de segurança em CORBA. Esses mecanismos provêm controle de acesso, autenticação, comunicação segura, etc. Maiores detalhes podem ser encontrados em [23].

- CORBA não possui nenhum mecanismo de coleta de lixo distribuída, basicamente porque nem todas as linguagens de programação que podem ser utilizadas possuem essa característica.

2.2.4 Exemplos

Como mencionado anteriormente, existem diversas implementações da arquitetura CORBA disponíveis. Neste trabalho foram utilizadas as implementações abertas Java IDL [19] e JacORB [20]. A próxima seção contém os passos mínimos necessários para a implementação de uma aplicação distribuída em CORBA utilizando-se Java IDL. Em seguida são apresentadas as diferenças encontradas ao se utilizar JacORB.

JavaIDL

Para um tutorial introdutório completo sobre Java IDL pode-se consultar [24]. Outros tutoriais assim como toda a documentação de Java IDL está disponível em [25].

O primeiro passo para a implementação de uma aplicação distribuída em CORBA é a definição da interface do objeto remoto em OMG IDL (`X.idl`):

```
module xMod
{
  interface X
  {
    // Método a ser executado remotamente
    string metodoRemoto(int i);
  };
};
```

Em seguida deve-se mapear a definição da interface do objeto remoto para a linguagem de programação desejada. Em Java IDL utiliza-se a ferramenta `idlj`:

```
# idlj -fall X.idl
```

Os arquivos gerados quando se utiliza a opção "`-fall`" são o *stub* do cliente (`_XStub.java`), o *skeleton* do servidor (`XPOA.java`), a interface do objeto remoto (`X.java`) e algumas classes auxiliares. A opção "`-fall`" instrui o compilador `idlj` a gerar tanto os arquivos necessários para a aplicação cliente, quanto os necessários para a aplicação servidora. O adaptador de objetos utilizado é o POA (*Portable Object Adapter*).

O próximo passo é a implementação do objeto remoto. Para isso basta criar uma classe que estenda a classe *skeleton* do servidor e defina os métodos da interface remota (`XImpl.java`):

```
package xMod;

import xMod.X;

class XImpl extends XPOA
{
  // Método a ser executado remotamente
  public String metodoRemoto(int i)
  {
    ...
  }
};
```

Após a implementação do objeto remoto é possível implementar a aplicação servidora que o hospeda. As principais tarefas do servidor são: inicialização da ORB, ativação de um gerenciador POA, criação do objeto remoto, criação de uma referência para o mesmo e associação de um nome à referencia do objeto remoto:

```
package xMod;

import xMod.XPOA;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import xMod.XImpl;
import xMod.X;
import xMod.XHelper;

public class XServer
{
  public static void main(String args[])
  {
    ...

    try {
    // 1 - Criação e inicialização da ORB
    ORB orb = ORB.init(args, null);

    // 2 - Obtenção de uma referência ao POA raiz (root) e
    //   ativação do respectivo gerenciador POA (POA Manager)
    POA rootpoa =
      POAHelper.narrow(orb.resolve_initial_references ("RootPOA");
    Rootpoa.the_POAManager().activate();

    // 3 - Criação do objeto remoto
    XImpl x;
    x = new Ximpl ();

    // 4 - Criação de uma referência para o objeto remoto
```

```
    org.omg.CORBA.Object ref = rootpoa.servant_to_reference(x);
    X xref = XHelper.narrow(ref);

    // 5 - Obtenção de uma referência para o servidor de nomes
    org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
    NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

    // 6 - associação de um nome à referência ao objeto remoto
    String name = "X";
    NameComponent path[] = ncRef.to_name(name);
    ncRef.rebind(path, xref);

    // 7 - servidor entra em um estado de espera por chamadas de
    //   clientes remotos
    orb.run();

    }

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

A aplicação cliente precisa apenas obter uma referência para o objeto remoto de forma a poder executar um método do mesmo:

```
package xMod;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

import xMod.X;
import xMod.XHelper;

public class XClient
{
  public void algumMetodo()
  {
    ...

    try {

    // 1 - Criação e inicialização da ORB
    ORB orb = ORB.init(args, null);

    // 2 - Obtenção de uma referência para o servidor de nomes
    org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
    NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

    // 3 - Obtenção da referência para o objeto remoto utilizando
```

```
    //   o serviço de nomes
    X xRef;
    String name = "X";

    xRef = XHelper.narrow (ncRef.resolve_str(name));

    // 4 - Chamada ao método do objeto remoto
    String s;

    s = x.metodoRemoto(10);

    ...
    }

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

Observação: como uma boa prática de programação deve se fazer o tratamento de exceções específicas e não um tratamento genérico como no exemplo de código do servidor e do cliente acima. Tal tratamento não foi feito pois tornaria o exemplo mais extenso do que o desejado.

Após a compilação dos arquivos fonte (todos arquivos gerados pelo compilador IDL e os arquivos fonte do servidor e do cliente), é possível rodar o exemplo. Em primeiro lugar deve-se inicializar o servidor de nomes, através da execução do programa `orbd`. Em seguida é possível executar o servidor, seguido do cliente.

JacORB

O exemplo anterior implementado em JavaIDL pode ser utilizado sem quaisquer alterações em JacORB, pois as interfaces de programação utilizadas das classes `ORB`, `POA`, `NamingContextExt` e `NameComponent` são padronizadas. Ou seja, o mesmo código pode ser utilizado nestas duas implementações CORBA (JavaIDL e JacORB). Entretanto as ferramentas e o ambiente de execução de JacORB possuem algumas diferenças.

JacORB é uma implementação CORBA totalmente desenvolvida em Java . Sendo assim, seu compilador IDL é uma aplicação Java. Para compilar o arquivo `X.idl` e gerar os arquivos necessários para a implementação das aplicações cliente e servidor, a seguinte linha de comando pode ser utilizada (assumindo que a variável de ambiente `JACORB_BASE` contenha o diretório de instalação de JacORB):

```
# java -classpath \
```

```
"$JACORB_BASE/lib/idl.jar:$JACORB_BASE/lib/jacorb.jar" \
org.jacorb.idl.parser \
-I$JACORB_BASE/idl/omg \
-d . \
X.idl
```

Assim como em JavaIDL, para a execução de aplicações em JacORB é necessário primeiramente inicializar o servidor de nomes. Em JacORB usa-se o utilitário `ns`. Em seguida é possível executar o servidor seguido do cliente. Aplicações JacORB utilizam um arquivo de configuração chamado `jacorb.properties` que entre outras informações contém endereço do servidor de nomes.

2.3 RMI-IIOP

RMI-IIOP – *RMI over IIOP* [26] – é uma tecnologia de programação de objetos distribuídos disponível a partir da versão 1.3 de Java SDK que pode ser vista como uma combinação de RMI e CORBA. RMI-IIOP é resultado de um desenvolvimento conjunto da Sun e da IBM, com o objetivo de disponibilizar uma tecnologia com as melhores características de RMI e CORBA. É importante notar também que RMI-IIOP não seria viável caso modificações nas especificações de CORBA não fossem adotadas pela OMG.

Com RMI-IIOP, aplicações distribuídas são desenvolvidas de forma similar às aplicações desenvolvidas em RMI. Ou seja, as interfaces dos objetos remotos são descritas em Java, sem a necessidade de utilização de uma linguagem específica para a descrição de interfaces, como OMG IDL. E a implementação das interfaces remotas também é realizada em Java, utilizando as interfaces de programação (API – *Application Program Interfaces*) de Java RMI.

Além disso, aplicações desenvolvidas em RMI-IIOP se beneficiam da interoperabilidade entre diferentes linguagens de programação e plataformas de *hardware* de CORBA, graças à possibilidade de utilização do protocolo de comunicação não proprietário (IIOP). Dessa forma clientes RMI-IIOP podem se comunicar com servidores CORBA, assim como servidores RMI-IIOP podem disponibilizar serviços para clientes CORBA.

2.3.1 Arquitetura

A tecnologia de programação de objetos distribuídos RMI-IIOP foi viabilizada

devido à adoção pela OMG de determinadas modificações nas especificações de CORBA. Dentre elas é importante destacar a definição do mapeamento de Java para OMG IDL e para IIOP [27], além de alterações específicas no próprio protocolo CORBA/IIOP. Através desse mapeamento, interfaces de objetos remotos descritas em Java RMI podem ser convertidas para OMG IDL, permitindo a implementação de aplicações CORBA cliente e servidor em diferentes linguagens de programação. Além disso, a especificação [27] também é utilizada para o mapeamento de chamadas Java RMI para o protocolo de comunicação IIOP.

Em RMI-IIOP um servidor pode disponibilizar a interface de um objeto remoto através dos protocolos JRMP, IIOP ou ambos (*dual export* – [28]). A utilização de IIOP por um cliente ou servidor RMI-IIOP implica na utilização de uma ORB para a comunicação entre as partes.

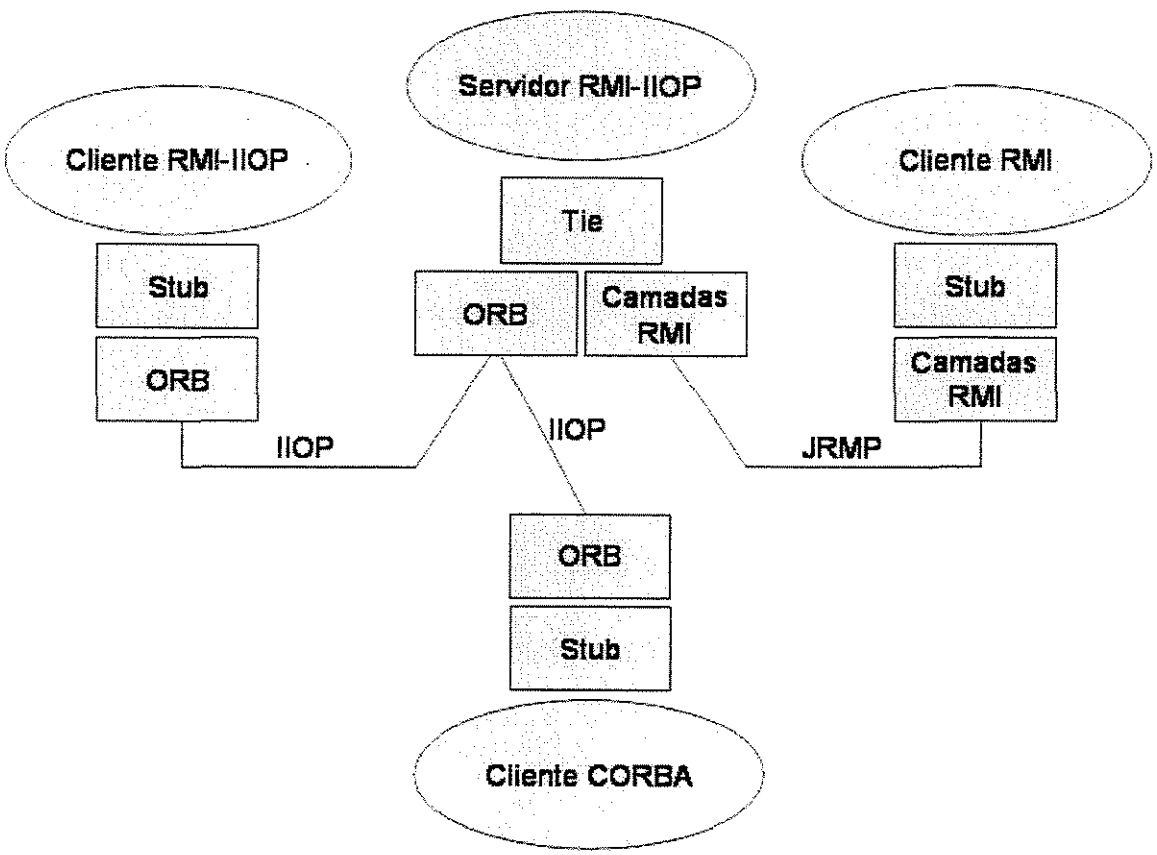

Figura 5 - Arquitetura RMI-IIOP

Assim como RMI, a arquitetura RMI-IIOP é baseada na definição da interface do

serviço remoto. Tanto o objeto servidor quanto o *stub* do cliente implementam a interface do serviço remoto. A diferença em RMI-IIOP é que o compilador de interfaces `rmic` é utilizado para a geração de *stubs* e *ties* IIOP, ao invés de *stubs* e *skeletons* JRMP. A ferramenta `rmic` também pode ser utilizada para a conversão da interface do objeto remoto em Java RMI-IIOP para OMG IDL, de forma a disponibilizar a interface para clientes e servidores CORBA.

Conforme descrito em [28], um servidor implementado em RMI-IIOP é compatível com clientes desenvolvidos em RMI, RMI-IIOP e CORBA. Já um cliente RMI-IIOP é compatível com servidores RMI e RMI-IIOP, mas a compatibilidade não é total com servidores CORBA. Essa incompatibilidade se deve ao fato de que é possível definir construções em OMG IDL que não são mapeáveis para Java RMI-IIOP. Portanto, é possível definir objetos CORBA não accessíveis por clientes RMI-IIOP. A compatibilidade só é garantida quando a interface OMG IDL utilizada pelo servidor CORBA for gerada a partir de uma definição de interface em Java RMI-IIOP.

Assim como Java IDL, RMI-IIOP utiliza *COS Naming Service* [29] como serviço de nomes, serviço implementado pela ORB de Java SDK.

2.3.2 Características

Como características principais de RMI-IIOP é possível destacar:

- **Interoperabilidade com CORBA:** um dos principais objetivos da implementação de RMI sobre IIOP propriamente dita. Entretanto, conforme descrito acima e em [28], essa interoperabilidade não é total. Mesmo assim RMI-IIOP torna possível a interoperabilidade entre aplicações RMI-IIOP e aplicações CORBA implementadas em diferentes linguagens de programação e plataformas de hardware.

- **Interoperabilidade com Java RMI:** além do protocolo IIOP, RMI-IIOP implementa também o protocolo JRMP, utilizado em Java RMI, garantindo a compatibilidade com aplicações Java RMI.

- **Limitações em relação a Java RMI:**
    - Como coleta de lixo distribuída não é implementada em CORBA, a

mesma não pôde ser implementada em RMI-IIOP;

- Não é possível utilizar *casts* da linguagem Java para referências a objetos remotos. O método *narrow* definido em CORBA deve ser utilizado.

- **Seriação de objetos (*Object Serialization*):** RMI-IIOP usa seriação de objetos para a implementação de objetos passados por valor (*objects by value*) de CORBA.

- **Passagem de parâmetros:** RMI-IIOP implementa as mesmas formas de passagem de parâmetros disponíveis em RMI.

- **Carga de código (code downloading):** RMI-IIOP disponibiliza carga dinâmica de código para *stubs*, *ties* e objetos passados por valor, baseada na utilização da propriedade `codebase`. Essa característica é semelhante à carga dinâmica de classes e *stubs* de RMI. Entretanto essa funcionalidade só é possível quando as aplicações cliente e servidor são desenvolvidas em Java.

- **Ativação de objetos remotos:** em RMI-IIOP a ativação de objetos remotos é realizada através da utilização da respectiva funcionalidade disponibilizada pelo *Portable Object Adapter* (POA) de CORBA.

- **Mapeamento de exceções:** em RMI-IIOP, exceções RMI são mapeadas para exceções CORBA e vice-versa [27].

2.3.3 Exemplo

Nesta seção são descritos os passos mínimos necessários para o desenvolvimento de uma aplicação distribuída em RMI-IIOP. Para um tutorial introdutório completo pode-se consultar [31].

O desenvolvimento de uma aplicação RMI-IIOP é similar ao de uma aplicação RMI. Em [30] são descritos os passos para a conversão de aplicações RMI para RMI-IIOP e algumas restrições em relação à utilização das interfaces de programação RMI em aplicações RMI-IIOP.

Em RMI-IIOP o primeiro passo também é a definição da interface do objeto remoto, de forma idêntica à de RMI: a interface deve ser pública e estender a classe

`java.rmi.Remote` e os métodos remotos devem lançar exceções do tipo `java.rmi.RemoteException`:

```
package exemplo_rmiiiop;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface XInt extends java.rmi.Remote
{
  // Método a ser executado remotamente
  public String metodoRemoto(int i) throws java.rmi.RemoteException;
}
```

Em RMI-IIOP, para implementar o objeto remoto, não se deve estender a classe `java.rmi.UnicasRemoteObject`, como se faz em RMI. Uma forma de implementar um objeto remoto que utilize IIOP como protocolo de comunicação é estender a classe `javax.rmi.PortableRemoteObject`. Nesse caso é preciso definir um construtor público sem parâmetros que lance uma exceção `java.rmi.RemoteException`. Finalmente, basta implementar a interface do objeto remoto (arquivo `XImpl.java`):

```
package exemplo_rmiiiop;

import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;

import exemplo_rmiiiop.XInt;

public class XImpl extends javax.rmi.PortableRemoteObject implements XInt
{
  // Construtor
  public XImpl() throws java.rmi.RemoteException
  {
    ...
  }

  // Método a ser executado remotamente
  public String metodoRemoto(int i) throws java.rmi.RemoteException
  {
    ...
  }
}
```

A implementação do servidor que hospeda o objeto remoto RMI-IIOP é similar à mesma efetuada em RMI. Além da criação do objeto remoto é preciso registrá-lo utilizando o serviço de nomes. A diferença em relação a RMI é que se utiliza o serviço de nomes de CORBA ao invés de RMI *Registry*:

```
package exemplo_rmiiiop;
```

```
import javax.naming.InitialContext;
import javax.naming.Context;

import exemplo_rmiiiop.XImpl;

public class XServer
{
  public static void main (String args[])
  {
    ...

    try {
    // 1 - Criação do objeto remoto
    XImpl x;
    x = new XImpl();

    // 2 - Registro do objeto remoto no serviço de nomes
    //   utilizando-se JNDI (Java Naming and Directory Interface)
    Context initialNamingContext = new InitialContext();
    initialNamingContext.rebind("Xserver", x);

    ...
    }

    catch (NamingException e) {
      ...
    }
    ...
  }
}
```

Assim como em RMI, para um cliente RMI-IIOP chamar um método de um objeto remoto basta obter uma referência para o objeto remoto e executar a chamada propriamente dita. Novamente, ao invés de se utilizar RMI *Registry*, utiliza-se o serviço de nomes de CORBA:

```
package exemplo_rmiiiop;

import javax.rmi.*;
import java.rmi.RemoteException;
import javax.naming.NamingException;
import javax.naming.InitialContext;
import javax.naming.Context;

import exemplo_rmiiiop.XInt;

public class XClient
{
  public void algumMetodo()
  {
    ...

    try {
```

```
    // 1 - Obtenção da referência para o objeto remoto do
    //   serviço de nomes utilizando-se JNDI.
    Context ic;
    Object objRef;

    ic = new InitialContext();
    objRef = ic.lookup("Xserver");

    // 2 - Conversão da referência genérica para a referência real
    //   à interface do objeto remoto
    XInt x;
    X = (Xint)PortableRemoteObject.narrow(objRef, XInt.class);

    // 3 - Chamada ao método do objeto remoto
    s = x.metodoRemoto(10);

    ...
    }

    catch (RemoteException e) {
      ...
    }
    ...
  }
}
```

Uma outra diferença em relação a RMI é que em RMI-IIOP, o compilador `rmic` deve receber como entrada o arquivo `.class` relativo à implementação da interface do objeto remoto, ao invés do arquivo `.java`. Sendo assim, antes de se executar o compilador `rmic` é necessário compilar a implementação da interface do objeto remoto.

Em seguida é possível gerar o *stub* do cliente e o *tie* do servidor a partir da classe que implementa a interface remota (`XImpl.class`):

```
# rmic -iiop XImpl
```

O comando acima gera as classes `_XInt_Stub.class` e `_XImpl_Tie.class`.

Após a compilação dos arquivos contendo a interface do objeto remoto, o servidor e o cliente, é possível rodar o exemplo, começando pela ativação do serviço de nomes (através da execução do *daemon* `orbd`) e então executando-se o servidor seguido do cliente.

2.4 JAX-RPC (SOAP)

JAX-RPC – *Java API for XML-based RPC* [32], [40], [41] – é uma tecnologia de

programação de objetos distribuídos cujo mecanismo de RPC é baseado em XML [33] de acordo com as especificações SOAP (*Simple Object Access Protocol* – [34], [35]).

A tecnologia JAX-RPC foi especificada segundo o JCP (*Java Community Process* – [35]) através da iniciativa JSR-101 [36]. JCP é um processo aberto, participativo, criado pela Sun, responsável pelo desenvolvimento e pela revisão das especificações da plataforma Java. JCP foi criado em 1995 e atualmente conta com mais de 500 membros, entre empresas e participantes individuais. JSR (*Java Specification Request*) é um documento submetido por um ou mais membros ao grupo da Sun responsável pelo JCP com o objetivo de propor o desenvolvimento de uma nova especificação ou uma grande revisão numa especificação já existente. O desenvolvimento de uma especificação propriamente dito é realizado por um grupo formado por especialistas dentre os membros do JCP.

XML (*eXtensible Markup Language*) é uma linguagem texto padrão especificada pelo W3C [38] para a representação de dados de uma maneira independente de plataforma. Em XML os dados são dispostos entre etiquetas (*tags*) em um documento XML, de uma maneira similar a HTML. XML foi especificado com o objetivo de possibilitar a descrição de informações. XML pode ser utilizado para a troca de dados e seu compartilhamento entre aplicações, para o armazenamento de dados e para criar novas linguagens, entre outras aplicações. XML permite a definição da estrutura de um documento através da utilização de um esquema (*schema*). Um esquema é um documento texto que define etiquetas e uma estrutura hierárquica para a utilização de etiquetas em um documento XML.

As principais diferenças entre XML e HTML são:

- as etiquetas usadas em XML se referem ao significado do conteúdo dos dados que elas englobam, enquanto que em HTML as etiquetas apenas determinam como uma porção de texto deve ser apresentada graficamente;
- XML é estendível, permitindo a definição de novas etiquetas, enquanto que as etiquetas utilizadas em HTML são apenas as que fazem parte da sua especificação;
- com XML é possível definir a estrutura de um documento e a hierarquia para a utilização de etiquetas.

A utilização de esquemas torna XML uma linguagem portável. Como documentos

XML e esquemas são documentos texto, é possível interpretá-los em qualquer plataforma.

SOAP é um protocolo especificado pela W3C para a chamada de procedimentos remotos. Suas mensagens são definidas utilizando-se XML. Como protocolo de transporte, a especificação SOAP define a utilização de HTTP. Entretanto outros protocolos tais como FTP e SMTP poder sem utilizados, pois o principal requisito é a capacidade de transportar texto (definições XML). SOAP pode ser comparado aos protocolos JRMP de Java RMI e IIOP de CORBA.

SOAP define um formato padrão em XML para a representação de chamadas de procedimentos remotos e das respectivas respostas através das denominadas "mensagens SOAP". A versão atual de JAX-RPC (1.1) determina a utilização de SOAP 1.1 sobre o protocolo de transporte HTTP 1.1.

Como as demais tecnologias apresentadas anteriormente, JAX-RPC é centrada na definição da interface do serviço remoto. A linguagem utilizada para isso é WSDL (*Web Services Description Language* – [37]). WSDL utiliza XML para a descrição da interface. Em WSDL também é necessário definir o formato das mensagens e o protocolo de transporte (por exemplo, SOAP sobre HTTP). Uma diferença em relação às demais linguagens é que em WSDL também é especificado o endereço pelo qual o serviço remoto pode ser acessado.

A partir da definição do serviço em WSDL é possível gerar as classes Java que podem ser utilizadas na implementação de servidores (*ties*) e clientes (*stubs*). JAX-RPC define o mapeamento das definições em WSDL e XML para Java e vice-versa. Entretanto não há nenhuma restrição quanto à implementação de servidores e clientes utilizando outras linguagens de programação, dado que toda a comunicação é realizada através do intercâmbio de documentos XML, uma linguagem texto padrão. Pelo mesmo motivo, ou seja, pela utilização de especificações padrão tais como XML, SOAP e HTTP, é possível obter interoperabilidade entre diferentes plataformas de *hardware*.

Assim como CORBA, JAX-RPC também provê uma interface dinâmica para a invocação de procedimentos remotos (DII – *Dynamic Invocation Interface*), dispensando a utilização de *stubs* do lado do cliente. JAX-RPC define ainda uma outra forma para invocação sem a necessidade de *stubs* chamada *Dynamic Proxy*. A forma de invocação baseada na utilização de *stubs* é denominada *Static Stub*.

Para a chamada de um método remoto, a aplicação cliente não obtém uma referência para um objeto específico do servidor, mas sim uma referência para um objeto que representa o serviço oferecido como um todo. De posse dessa referência, a aplicação cliente efetua a chamada ao método remoto como se fosse uma chamada a um método local.

Esse modelo de utilização de uma referência genérica para o serviço atende principalmente a aplicações chamadas de *Web Services* ("Serviços da Web") – mas nada impede que outros tipos de aplicação o utilizem. *Web Services* são aplicações que rodam em servidores da Web e aceitam requisições de clientes distribuídos pela Internet. A utilização de SOAP sobre HTTP como protocolo e de WSDL como linguagem para descrição da interface do serviço remoto se tornou o padrão de mercado para *Web Services*. Ou seja, JAX-RPC se enquadra perfeitamente no modelo de *Web Services*.

A especificação JAX-RPC não define nenhum mecanismo para o registro de um serviço pelo servidor e para a descoberta de serviços por clientes. Para isso é necessário utilizar algum outro mecanismo, tal como JAXR (*Java API for XML Registries* – [39]).

2.4.1 Arquitetura

Um dos objetivos da especificação JAX-RPC é definir um conjunto de interfaces de programação (API) Java que simplifique a utilização de um mecanismo de RPC baseado em XML (SOAP). Com JAX-RPC o programador não deve precisar conhecer os detalhes envolvidos na utilização de WSDL, XML e SOAP.

JAX-RPC também define um ambiente de execução (*runtime system*) como componente central de uma implementação JAX-RPC. De acordo com a especificação de JAX-RPC, o ambiente de execução do lado do servidor pode ser implementado através de um ambiente baseado em: J2SE (*Java 2 Standard Edition* – [42]), contêiners de *Servlets* (classes Java utilizadas para o tratamento de requisições e respostas HTTP – [43]) ou contêiners J2EE (*Java 2 Enterprise Edition* – [44]).

A arquitetura de uma implementação JAX-RPC pode ser representada pelo seguinte diagrama:

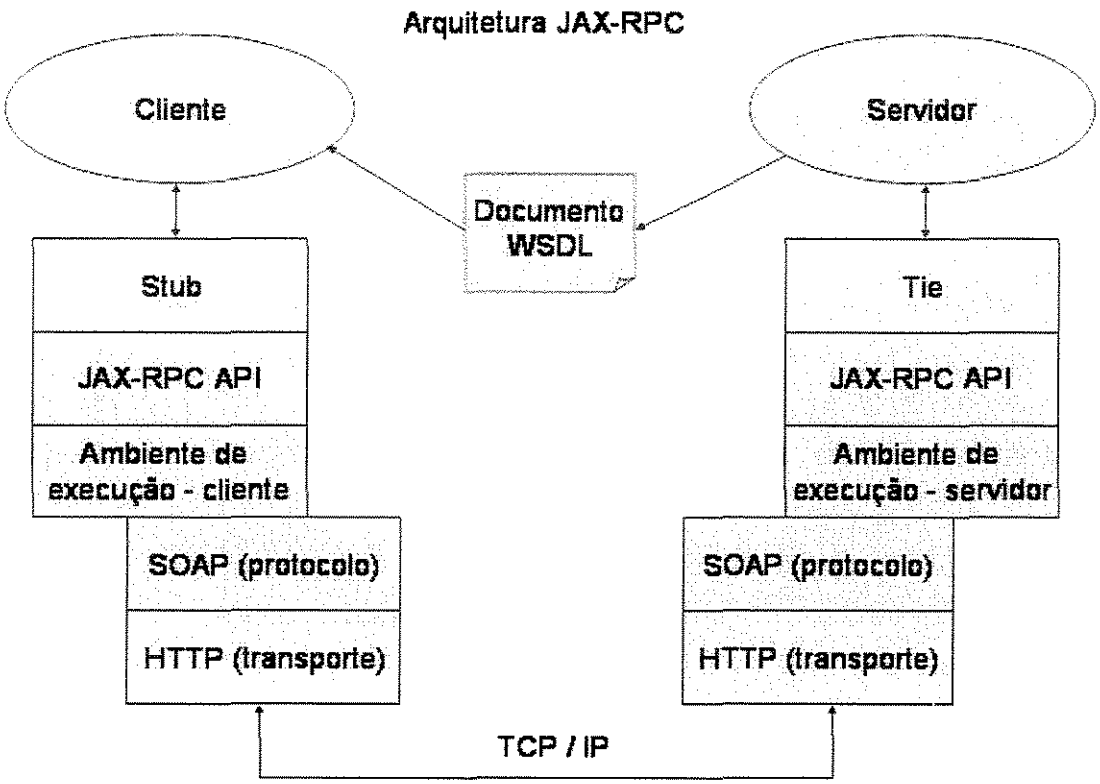

Figura 6 - Arquitetura JAX-RPC

O diagrama acima contém os módulos envolvidos em uma chamada de procedimento remoto onde o cliente utiliza *stubs* (*Static Stub invocation*). Nas demais formas de invocação, a comunicação entre a aplicação cliente e o ambiente de execução é direta.

O *stub* do cliente é responsável por converter uma chamada de procedimento remoto efetuada pela aplicação em chamadas a serviços do ambiente de execução. Além disso, o *stub* aguarda pela resposta da chamada e a repassa para a aplicação.

O ambiente de execução do cliente faz o mapeamento das chamadas a seus serviços para uma mensagem SOAP e a envia para o servidor através de uma requisição HTTP. Ao receber a resposta HTTP da chamada, o ambiente de execução extrai a mensagem SOAP e a repassa para o *stub* do cliente.

Do lado do servidor, o ambiente de execução recebe requisições HTTP e extrai as

respectivas mensagens SOAP. Essas mensagens são mapeadas para chamadas a métodos da classe *tie* do servidor. Ao receber o código de retorno da execução de um método, o ambiente de execução o converte em uma mensagem SOAP e a transmite para o cliente como uma resposta HTTP.

O *tie* do servidor é responsável pela chamada do procedimento remoto propriamente dito e pela conversão da resposta em chamadas ao ambiente de execução do servidor.

Implementações JAX-RPC devem incluir uma ferramenta para o mapeamento de definições WSDL para Java e vice-versa. Essa ferramenta é utilizada para a geração de todos os artefatos necessários para a comunicação do cliente e do servidor com os seus respectivos ambientes de execução. Também deve ser possível partir de uma definição de um serviço em Java (similar à definição de uma interface Java RMI) e gerar a respectiva definição em WSDL.

2.4.2 Características

A seguir são apresentadas algumas das principais características de JAX-RPC:

- **Independência de linguagem de programação e plataforma de hardware [46]:** embora aplicações JAX-RPC sejam implementadas em Java (nas plataformas onde uma máquina virtual está disponível), a utilização de tecnologias padronizadas (XML, SOAP, HTTP e WSDL) permite que clientes e servidores também sejam implementados em outras linguagens e plataformas.

- **Associação de Protocolos (*Protocol Bindings*):** JAX-RPC obriga que suas implementações permitam a utilização do protocolo SOAP, mas também permite que outros protocolos baseados em XML sejam utilizados.

- **Protocolo de Transporte:** JAX-RPC exige a implementação do transporte de mensagens SOAP sobre HTTP, mas também permite que outros protocolos de transporte sejam utilizados.

- **Modos de programação de aplicações cliente:**
    - ***Static Stub*:** chamadas de procedimentos remotos através da utilização de stubs, classes cuja implementação não é padronizada;

- ***Dynamic Proxy***: chamadas efetuadas através da utilização de uma classe criada durante a execução, ou seja, uma aplicação cliente utilizando esse método é independente da implementação de JAX-RPC;
- ***Dynamic Invocation Interface***: chamadas de procedimentos remotos cujo nome e assinatura não são conhecidos em tempo de compilação.

- **Modos de invocação de serviços remotos (no nível de aplicação):**
  - **síncrono:** a aplicação cliente fica bloqueada esperando a resposta da invocação do método remoto (ou por uma exceção caso ocorra um erro);
  - **sem resposta (*one-way* RPC):** nesse modo não existe resposta. A aplicação cliente invoca o método remoto e continua sua execução. Não existe garantia que a requisição será atendida. O ambiente de execução pode lançar uma exceção se detectar a ocorrência de um erro. Na versão atual de JAX-RPC, a única forma de utilização desse modo de invocação é através da interface de invocação dinâmica;
  - **não bloqueante (*non-blocking* RPC):** a aplicação cliente faz a chamada e não é bloqueada. Para obter a resposta a aplicação pode fazer uma chamada específica para verificar se a resposta já está disponível (e então recebê-la) ou bloquear a sua execução enquanto a resposta não é recebida.

- **Modos de passagem de parâmetros:** JAX-RPC utiliza apenas passagem de parâmetros por valor (cópia). O análogo também é válido para códigos devolvidos nas chamadas de métodos remotos. Além disso, uma aplicação cliente tem acesso apenas ao estado (valores dos atributos) de objetos recebidos por valor. O comportamento (implementações de métodos) dos objetos recebidos não é recebido.

- **Mapeamento de tipos definidos em Java para XML/WSDL:** a especificação de JAX-RPC contempla o mapeamento de apenas um subconjunto dos tipos que podem ser definidos em Java: a maioria dos tipos primitivos (com exceção de `char`) e as respectivas classes *wrapper*, um subconjunto das classes padrão de Java, vetores de tipos contemplados pela especificação, exceções e *value types* (objetos passados por valor). Existem requisitos específicos para *value types*. Maiores detalhes podem ser encontrados em [32].

- **Segurança:** o único mecanismo de segurança que JAX-RPC exige que seja

implementado é autenticação. Entretanto JAX-RPC permite que sejam utilizados protocolos de transporte seguros, tais como HTTPS.

- **A especificação JAX-RPC não provê mecanismos para a implementação de:**
  - *stubs* e *skeletons/ties* portáveis entre diferentes implementações;
  - Serviço de Registro (ou nomes/diretório);
  - objetos passados por referência.

2.4.3 Exemplo

A seguir são apresentados os passos mínimos necessários para a implementação de um cliente e um servidor utilizando a implementação de referência da Sun para JAX-RPC, parte integrante do pacote de desenvolvimento da Sun para aplicações XML, *Web Services* e aplicações para a Web (Java WSDP – *Java Web Services Development Pack* – [47]).

O capítulo referente a JAX-RPC do tutorial que acompanha o Java WSDP [48] contém maiores detalhes. O exemplo toma como base a utilização de Java WSDP no ambiente de desenvolvimento e execução do J2SE. Java WSDP inclui o contêiner de servlets Tomcat, da *Apache Software Foundation*, utilizado para hospedar aplicações servidoras.

É possível definir a interface de um serviço remoto em Java e depois mapeá-la para WSDL. Essa definição é muito parecida com uma definição de um serviço em RMI. Em JAX-RPC também é necessário estender a classe `java.rmi.Remote` e que os métodos lancem exceções do tipo `java.rmi.RemoteException`. Uma diferença é que não pode haver a declaração de nenhuma constante (`public final static`). Além disso, parâmetros e códigos de retorno de métodos devem seguir as restrições da especificação de JAX-RPC.

```
package xserver;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface XInt extends java.rmi.Remote
{
  // Método a ser executado remotamente
  public String metodoRemoto(int i) throws java.rmi.RemoteException;
}
```

Para a implementação do objeto remoto basta implementar a respectiva interface. Não há necessidade de estender nenhuma classe (arquivo `XImpl.java`):

```
package xserver;

import java.rmi.RemoteException;

import xserver.XInt;

public class XImpl implements XInt
{
  // Método a ser executado remotamente
  public String metodoRemoto(int i) throws java.rmi.RemoteException
  {
    ...
  }
}
```

Após a compilação das definições acima é necessário executar a ferramenta `wscompile` com a opção `-define`, de forma a gerar o documento WSDL correspondente à definição da interface do serviço remoto e um modelo XML necessário para a implantação do servidor:

```
# wscompile.sh -define -d <output directory> -nd <output directory> \
               -classpath <directory containing compiled classes> \
               config-interface.xml \
               -model <output directory>/model.gz
```

O arquivo `config-interface.xml` é um arquivo de entrada e deve conter definições a respeito do serviço sendo implementado: nome do serviço, nome do pacote contendo as classes que implementam o serviço e nome da classe que representa a interface do serviço:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service
      name="XService"
      targetNamespace="urn:Foo"
      typeNamespace="urn:Foo"
      packageName="xserver">
      <interface name="xserver.XInt"/>
  </service>
</configuration>
```

O documento WSDL gerado pelo comando `wscompile` acima é o seguinte (arquivo `XService.wsdl`):

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

```
<definitions name="XService" targetNamespace="urn:Foo"
             xmlns:tns="urn:Foo" xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <types/>
  <message name="XInt_metodoRemoto">
    <part name="int_1" type="xsd:int"/>
  </message>
  <message name="XInt_metodoRemotoResponse">
    <part name="result" type="xsd:string"/>
  </message>
  <portType name="XInt">
    <operation name="metodoRemoto" parameterOrder="int_1">
      <input message="tns:XInt_metodoRemoto"/>
      <output
         message="tns:XInt_metodoRemotoResponse"/>
    </operation>
  </portType>
  <binding name="XIntBinding" type="tns:XInt">
    <operation name="metodoRemoto">
      <input>
        <soap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                use="encoded" namespace="urn:Foo"/>
      </input>
      <output>
        <soap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                use="encoded" namespace="urn:Foo"/>
      </output>
      <soap:operation soapAction=""/></operation>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                  style="rpc"/>
  </binding>
  <service name="XService">
    <port name="XIntPort" binding="tns:XIntBinding">
      <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
  </service>
</definitions>
```

Diferentemente das outras tecnologias, em JAX-RPC não é necessário desenvolver uma aplicação que crie uma instância do objeto servidor e atenda a requisições de aplicações clientes. Essa funcionalidade é provida pela tecnologia utilizada para a implementação do ambiente de execução de JAX-RPC (J2SE – Servlets, J2EE, etc).

Em contrapartida é necessário criar um pacote para a instalação do serviço. Primeiramente é necessário utilizar a ferramenta `jar` do SDK para criar um pacote (arquivo `.war`) contendo as classes do servidor, o respectivo modelo XML (`model.gz`) e mais dois arquivos auxiliares (`web.xml` e `x-ri.xml`).

Arquivo web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
  <display-name>X Test Application</display-name>
  <description>A web application with a JAX-RPC endpoint</description>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>
```

Arquivo x-ri.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<webServices
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd" version="1.0"
    targetNamespaceBase="urn:Foo" typeNamespaceBase="urn:Foo"
    urlPatternBase="/ws">
    <endpoint
        name="MyXTest"
        displayName="X Test Service"
        description="A simple web service"
        interface="xserver.XInt"
        model="/WEB-INF/model.gz"
        implementation="xserver.XImpl"/>
    <endpointMapping
        endpointName="MyXTest"
        urlPattern="/XTest"/>
</webServices>
```

O parâmetro de configuração `urlPattern` contido no arquivo `x-ri.xml` é o nome utilizado para acessar o servidor.

Para a criação do arquivo `x-portable.war`, basta colocar todos arquivos mencionados acima em um diretório chamado `WEB-INF` e executar:

```
# jar -cvf x-portable.war WEB-INF
```

O arquivo `x-portable.war` gerado acima é um arquivo que contém definições portáveis entre diferentes implementações de JAX-RPC. Para completar a criação do pacote de instalação do serviço requerido pelo ambiente de Java WSDP, basta executar a ferramenta `wsdeploy` da seguinte forma:

```
# wsdeploy.sh -o x.war x-portable.war
```

A ferramenta `wsdeploy` gera as classes *tie* do servidor e o arquivo WSDL que

representa o serviço e os empacota no arquivo `x.tar` juntamente com os arquivos contidos em `x-portable.tar`.

O último passo é implantar o serviço no Tomcat. A forma mais simples é utilizar a interface gráfica do seu gerenciador de aplicações, selecionando a opção "*Deploy*", passando o arquivo `x.war` como entrada. Em instalações padrão de Java WSDP esse gerenciador pode ser acessado através da URL `http://localhost:8080/manager/html/`. Nesse caso, para verificar a instalação da aplicação basta acessar `http://localhost:8080/x/XTest` (`/x` deriva do nome do arquivo `x.war` e `/XTest` é o parâmetro `urlPattern` do arquivo de configuração `x-ri.xml`).

A forma mais simples de implementação de um cliente JAX-RPC é através da utilização de *stubs* geradas em tempo de compilação (*static stub*). Nesse método antes de um cliente efetuar uma chamada a um método remoto é necessária a criação de um objeto do tipo `javax.xml.rpc.Stub` e configurar o endereço do serviço remoto:

```
import javax.xml.rpc.Stub;

public class XClient
{
  public static void main(String args[])
  {
    ...

    try {

    // 1 - Criação do objeto Stub
    Stub stub;

    stub = (Stub)(new XService_Impl().getXIntPort());

    // 2 - Configuração do endereço do service remoto
    stub._setProperty(java.xml.rpc.Stub.ENDOPOINT_ADDRESS_PROPERTY,
                      args[0]);

    // 3- - Conversão do tipo Stub para o tipo da interface do serviço
    XInt x;

    x = (Xint)stub;

    // 4 - Chamada ao método do objeto remoto
    String str;

    str = x.metodoRemoto(10);

    ...
    }
```

```
    catch (RemoteException e) {
      ...
    }
    ...
  }
}
```

Em seguida é possível gerar a classe *stub* do cliente e outras classes auxiliares requeridas para a execução do mesmo utilizando a ferramenta `wscompile`:

```
# wscompile.sh -gen:client \
               -d <output directory> \
               -classpath <directory containing compiled classes> \
                config-wsdl.xml
```

O arquivo de configuração `config-wsdl.xml` contém as informações necessárias para a ferramenta `wscompile` acessar o arquivo WSDL contendo as definições do serviço remoto:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
      xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location="http://localhost:8080/x/XTest?WSDL"
       packageName="xclient"/>
</configuration>
```

Após a compilação do cliente, é possível executá-lo através do comando:

```
# java -classpath <directory containing compiled classes> \
       xclient.XClient <endpoint-address>
```

2.4.4 Comparação de Características

Por estarem disponíveis para a linguagem de programação Java, as tecnologias descritas acima podem ser utilizadas em diversas plataformas de *hardware*.

A seguinte tabela contém uma lista com as principais características encontradas nas tecnologias apresentadas e uma comparação entre a forma que as diferentes tecnologias implementam (ou não) as mesmas:

	RMI	RMI-IIOP	CORBA (JavaIDL/JacROB)	JAX-RPC
Independência de linguagem de programação	Disponível apenas em Java	Disponível apenas em Java	Disponível em diversas linguagens (Java, C++,	Disponível em algumas linguagens

			Smalltalk, etc	
Linguagem para definição de interfaces	Java	Java	OMG IDL	WSDL ou Java
Passagem de objetos como parâmetros	Por referência (objetos remotos) ou por valor	Por referência (objetos remotos) ou por valor	Por referência (objetos remotos) ou por valor (`valuetype`)	Apenas por valor e restrito ao estado do objeto (valores dos atributos)
Implementação de polimorfismo distribuído	Sim	Não	Não	Não
Carga dinâmica de classes (definição e métodos/código)	Sim	Sim, mas apenas para cliente e servidor em Java	Não	Não
Coleta de lixo distribuída	Sim	Não	Não	Não
Ativação de objetos remotos	Sim	Sim	Sim	Sim
Invocação dinâmica de métodos remotos	Não	Não	Sim	Sim
Parâmetros de saída	Não	Não	Sim	Não
Invocação assíncrona de métodos remotos	Não	Não	Sim	Sim
Tipo de dado abstrato	Não	Não	Sim (OMG IDL `any`)	Não

Analisando-se a tabela acima, é possível observar que RMI e CORBA possuem características interessantes que as diferenciam das demais. JAX-RPC (SOAP) não apresenta nenhuma característica não encontrada nas outras. RMI-IIOP tem a vantagem de possibilitar a interoperabilidade entre aplicações RMI e CORBA.

Apesar de ser o padrão atual do mercado para *Web Services*, SOAP tem várias limitações [52]:

- a especificação de SOAP foi baseada na utilização de HTTP, cujo modelo de requisição e resposta não é apropriado para todas as situações envolvidas em chamadas de procedimentos remotos;
- as mensagens trocadas entre clientes e servidores na forma de documentos XML torna necessário o uso de analisadores sintáticos e semânticos (*parsers*) em tempo de execução;
- SOAP não provê mecanismos para a passagem de objetos por referência;
- SOAP não define um mecanismo para a descoberta de serviços disponíveis (*yellow pages*). Para obter essa funcionalidade, outras especificações precisam ser atualizadas, tais como UDDI [54] e ebXML *Registry and Repository* [55].

SOAP não é um protocolo completamente orientado a objetos, assim como sua arquitetura. "SOAP foi uma arma eficaz para acabar com a possibilidade de termos um protocolo decente orientado a objetos utilizado largamente em aplicações para Internet no mundo comercial" [53].

Capítulo 3

Metodologia

Chamadas de procedimentos remotos são mais complexas do que chamadas de procedimentos locais, dado que as primeiras envolvem comunicação entre processos distintos (tipicamente rodando em máquinas diferentes) e o empacotamento (*marshalling*) / desempacotamento das chamadas e dos valores devolvidos pelas mesmas em um formato independente de plataforma e adequado para essa comunicação.

Neste trabalho é apresentada uma comparação de desempenho das tecnologias de programação distribuída apresentadas no capítulo anterior. Este capítulo descreve a metodologia adotada nas comparações e contém detalhes sobre os programas de teste desenvolvidos. Em seguida são analisados trabalhos contendo comparações similares entre tecnologias de programação distribuída ([49] e [50]). O próximo capítulo contém a descrição dos resultados obtidos nos testes comparativos.

3.1 Ambiente de Testes

3.1.1 Cenários

Para a comparação do desempenho das diferentes tecnologias, os programas de teste desenvolvidos foram executados em três cenários diferentes:

- **Cenário A**: cliente e servidor rodando na mesma máquina (`narizinho.lsd.unicamp.br`);
- **Cenário B**: cliente e servidor rodando em máquinas distintas de uma mesma sub-rede (respectivamente `cupuacu.lsd.unicamp.br` e `narizinho.lsd.unicamp.br`);
- **Cenário C**: cliente e servidor rodando em máquinas distintas de redes diferentes (respectivamente `narizinho.lsd.unicamp.br` e `20844209.cps.virtua.com.br`).

A comparação entre o desempenho nos cenários A e B possibilita medir o custo adicional (*overhead*) da rede no desempenho da comunicação entre cliente e servidor.

Como as máquinas `narizinho` e `cupuaçu` pertencem a uma pequena sub-rede isolada (LSD – Laboratório de Sistemas Distribuídos do IC-UNICAMP) e estão conectadas seqüencialmente em uma rede *Ethernet*, sua comunicação é direta, sofrendo de pouca interferência de tráfego de comunicação entre outras máquinas. A execução do comando `traceroute` mostra que a comunicação entre essas máquinas pode ser efetuada diretamente:

- resultado do comando `traceroute cupuacu` executado a partir de `narizinho`:

```
# traceroute cupuacu

traceroute to cupuacu.lsd.ic.unicamp.br (143.106.24.145), 30 hops
max, 38 byte packets
1  cupuacu (143.106.24.145)  0.318 ms  0.252 ms  0.242 ms
```

- resultado do comando `traceroute narizinho` executado a partir de `cupuacu`:

```
# traceroute narizinho

traceroute to narizinho.lsd.ic.unicamp.br (143.106.24.134), 30 hops
max, 38 byte packets
1  narizinho (143.106.24.134)  0.374 ms  0.273 ms  0.269 ms
```

O cenário C é mais genérico, com as aplicações rodando em máquinas de redes distintas separadas geograficamente, se comunicando através da Internet. Este cenário foi utilizado para se estudar o comportamento das tecnologias em um ambiente heterogêneo e sujeito a muita interferência de tráfego.

Para o cenário C, o resultado do comando `traceroute` a partir da máquina `narizinho` é o seguinte (foi omitido o resultado do comando `traceroute` para o caminho inverso, pois é similar em termos de número de nós envolvidos):

```
# traceroute 20844209.cps.virtua.com.br

traceroute to 200.208.44.209 (200.208.44.209), 30 hops max, 38 byte
packets
 1  143.106.24.190 (143.106.24.190)  2.396 ms  2.632 ms  2.129 ms
 2  capivari.dcc.unicamp.br (143.106.7.15)  0.603 ms  0.616 ms
0.596 ms
 3  ansp-gw.unicamp.br (143.106.2.45)  1.091 ms  1.448 ms  1.112 ms
 4  unicamp-fe02-core.cas.ansp.br (143.106.99.25)  1.768 ms  1.971
ms  1.463 ms
 5  advansp-border.core.unicamp.br (143.106.99.73)  1.952 ms  2.838
ms  2.254 ms
```

```
 6  spo-cps.ansp.br (143.108.254.129)  3.932 ms  4.952 ms  3.103 ms
 7  200.17.61.1 (200.17.61.1)  3.154 ms  3.686 ms  5.292 ms
 8  embratel-A4-0-58-acc12.spo.embratel.net.br (200.211.38.149)
4.310 ms  4.280 ms  5.451 ms
 9  ebt-G11-0-core01.spo.embratel.net.br (200.230.219.209)  7.922
ms  8.508 ms  9.083 ms
     MPLS Label=179 CoS=3 TTL=1 S=0
10  ebt-A1-0-1-dist02.cas.embratel.net.br (200.230.1.53)  6.461 ms
6.159 ms  6.307 ms
11  200.178.69.6 (200.178.69.6)  8.251 ms  9.732 ms  8.985 ms
12  200.208.44.1 (200.208.44.1)  11.037 ms  8.348 ms  10.800 ms
13  200.208.44.209 (200.208.44.209)  41.057 ms  39.355 ms  40.075
ms
```

3.1.2 Equipamentos utilizados

As máquinas utilizadas nos cenários A e B são computadores Pentium III de 800 MHz. A máquina `narizinho` possui 384 Mbytes de RAM enquanto que a máquina `cupuacu` possui 256 Mbytes de RAM. É importante notar que a máquina `narizinho` foi utilizada como servidora nesses dois cenários. A utilização de máquinas de capacidade de processamento similar nos cenários A e B auxilia a legitimar os resultados obtidos.

A máquina `20844209.cps.virtua.com.br` utilizada como servidora no cenário C é um Athlon XP 2600 (equivalente a um Pentium IV de 2.0 GHz) com 512 Mbytes de RAM.

3.1.3 Versões de Software Utilizadas

Todos os computadores foram utilizados com o sistema operacional Linux. As máquinas `narizinho` e `cupuacu` utilizam a distribuição Fedora (versão do núcleo 2.4.22-1) enquanto que a máquina `20844209.cps.virtua.com.br` utiliza Red Hat 9.0 (versão do núcleo 2.4.20-8).

As aplicações foram desenvolvidas, compiladas e executadas na versão 1.4.2_01 de J2SE (*Java 2 Standard Edition*). A versão de JacORB utilizada foi a 1.4.1, enquanto que a versão de JWSDP (*Java Web Services Development Pack*) foi a 1.3.

3.2 Testes Efetuados

Os testes efetuados consistem de comparações entre chamadas de procedimentos remotos utilizando as diferentes tecnologias estudadas: RMI, RMI-IIOP, CORBA (JavaIDL e JacORB) e JAX-RPC.

A fim de se determinar o custo adicional da chamada de métodos remotos em comparação com a chamada de métodos locais, os métodos remotos não efetuam quase nenhum processamento. Alguns métodos devolvem imediatamente o valor de uma variável e outros apenas executam um comando `switch` antes de devolver um valor. Ou seja, as medidas tomadas correspondem a todas as operações envolvidas em uma chamada de procedimento remoto, com exceção da execução do método propriamente dito.

Para a medida do tempo gasto na execução dos procedimentos remotos, foi utilizado o método `System.currentTimeMillis()` do pacote `java.lang.System` da seguinte forma:

```
...
  tempoInicial = System.currentTimeMillis();
  r = objetoRemoto.getX();
  tempoFinal = System.currentTimeMillis();
  tempoGasto = tempoInicial - tempoFinal;
...
```

Foi comparado o envio de cadeias de caracteres (*strings*) de diversos tamanhos: 1, 10, 20, 50, 100, 200, 500, 1.000, 2.000, 5.000, 10.000, 20.000, 50.000 e 100.000 caracteres. Também foi comparado o envio de listas de tamanho variável contendo objetos de um tipo definido especificamente para este experimento: 1, 10, 20, 50, 100, 200, 500, 700, 850 e 1.000 objetos.

Durante a execução dos testes foi observada a utilização de UCP (Unidade Central de Processamento) e memória através do utilitário `top` do Linux.

3.2.1 Interfaces dos Serviços Remotos

Os programas de teste desenvolvidos nas diferentes tecnologias implementam interfaces de serviços remotos equivalentes.

Para RMI, RMI-IIOP e JAX-RPC, a definição da interface do serviço remoto é feita em Java e é idêntica. Apenas para distinção entre as implementações, foi definido um pacote para cada tecnologia e um nome distinto para cada interface. Na definição a seguir `<nome_do_pacote>` deve ser substituído por `rmi`, `rmiiiop` ou `jaxrpcserver` e `<prefixo>` deve ser substituído por `RMI`, `RMIIIOP` ou `JAXRPC`:

```
// Package declaration
package <nome_do_pacote>;

// Imports
import java.rmi.Remote;
import java.rmi.RemoteException;

import util.TestObject;
import util.TestObjectList;

// Class definition
public interface <prefixo>TestInterface extends Remote
{
    TestObject getTestObject() throws RemoteException;

    String getString1() throws RemoteException;
    String getString10() throws RemoteException;
    String getString20() throws RemoteException;
    String getString50() throws RemoteException;
    String getString100() throws RemoteException;
    String getString200() throws RemoteException;
    String getString500() throws RemoteException;
    String getString1000() throws RemoteException;
    String getString2000() throws RemoteException;
    String getString5000() throws RemoteException;
    String getString10000() throws RemoteException;
    String getString20000() throws RemoteException;
    String getString50000() throws RemoteException;
    String getString100000() throws RemoteException;

    TestObjectList getTestObjectList(int size) throws RemoteException;
}
```

O método `getTestObject()` foi definido apenas para a validação da devolução de um objeto por valor e seu desempenho não foi comparado. Os métodos utilizados nas medições são os `getString<n>()` e `getTestObjectList()`.

O objeto `TestObjectList` é uma lista ligada de objetos do tipo `TestObject`. `TestObjectList` possui um construtor que recebe como parâmetro o tamanho da lista que deve ser criada. O seguinte trecho de código contém as definições principais da classe `TestObjectList`:

```
public class TestObjectList implements Serializable
{
    private int size;        /* size of the list */
    private TestObject head; /* list head        */

    public TestObjectList(int s)
    {
      ...
    }
```

```
    ...
}
```

É importante notar que a classe `TestObjectList` implementa a interface `Serializable`, necessário para a passagem e devolução de objetos por valor em RMI e RMI-IIOP.

`TestObject` é uma classe cujos atributos são compostos por tipos primitivos da linguagem Java (com exceção de `char`, não implementado por JAX-RPC) e por uma referência para um objeto do mesmo tipo. O seguinte trecho de código contém a definição dos atributos da classe `TestObject`:

```
public class TestObject implements Serializable
{
    private boolean bool; /* true/false - 1 bit*/
    private byte b;       /* -128 to 127 (8 bits) */
    private short s;      /* -32.768 to 32.767 (16 bits) */
    private int i;        /* -2.147.483.648 to 2.147.483.647 (32 bits) */
    private long l;       /* -9.223.372.036.854.775.808 to
                             9.223.372.036.854.775.807 (64 bits) */
    private float f;      /* floating point - 32 bits */
    private double d;     /* floating point - 64 bits */
    private TestObject objRef; /* objectReference - 32 bits */

    ...
}
```

Um requisito para que a mesma interface do serviço remoto de RMI/RMI-IIOP possa ser utilizada por aplicações JAX-RPC é que sejam definidos métodos *get* e *set* para todos os atributos privados (`private`) cujo valor deve ser disponibilizado para o usuário do serviço remoto nas passagens e devoluções de objetos por valor.

Para CORBA (JavaIDL e JacORB) a definição da interface do serviço remoto deve ser feita em OMG IDL. Como nas tecnologias descritas anteriormente, para distinção entre as implementações foi definido um pacote para cada tecnologia, módulos e interfaces com nomes distintos. Na definição a seguir `<tecnologia>` deve ser substituído por `javaidl` ou `jacorb` e `<prefixo>` deve ser substituído por `JavaIDL` ou `JacORB`:

```
module <tecnologia>
{
  module <tecnologia>util
  {
    valuetype <prefixo>TestObject
    {
      private boolean bool;  // for Java boolean
```

```
      private octet b;        // for Java byte
      private short s;        // for Java short
      private long i;         // for Java int
      private long long l;    // for Java long
      private float f;        // for Java float
      private double d;       // for Java double
      private ::<tecnologia>::<tecnologia>util::<prefixo>TestObject o;
                                          // for Java object reference

      void setObjectReference (
             in ::<tecnologia>::<tecnologia>util::<prefixo>TestObject r);

      ::<tecnologia>::<tecnologia>util::<prefixo>TestObject
           getObjectReference();

      void print();
    };

    valuetype <prefixo>TestObjectList
    {
      private long size;
      private ::<tecnologia>::<tecnologia>util::<prefixo>TestObject head;

      void print();
    };
  };

  module <tecnologia>test
  {
    interface <prefixo>TestInterface
    {
      wstring getString1();
      wstring getString10();
      wstring getString20();
      wstring getString50();
      wstring getString100();
      wstring getString200();
      wstring getString500();
      wstring getString1000();
      wstring getString2000();
      wstring getString5000();
      wstring getString10000();
      wstring getString20000();
      wstring getString50000();
      wstring getString100000();
      ::<tecnologia>::<tecnologia>util:: <prefixo>TestObject
            getTestObject();
      ::<tecnologia>::<tecnologia>util:: <prefixo>TestObjectList
            getTestObjectList(in long s);
    };
  };
};
```

Comparando as definições das interfaces em Java e em OMG IDL, é possível notar que elas foram definidas levando-se em consideração as diferenças entre os tipos primitivos

das duas linguagens: tipos primitivos `long`, `byte` e `int` de Java correspondem respectivamente aos tipos `long long`, `octet` e `long` de OMG IDL. Além disso, a classe `String` de Java corresponde ao tipo `wstring` de OMG IDL.

3.2.2 Implementação dos Serviços Remotos

As implementações dos serviços remotos nas tecnologias analisadas diferem apenas na declaração da classe que é estendida (e da interface que é implementada quando necessário) e na utilização das respectivas classes do objeto de teste e da lista. O seguinte trecho de código mostra como os serviços foram implementados:

```
public class <prefixo>TestImpl extends X [implements Y]
{
    private String str1 = "1";
    private String str10 = "1234567890";
    private String str20 = str10 + str10;
    ...
    private String str100000 = str50000 + str50000;

    private TestObjectList list1 = new TestObjectList (1);
    private TestObjectList list10 = new TestObjectList (10);
    ...
    private TestObjectList list1000 = new TestObjectList (1000);

    public String getString1() throws RemoteException
    {
      return str1;
    }

    public String getString10() throws RemoteException
    {
      return str10;
    }

    public TestObjectList getTestObjectList(int size) throws
                                                    RemoteException
    {
      switch (size)
          {
          case 1:
            return list1;
          case 10:
            return list10;
          ...
          case 1000:
            return list1000;
          }
    }
}
```

3.2.3 Implementação das Aplicações Cliente e Servidora

Com exceção de JAX-RPC, as aplicações servidoras basicamente criam uma instância do objeto remoto e a registram utilizando o serviço de nomes apropriado. Conforme mencionado anteriormente, em JAX-RPC essa funcionalidade é implementada pelo ambiente de execução de JAX-RPC.

As aplicações cliente desenvolvidas também são muito similares. A única diferença está na obtenção de uma referência para o objeto remoto. As chamadas dos procedimentos remotos são idênticas. Por exemplo:

```
...
        // Strings of 500 char
        fileName = "./" + PREFIX + "str500.txt";
        writer = new PrintWriter(new FileOutputStream(fileName));
        for (i = 0; (i < NUM_OF_SAMPLES); i++)
            {
              startTime = System.currentTimeMillis();

              s = remoteObject.getString500();

              stopTime = System.currentTimeMillis();
              elapsedTime = stopTime - startTime;
              writer.println(elapsedTime);
            }
        writer.close();
...
```

3.3 Análise dos Resultados Obtidos

Para cada tecnologia, os métodos definidos na interface do serviço remoto foram executados 100 vezes em cada cenário. As distribuições dos valores obtidos foram analisadas de forma a se determinar se a sua média representaria uma medida consistente do desempenho das tecnologias estudadas. Alguns valores foram descartados, observando-se o critério de descartar o menor número de medidas possível (no máximo 10 por cento dos maiores valores das medidas das chamadas de um método em um determinado cenário). Os valores descartados foram aqueles que estavam muito acima dos demais, tipicamente em uma ordem de grandeza maior. Possíveis explicações que podem ser mencionadas para a ocorrência desses valores discrepantes são:

- a coleta de lixo automática implementada pelas máquinas virtuais Java que as aplicações não podem determinar ou controlar a sua ativação (coleta de lixo pode ser disparada a qualquer momento por uma alocação ou liberação de memória);
- troca de contexto de processos efetuada pelo sistema operacional;
- esgotamento do *caching* de memória;

- interferência de tráfego de rede de outras aplicações;
- fragmentação da memória, que ocorre normalmente após execuções exaustivas de casos de teste.

Depois de descartados esses valores discrepantes foram calculados os tempos médios da execução das chamadas e os respectivos desvios padrão. As médias foram utilizadas nas comparações de desempenho de cada tecnologia enquanto que os desvios padrão serviram de referência para a análise dos resultados obtidos.

Por exemplo, antes dos ajustes efetuados, o desempenho médio dos métodos que devolvem cadeias de até 10.000 caracteres no cenário A pode ser representado pelo gráfico:

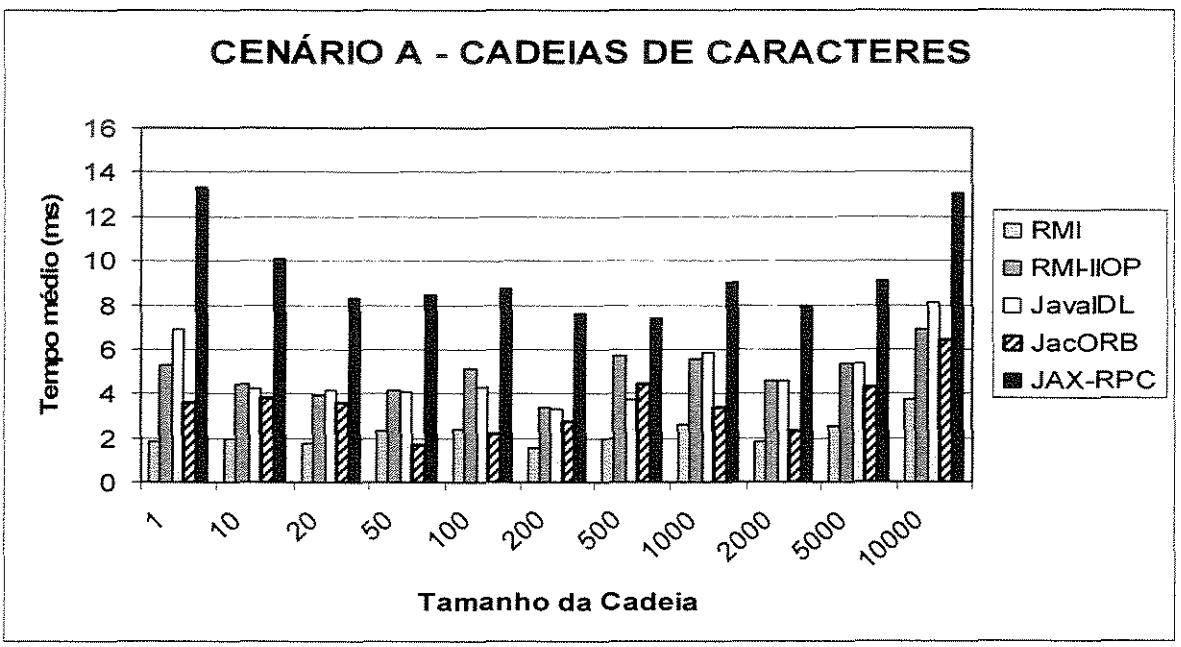

Gráfico 1 Dados Originais - Cenário A – cadeias de até 10.000 caracteres

Após os ajustes, pequenas, mas importantes alterações podem ser observadas:

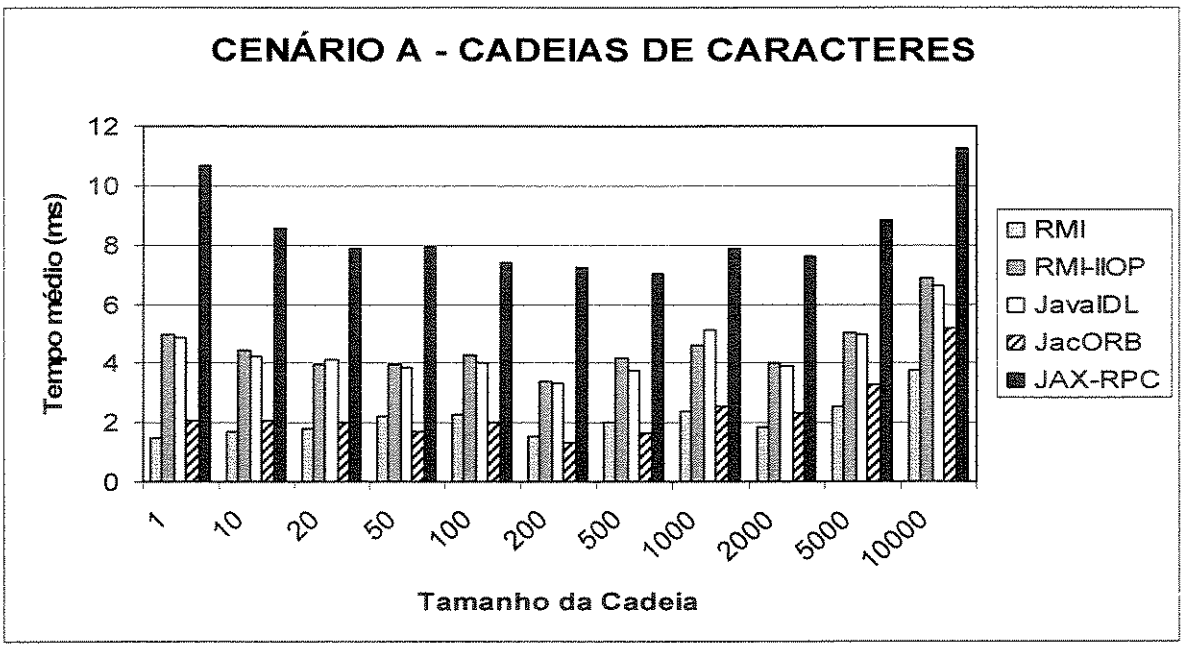

Gráfico 2 **Dados Ajustados - Cenário A – cadeias de até 10.000 caracteres**

3.4 Trabalhos Relacionados

Nesta seção são descritos dois trabalhos de comparação de tecnologias de programação distribuída.

No primeiro deles [49] os autores concentraram-se na comparação de desempenho entre RMI (Java SDK 1.1.4) e CORBA (Visigenic Visibroker for Java 3.0). O objetivo foi medir o custo adicional dessas tecnologias nas chamadas de métodos remotos.

Três cenários foram utilizados: cliente e servidor na mesma máquina, cliente e servidor em computadores separados e servidor rodando em um computador com clientes sendo executados simultaneamente em 2, 3, 4, 5, 6, 7 e 8 computadores. Em todos os cenários foi utilizada uma rede com pouca interferência de outros tipos de tráfego. A comparação entre o primeiro cenário e o segundo teve o intuito de determinar o custo adicional da comunicação em rede, enquanto que a comparação entre o segundo e o terceiro cenários teve como objetivo determinar a degradação de desempenho em um ambiente com vários clientes simultâneos.

Foram utilizados métodos com códigos de retorno simples (todos os tipos primitivos de Java) e métodos devolvendo cadeias de caracteres. Uma ferramenta de análise de código foi utilizada com o objetivo de identificar os métodos que consumiram o maior

tempo de execução.

Os resultados obtidos determinaram que nenhuma das duas tecnologias é consideravelmente mais rápida ou mais lenta do que a outra. Em cenários simples, como métodos devolvendo tipos primitivos ou cadeias de caracteres pequenas com poucos clientes simultâneos, RMI obteve um desempenho melhor. Nos casos onde vários clientes foram utilizados ao mesmo tempo CORBA demonstrou ser uma tecnologia mais robusta. Para cadeias de caracteres grandes, RMI obteve também um melhor desempenho.

No segundo trabalho analisado [50], além de uma comparação de desempenho, foi efetuada uma comparação de usabilidade e das características de diversas tecnologias de programação distribuída em Java: Java RMI (1.4.2_01), CORBA (JavaIDL 1.4.2_01 e JacORB 1.4.1), Java/ICE (1.1.1) e Java/SOAP (Apache Axis 1.1).

Os métodos testados foram divididos em dois grupos: métodos com códigos de retorno e parâmetros simples (tipos primitivos de Java) e métodos com parâmetros e códigos de retorno do tipo vetor de *bytes* de diversos tamanhos.

Embora três cenários tenham sido utilizados (cliente e servidor na mesma máquina, cliente e servidor em máquinas diferentes – rede local e Internet), os resultados de apenas um deles foram apresentados, pois o autor concluiu que as razões entre as diferenças de tempo foram mantidas em todos os cenários.

A análise de usabilidade efetuada indicou que a tecnologia mais simples de ser utilizada é Java RMI. CORBA e ICE empataram em segundo lugar, com SOAP sendo considerada a tecnologia de maior dificuldade de utilização.

Em termos de desempenho, para métodos simples CORBA (JacORB) foi pouco melhor do que RMI. ICE ficou em terceiro lugar e SOAP ocupou a última posição, com um desempenho muito inferior às demais tecnologias. Para métodos devolvendo vetores de *bytes* de diversos tamanhos, ICE obteve o melhor desempenho, seguido de perto por Java RMI e CORBA. SOAP novamente teve o pior desempenho entre as tecnologias analisadas.

Capítulo 4

Testes Comparativos

Neste capítulo são apresentados os resultados obtidos nos testes de comparação das tecnologias em estudo, efetuados nos cenários descritos no capítulo anterior. Primeiramente cada uma das tecnologias é analisada isoladamente e uma comparação é feita entre os seus desempenhos nos cenários A e B. Em seguida as tecnologias são comparadas entre si nos cenários A, B e C.

4.1 Resultados por Tecnologia

Nesta seção são comparados os resultados de cada tecnologia nos cenários A (cliente e servidor na mesma máquina) e B (cliente e servidor em máquinas "vizinhas" da mesma sub-rede) de forma a observar o impacto da rede na comunicação entre cliente e servidor.

São apresentados gráficos representando o desempenho dos métodos remotos que devolvem cadeias de caracteres (*strings*) e gráficos representando o desempenho dos métodos que devolvem listas de objetos. Os gráficos foram divididos em intervalos de forma a auxiliar sua visualização, pois há medidas de diferentes ordens de grandeza:

- cadeias de até 1.000 caracteres e cadeias de 1.000 a 100.000 caracteres;
- listas de até 200 objetos e listas de 200 a 1.000 objetos.

Cada gráfico possui quatro curvas:

- Média A: valores médios das chamadas no cenário A;
- Média B: valores médios das chamadas no cenário B;
- Mínimo A: valores mínimos das chamadas no cenário A;
- Mínimo B: valores mínimos das chamadas no cenário B.

É importante observar que durante a execução das chamadas no cenário A em todas as tecnologias, foi observado através do utilitário `top` que a taxa de utilização de UCP ficou maior do que 90% na maior parte do tempo. Essa exaustão de UCP pode ser a causa de um desempenho pior no cenário A para certas chamadas.

4.1.1 RMI

Como é possível observar no gráfico abaixo, para cadeias de até 1.000 caracteres, os tempos médios de RMI nos cenários A e B variaram em um intervalo absoluto muito restrito (de 1 a 3 ms) e na maioria das vezes (6 em 8 pontos) o desempenho no cenário B foi melhor do que no cenário A, ou seja, nesses casos não é possível concluir que a comunicação entre as máquinas do cliente e do servidor introduziu custo adicional:

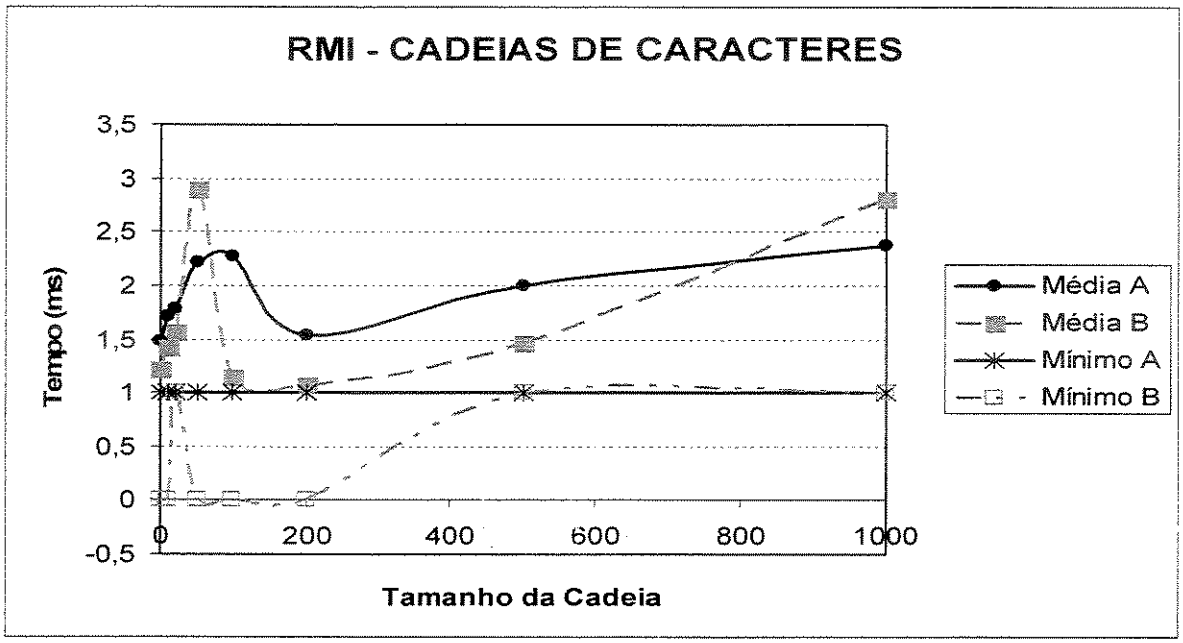

Gráfico 3 RMI - cadeias de até 1.000 caracteres

Para cadeias de 1.000 a 100.000 caracteres, o tempo médio gasto no cenário B foi consideravelmente maior, observando-se que no cenário A o tempo médio cresceu a uma taxa menor para cadeias de até 50.000 caracteres.

O desvio padrão dessas amostras variou entre 0,60 e 2,20 na maioria dos pontos. Desvios maiores foram obtidos no cenário B para cadeias de 50 caracteres (3,65) e para cadeias entre 10.000 (6,48) e 100.000 (18,65) caracteres. No cenário A, desvios maiores foram identificados apenas para cadeias de 50.000 (13,95) e 100.000 (43,39) caracters.

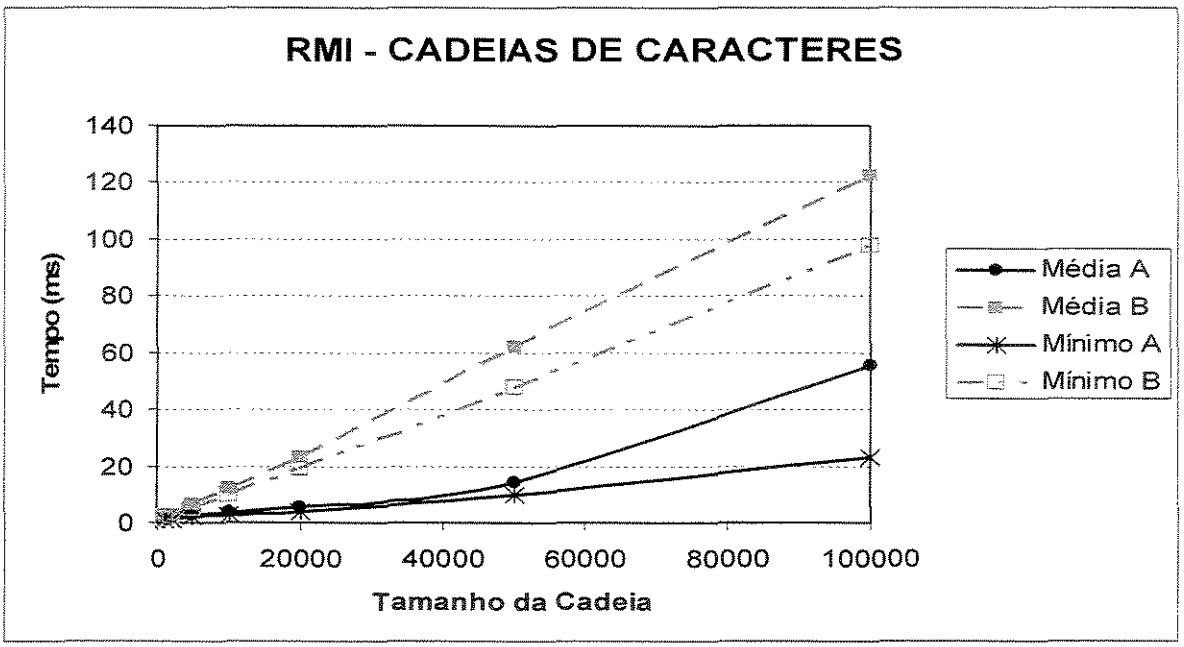

Gráfico 4 **RMI - cadeias de 1.000 a 100.000 caracteres**

Para listas de até 50 objetos, o desempenho de RMI nos cenários A e B foi muito similar, com exceção de listas contendo apenas 1 objeto, onde o desempenho no cenário A foi muito pior. Para listas de 100 e 200 objetos o desempenho no cenário B foi pior:

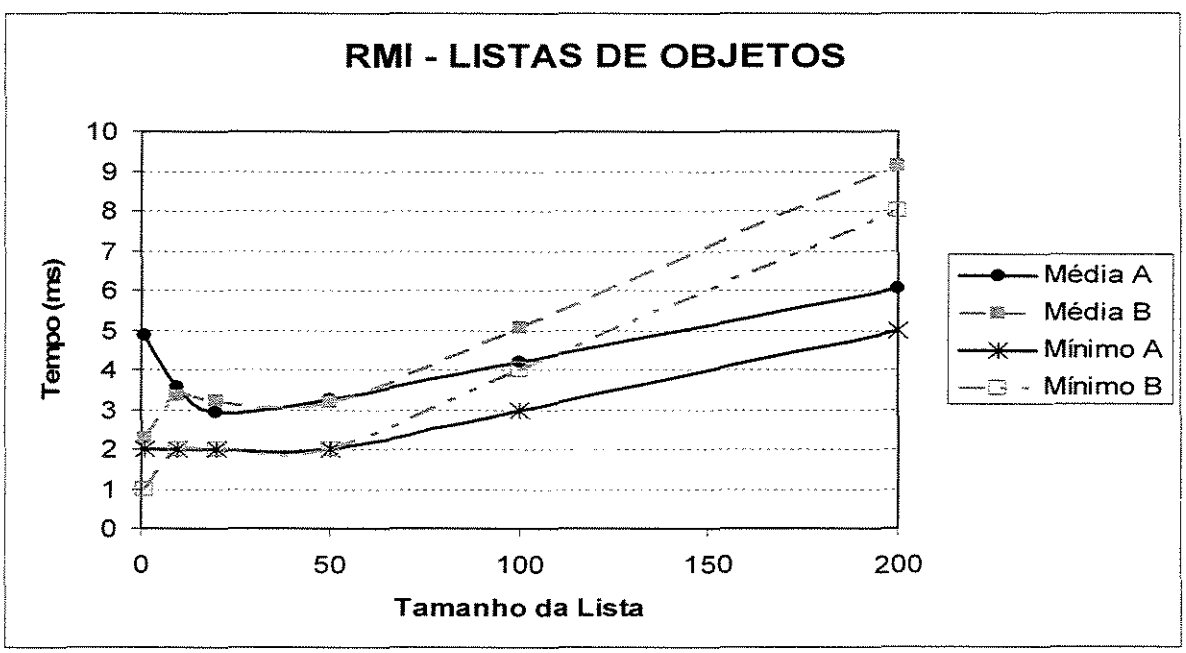

Gráfico 5 **RMI – listas de até 200 objetos**

Os desvios padrão das amostras das chamadas devolvendo listas de até 200 objetos foram similares, exceto listas de 1 objeto no cenário A, que apresentou um desvio muito

acima dos demais (4,02),

Para listas de mais de 200 objetos, o desempenho no cenário A voltou a ser pior, não sendo possível concluir que a comunicação entre as máquinas do cliente e do servidor introduz um custo adicional. Entretanto, tanto no cenário A quanto no cenário B os desvios padrão cresceram em função do tamanho da lista de objetos, sendo que o crescimento no cenário B foi um pouco maior.

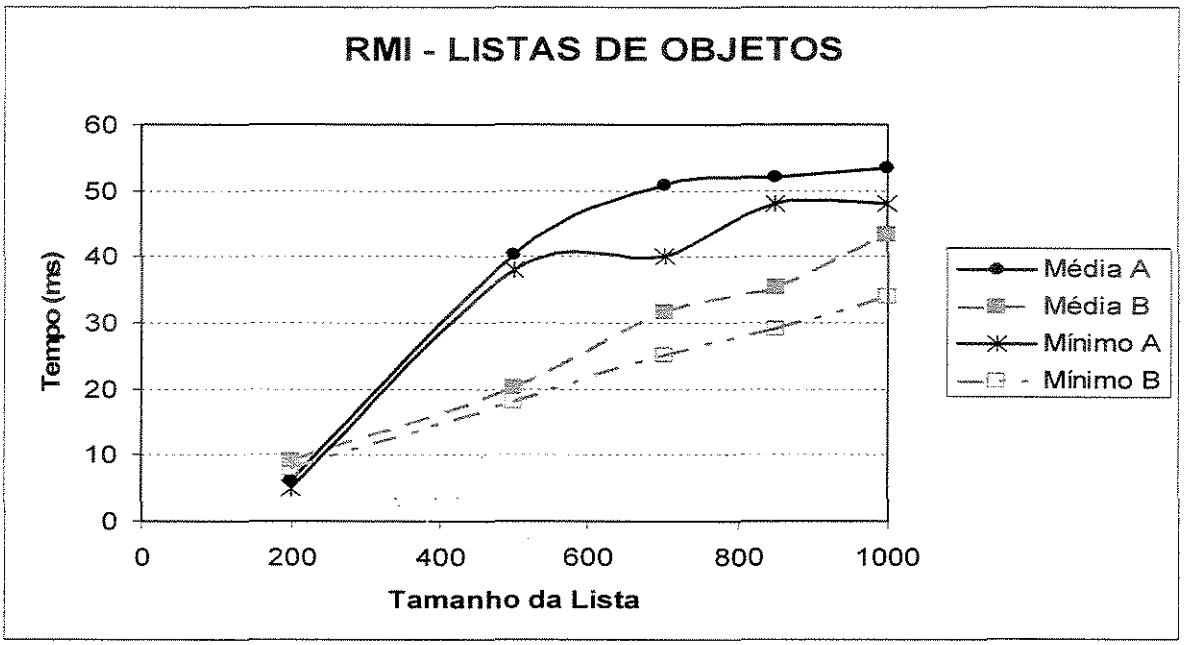

Gráfico 6 RMI – listas de 200 a 1.000 objetos

De uma forma geral, para RMI, é possível concluir que apenas nos casos de envios de cadeias de caracteres maiores do que 1.000 caracteres a comunicação entre as máquinas do cliente e do servidor introduziu custo adicional no desempenho das chamadas de procedimento remoto.

4.1.2 RMI-IIOP

Para cadeias de caracteres de até 500 caracteres o desempenho de RMI-IIOP no cenário A foi pior do que no cenário B, embora a diferença tenha variado em um intervalo pequeno (valores relativos):

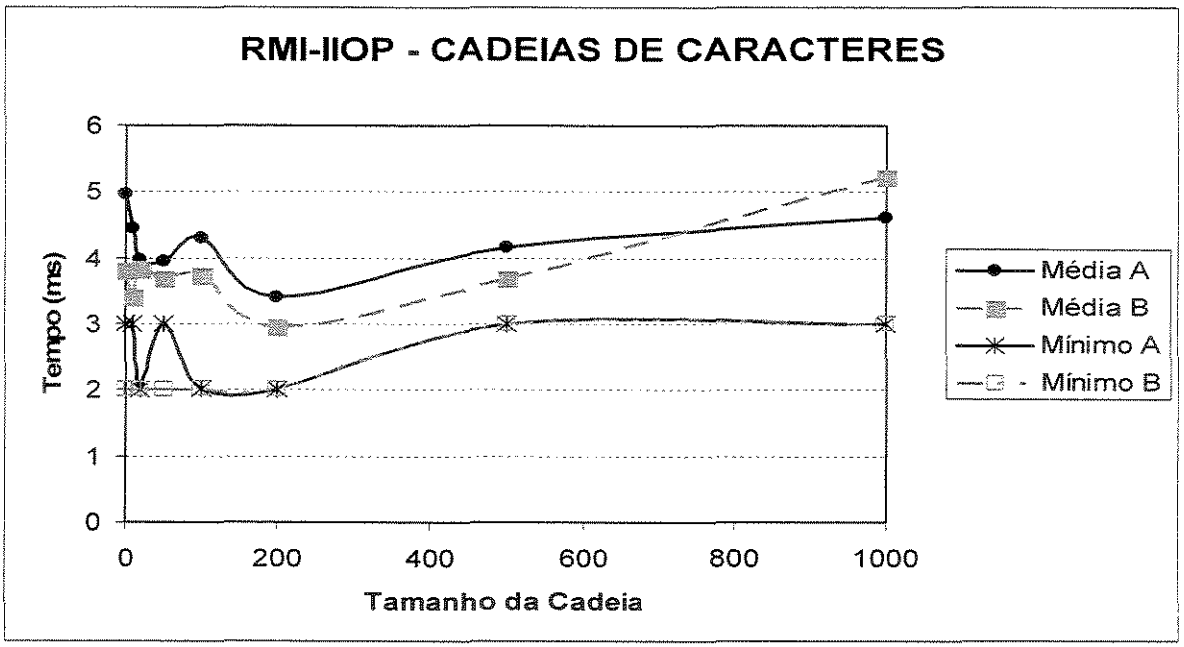

Gráfico 7 **RMI-IIOP - cadeias de até 1.000 caracteres**

Para cadeias de 1.000 a 100.000 caracteres, o desempenho de RMI-IIOP no cenário A foi consideravelmente melhor. O tempo médio no cenário B cresceu a uma taxa maior até 50.000 caracteres, como é possível observar no gráfico a seguir:

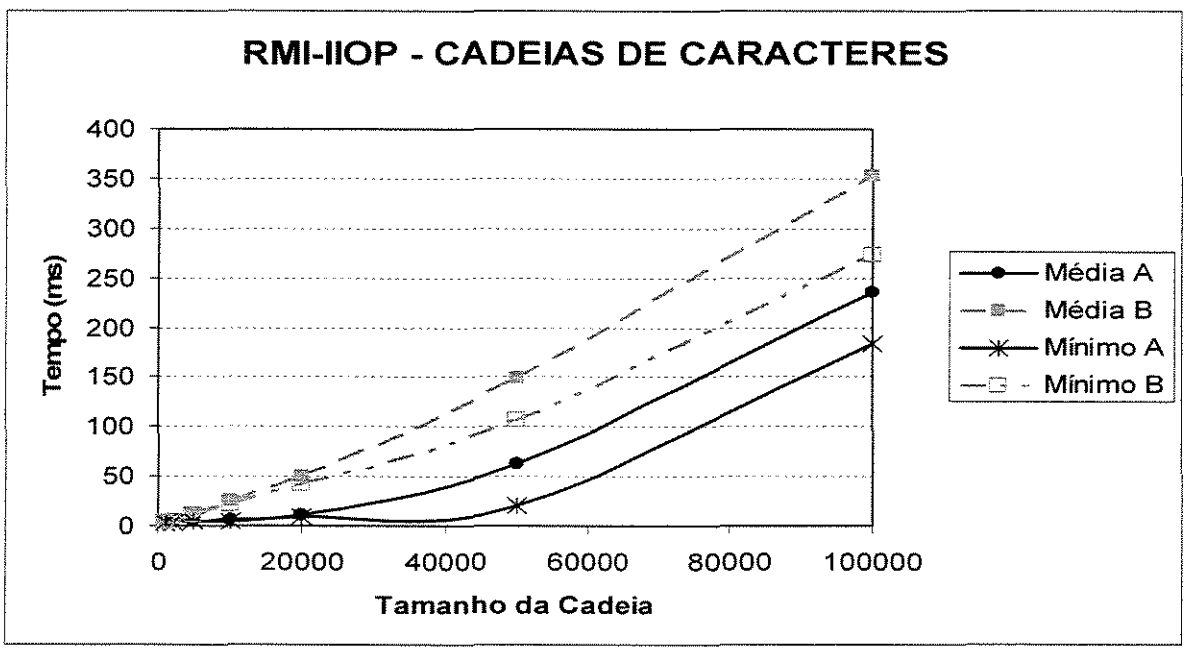

Gráfico 8 **RMI-IIOP - cadeias de 1.000 a 100.000 caracteres**

Para as chamadas devolvendo cadeias de até 20.000 caracteres, o desvio padrão

variou num intervalo relativamente pequeno (entre 0,98 e 3,62), com exceção de cadeias de 20.000 caracteres no cenário B (7,86). Entretanto, essa variação não foi proporcional ao tamanho das cadeias, apesar do desvio ser próximo nos dos cenários para cadeias do mesmo tamanho. Apenas para cadeias de 50.000 e 100.000 caracteres o desvio padrão observado foi bem maior (entre 43,39 e 58,43).

Na comparação de desempenho para os métodos que devolvem listas de objetos, o desempenho no cenário B foi pior na maioria dos pontos analisados, mas a diferença não ultrapassou 13% em nenhum ponto. Nos pontos iniciais onde o desempenho no cenário A foi pior, a diferença relativa variou no mesmo intervalo.

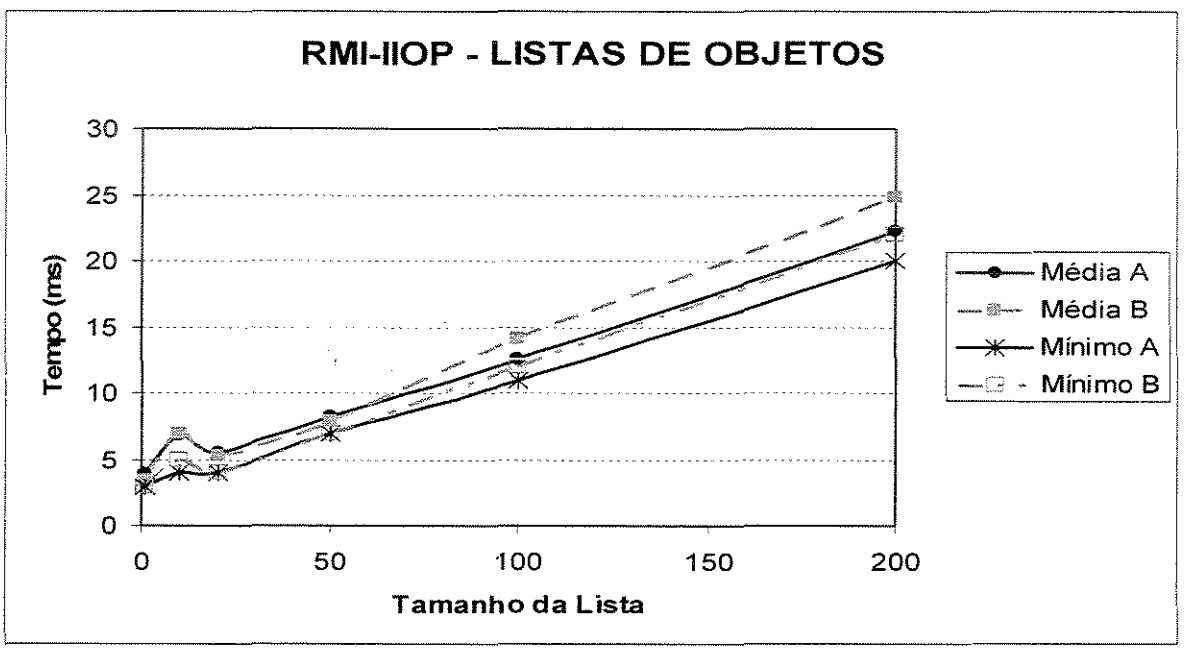

Gráfico 9 RMI-IIOP – listas de até 200 objetos

Para listas de 200 a 700 objetos o tempo médio no cenário B foi maior, entretanto o crescimento nos dois cenários ocorreu numa taxa bem próxima. Durante a execução dos testes de métodos que devolvem listas de objetos em RMI-IIOP, não foi possível medir o tempo gasto nas camadas de métodos devolvendo listas com mais do que 700 objetos. Ocorreu um erro de execução com uma exceção sendo lançada pela biblioteca de RMI-IIOP (`org.omg.CORBA.MARSHAL: Unable to read value from underlying bridge`).

Para listas de até 200 objetos o desvio padrão também não foi proporcional ao tamanho da lista, variando entre 0,70 e 3,83, apresentando valores próximos para listas do mesmo tamanho nos dois cenários. Apenas para listas de 500 e 700 objetos o desvio

padrão observado esteve num patamar muito acima (respectivamente, em torno de 12 e em torno de 53).

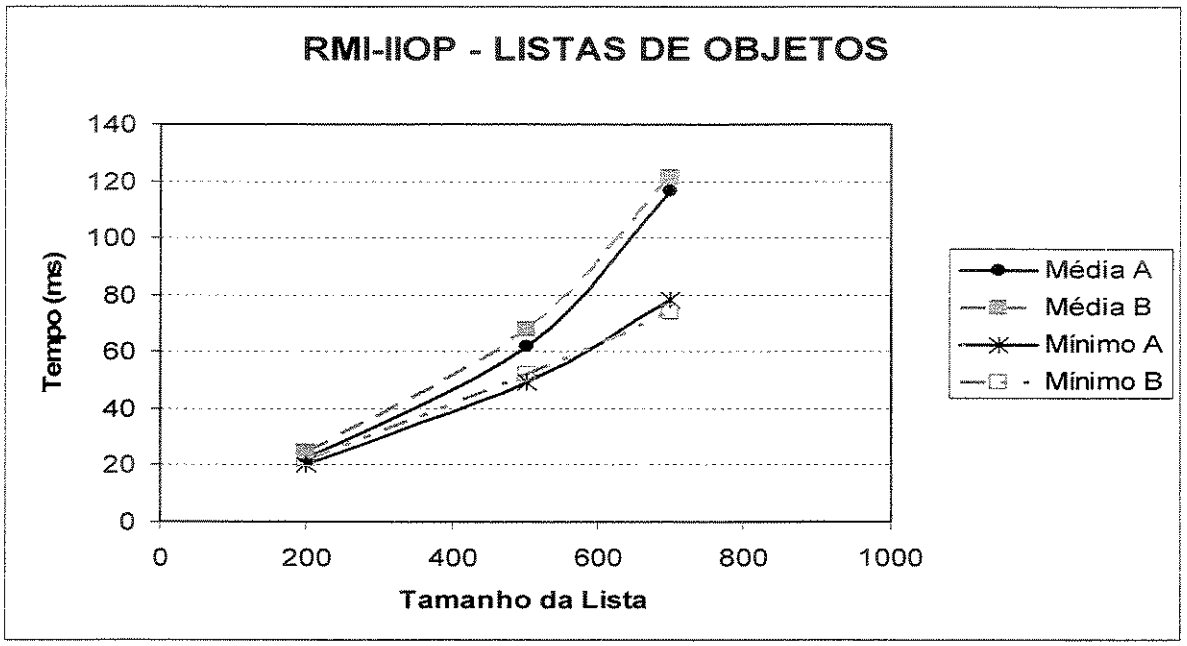

Gráfico 10 RMI-IIOP – listas de 200 a 1.000 objetos

Em resumo, a variação do desempenho de RMI-IIOP nos cenários A e B foi pequena. O desempenho no cenário A só foi pior para cadeias de até 500 caracteres. Nos demais cenários onde o desempenho no cenário B foi pior, é possível concluir que a comunicação entre as máquinas do cliente e do servidor acarretou um custo adicional no desempenho.

4.1.3 JavaIDL

Para cadeias de até 200 caracteres, o desempenho de JavaIDL no cenário A foi pior, como pode ser observado no gráfico a seguir:

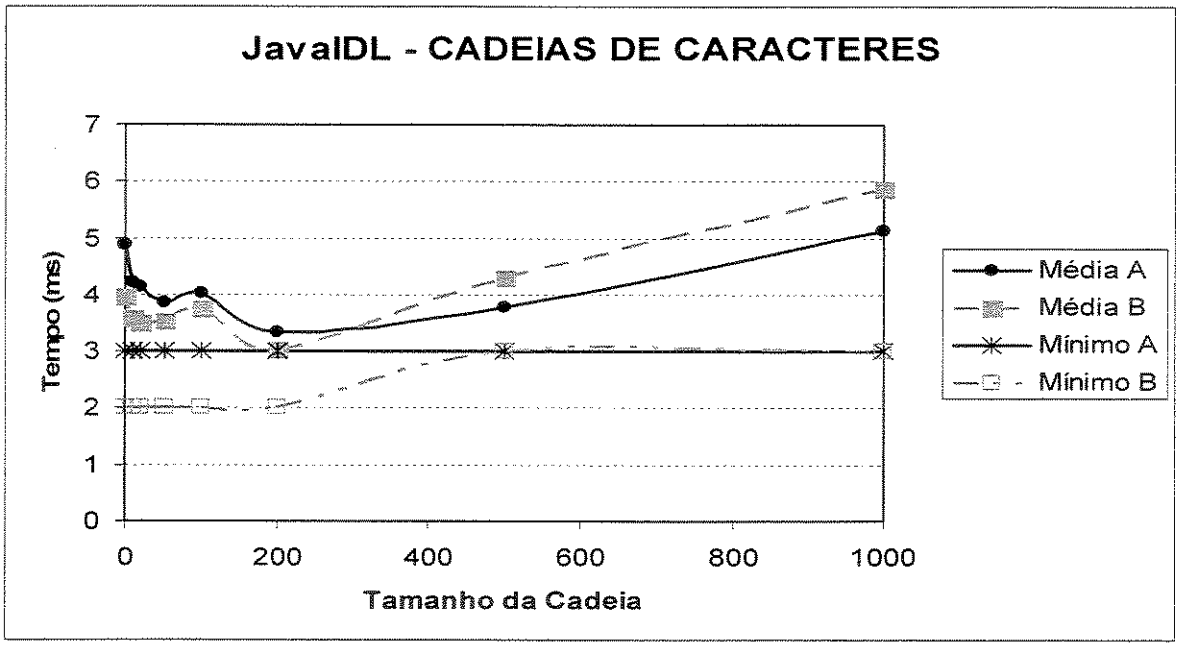

Gráfico 11 **JavaIDL - cadeias de até 1.000 caracteres**

Analisando-se juntamente os gráficos 11 e 12 (acima e abaixo), é possível notar que o desempenho no cenário B foi pior para cadeias de mais do que 200 caracteres, com a diferença relativa crescendo muito de 500 até 20.000 caracteres, e se mantendo num intervalo menor para 50.000 e 100.000 caracteres.

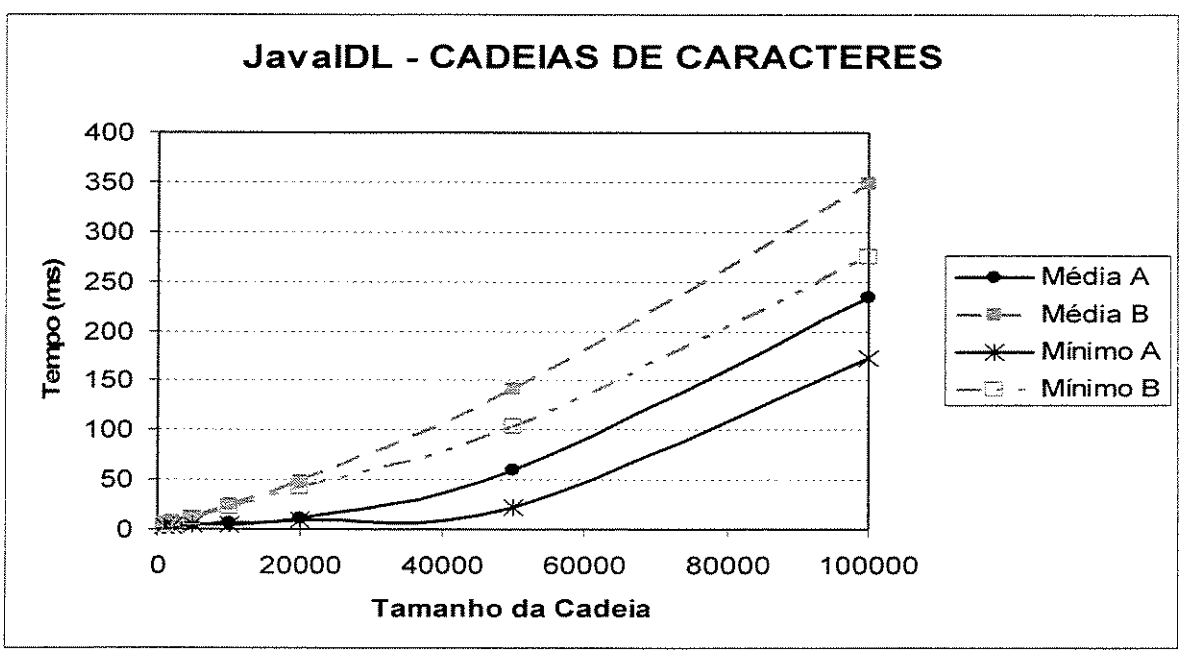

Gráfico 12 **JavaIDL - cadeias de 1.000 a 100.000 caracteres**

Os desvios padrão das amostras das chamadas devolvendo cadeias de até 1.000 caracteres foram similares se comparados os valores obtidos no cenário A com os obtidos no cenário B, variando no intervalo entre 1,03 e 3,77. Para cadeias entre 2.000 e 20.000 caracteres, os valores dos desvios no cenário B foram mais do que o dobro dos respectivos valores obtidos no cenário A. Para cadeias de 50.000 e 100.000 caracteres os desvios foram bem mais elevados (entre 43,15 e 52,16).

Os próximos dois gráficos apresentam o desempenho de JavaIDL para os métodos que devolvem listas de objetos:

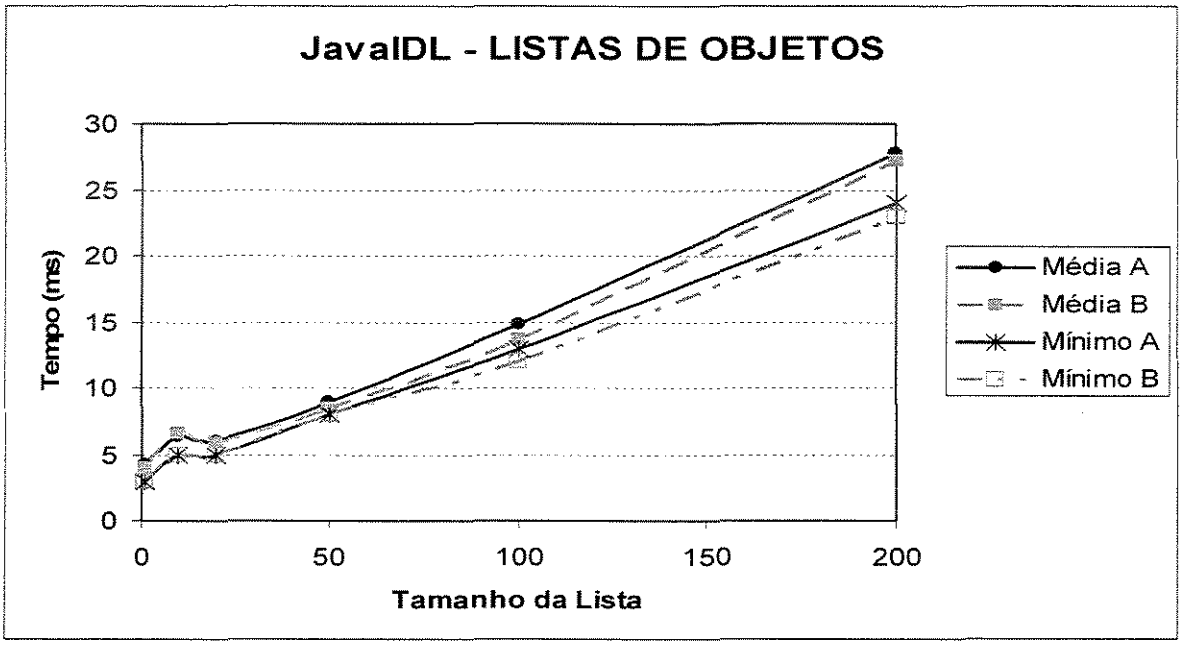

Gráfico 13 JavaIDL – listas de até 200 objetos

Apenas em um ponto observado o desempenho no cenário B foi pior: listas de 10 objetos. Em todos os demais pontos, o desempenho no cenário A foi pior, variando em um intervalo relativamente pequeno (até 15%).

Para listas de até 200 objetos, os desvios padrão obtidos no cenário A (entre 1,94 e 4,86) foram maiores do que os respectivos desvios obtidos no cenário B (entre 0,95 e 3,82), com exceção de listas de 10 objetos. Para listas de 500 objetos os desvios foram próximos nos dois cenários. Para listas entre 700 e 1.000 objetos os desvios variaram num patamar bem acima (cenário A entre 55,82 e 106,17 – cenário B entre 52,91 e 82,73).

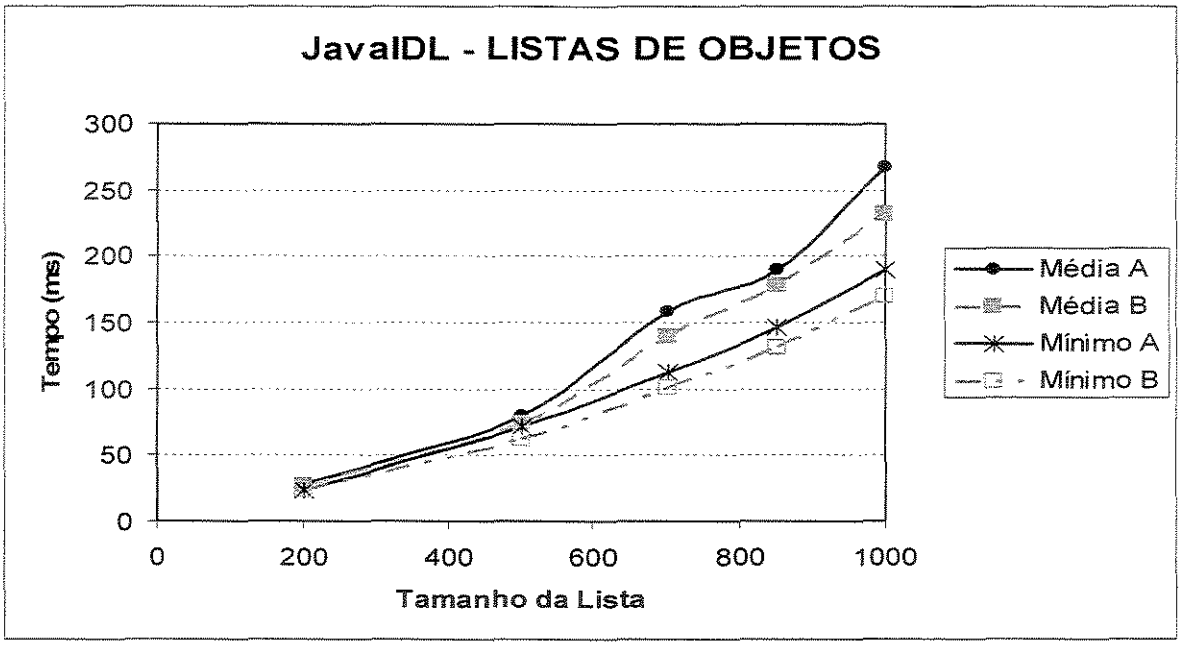

Gráfico 14 **JavaIDL – listas de 200 a 1.000 objetos**

Sendo assim, é possível afirmar apenas que a comunicação entre as máquinas do cliente e do servidor afetou o desempenho de JavaIDL para cadeias de mais do que 200 caracteres. Nos demais casos o cenário A apresentou um desempenho pior.

4.1.4 JacORB

Para as chamadas de métodos devolvendo cadeias de caracteres, o desempenho de JacORB no cenário B só não foi pior para cadeias de 1 e 10 caracteres:

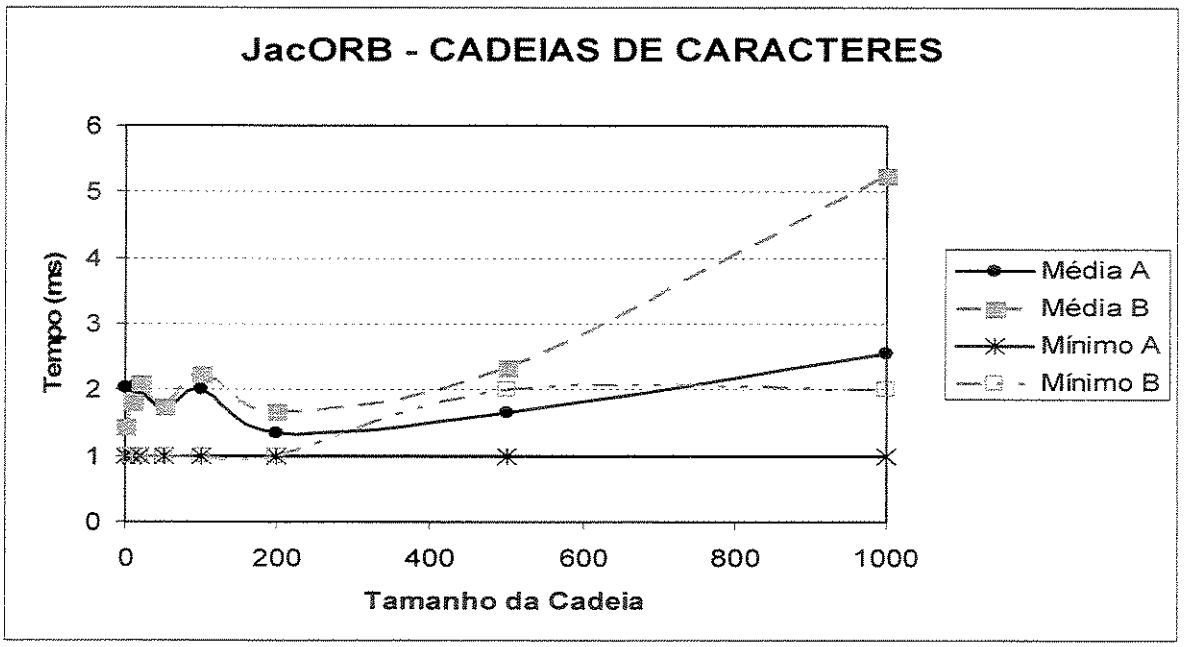

Gráfico 15 **JacORB - cadeias de até 1.000 caracteres**

Em todos os outros pontos o desempenho no cenário B foi pior, com tempos médios crescendo a uma taxa bem maior do que no cenário A. O ponto de pior desempenho relativo no cenário B foi para cadeias de 20.000 caracteres.

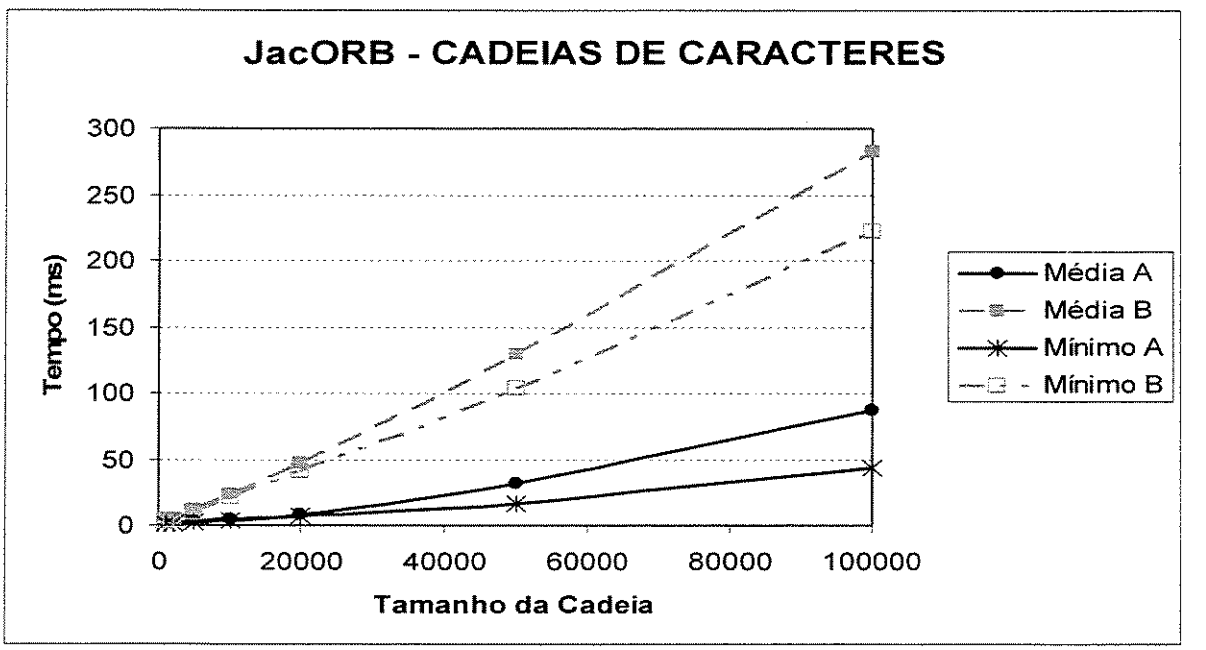

Gráfico 16 **JacORB - cadeias de 1.000 a 100.000 caracteres**

Os desvios padrão observados foram relativamente pequenos e similares se

comparados entre os valores obtidos para cadeias de até 20.000 caracteres no cenário A e B (entre 0,94 e 6,12). Esses desvios não variaram em função do tamanho das cadeias. Para cadeias de mais de 20.000 caracteres os desvios observados foram bem maiores: em torno de 31 para cadeias de 50.000 caracteres e em torno de 45,50 para cadeias de 100.000 caracteres.

Para os métodos que devolvem listas de objetos, o desempenho de JacORB foi pior no cenário A apenas para listas contendo 1 objeto:

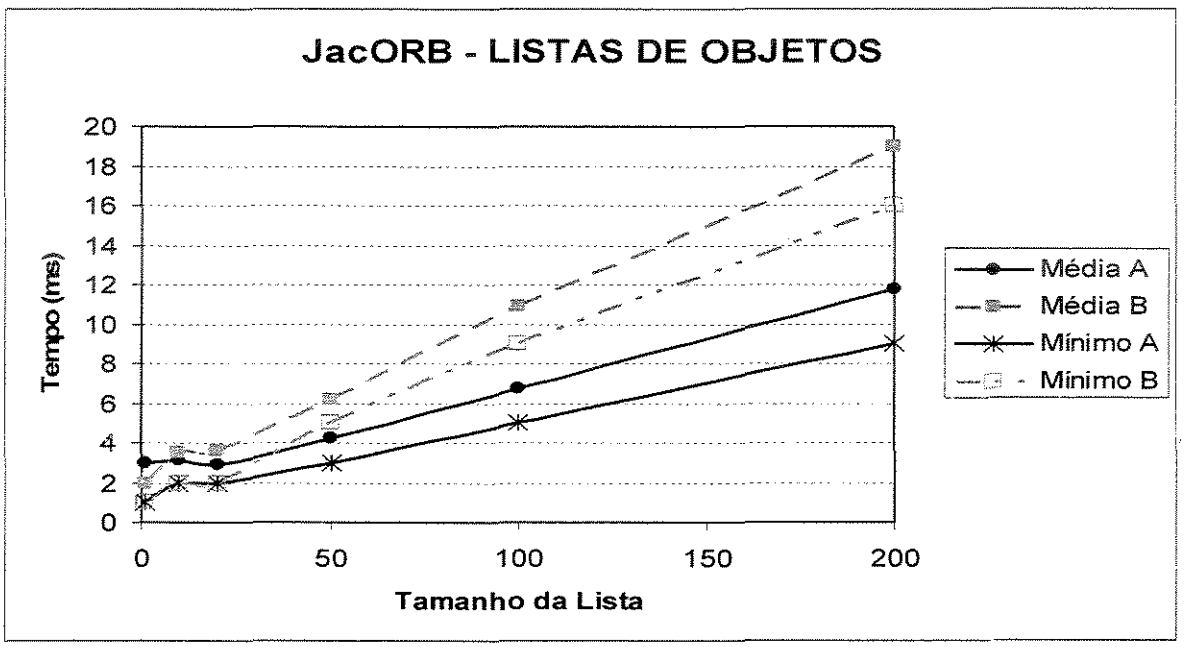

Gráfico 17 JacORB – listas de até 200 objetos

Em todos os outros pontos o desempenho no cenário B foi pior, com tempos médios crescendo a uma taxa maior. O ponto de pior desempenho relativo foi para listas de 100 objetos.

Os desvios padrão observados foram similares quando comparados os valores obtidos para listas de mesmo tamanho nos cenários A e B. Além disso, tanto no cenário A quanto no cenário B, o desvio padrão de uma forma geral cresceu em função do tamanho das listas de objetos. Por exemplo, no cenário B o desvio variou entre 2,06 e 17,70. As exceções foram listas de 1 e 700 objetos no cenário A, que apresentaram pontos fora dessa curva de crescimento do desvio padrão. O mesmo pode ser notado nos gráficos de tempo médio das chamadas (acima e abaixo).

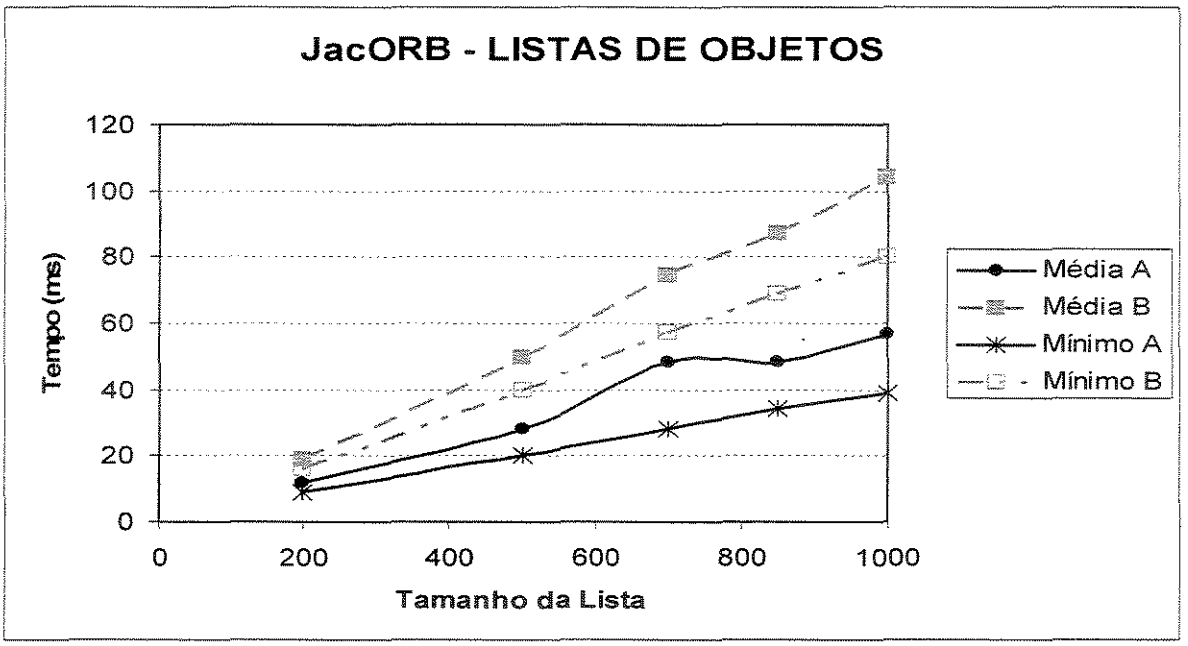

Gráfico 18 **JacORB – listas de 200 a 1.000 objetos**

Em resumo, é possível concluir que na grande maioria dos casos testados a comunicação entre as máquinas do cliente e do servidor afetaram negativamente o desempenho de JacORB.

4.1.5 JAX-RPC

Para cadeias de até 500 caracteres, o desempenho de JAX-RPC no cenário A foi pior, embora a variação tenha ocorrido em um intervalo pequeno:

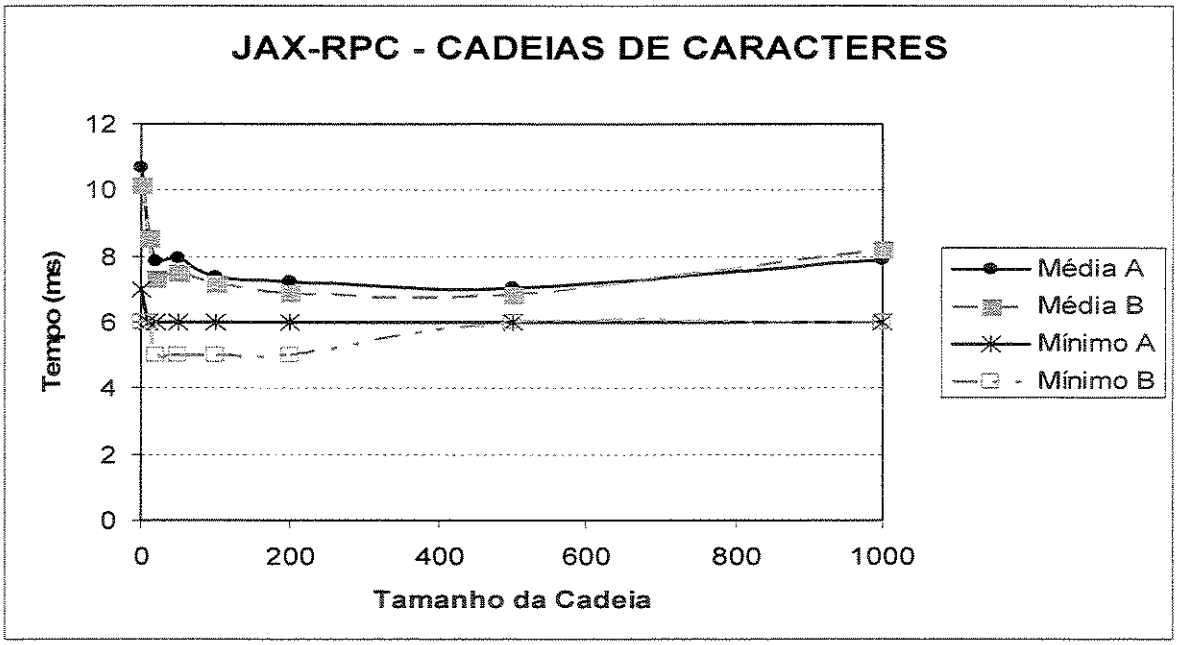

Gráfico 19 **JAX-RPC - cadeias de até 1.000 caracteres**

Para cadeias com mais de 500 caracteres, o desempenho no cenário B foi pior, com tempos médios crescendo a uma taxa maior do que no cenário A:

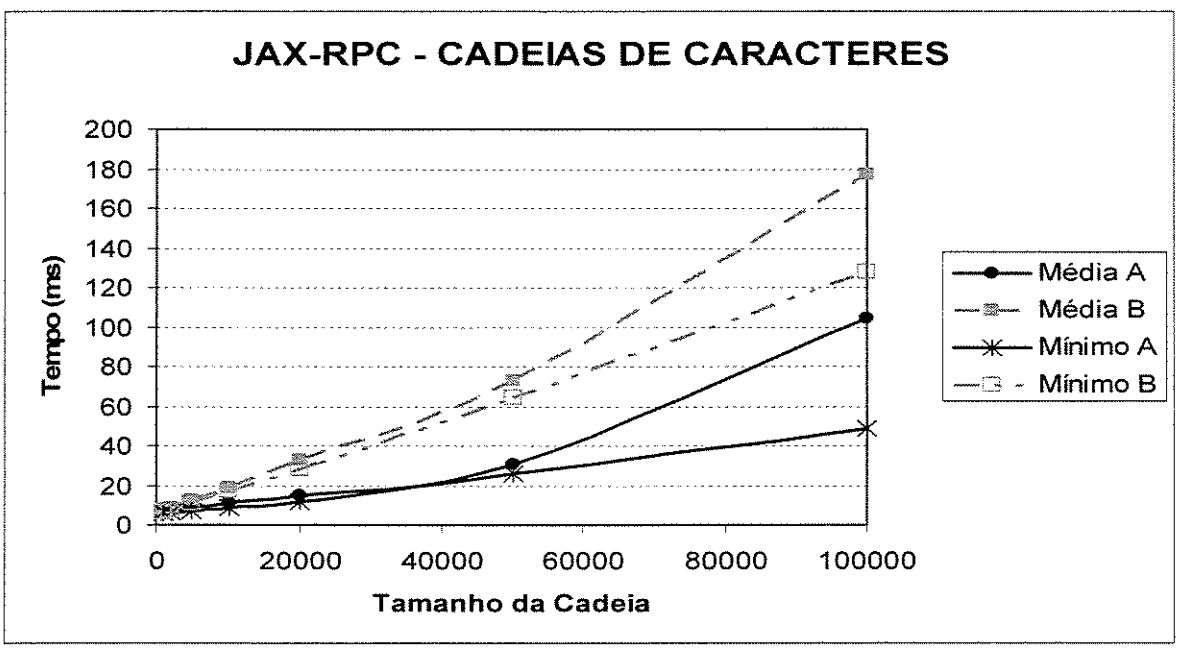

Gráfico 20 **JAX-RPC - cadeias de 1.000 a 100.000 caracteres**

O desvio padrão observado para cadeias de até 10.000 caracteres foi relativamente pequeno (menor do que 5). Para as cadeias maiores o desvio obtido foi bem maior (em

torno de 11 para 50.000 caracteres e em torno de 70 para cadeias de 100.000 caracteres).

Para os métodos que devolvem listas de objetos, o desempenho no cenário A foi pior apenas para listas contendo 1 objeto:

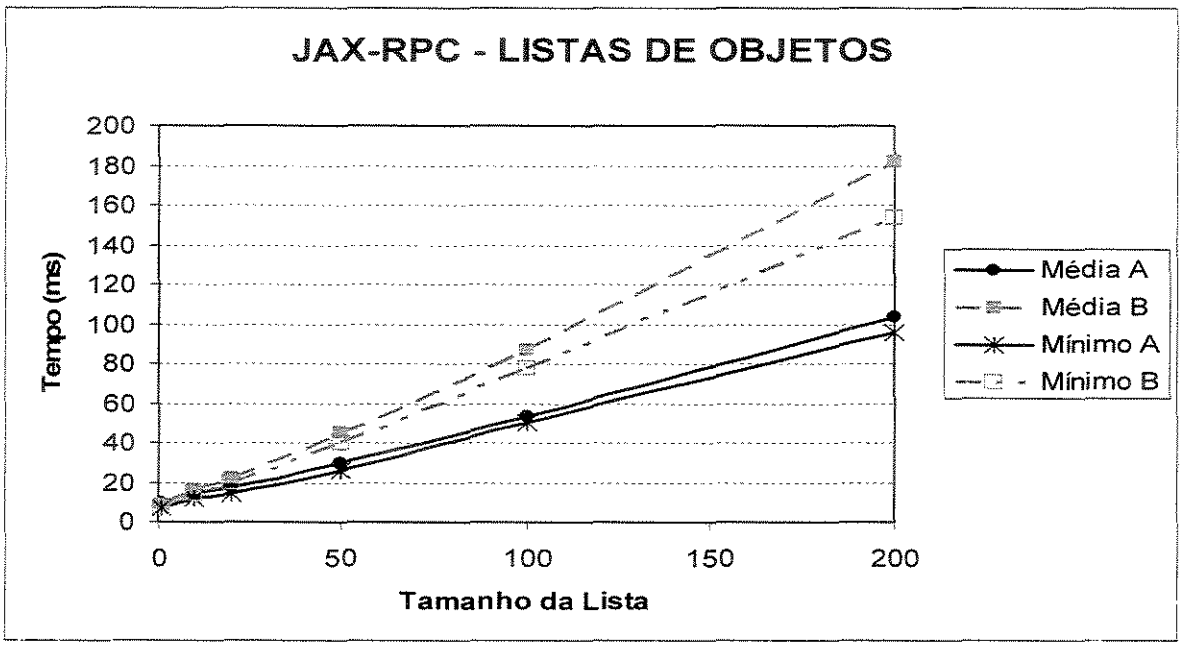

Gráfico 21 JAX-RPC – listas de até 200 objetos

Para a maioria dos pontos observados, o desempenho no cenário B foi pior, sendo que a maior diferença relativa foi para listas de 200 objetos.

Os desvios padrão observados foram relativamente pequenos para listas de até 50 objetos no cenário B e para listas de até 100 objetos no cenário A (desvios abaixo de 4,50). Para listas de 200 a 1.000 objetos no cenário B, os desvios variaram aleatoriamente entre 23,15 e 75,45, enquanto que no cenário A a variação foi de 7,56 a 90,48 (aleatória). Vale a pena notar que os maiores desvios obtidos nos dois cenários foram para listas de 500 e 700 objetos, As curvas dos tempos médios também apresentam um crescimento diferenciado nesses pontos, como é possível observar no gráfico a seguir:

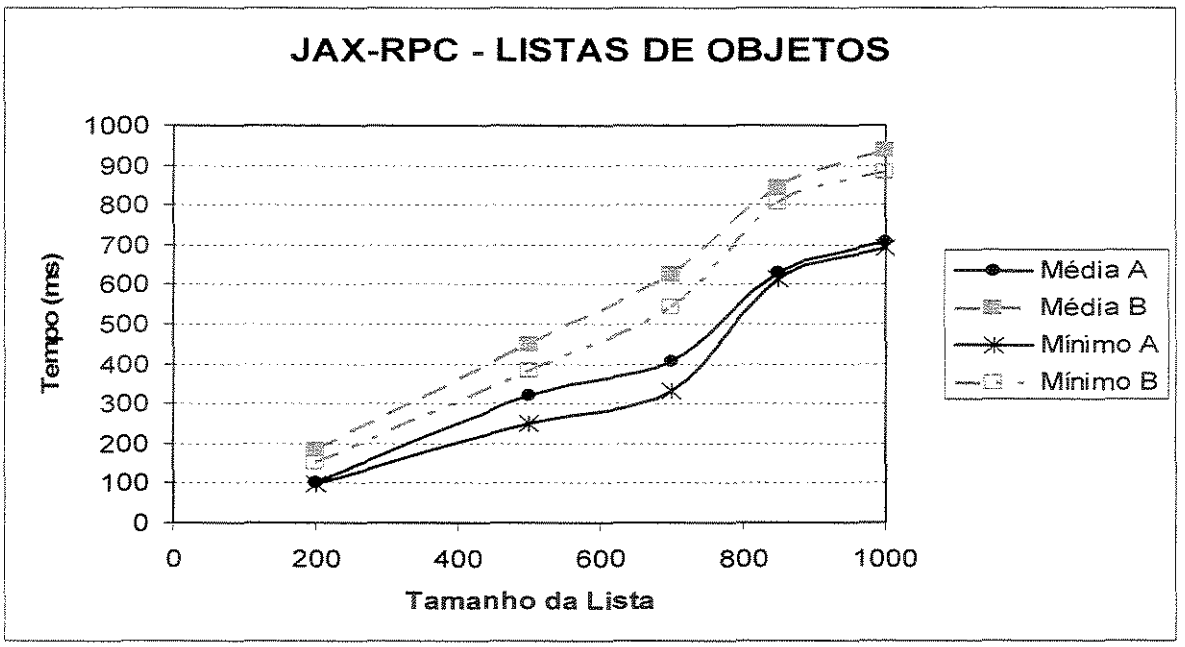

Gráfico 22 **JAX-RPC – listas de 200 a 1.000 objetos**

Em resumo, é possível concluir que para cadeias de comprimento maior ou igual a 1.000 caracteres e para listas contendo 10 objetos ou mais a comunicação entre as máquinas do cliente e do servidor afetaram negativamente o desempenho de JAX-RPC.

4.2 Resultados por Cenário

Nesta seção são comparados os resultados entre as tecnologias nos cenários A (cliente e servidor na mesma máquina), B (cliente e servidor em máquinas "vizinhas" da mesma sub-rede) e C (cliente e servidor em máquinas de redes diferentes – geograficamente distantes). O objetivo é comparar desempenho entre as tecnologias em cada cenário.

São apresentados gráficos representando o desempenho dos métodos remotos que devolvem cadeias de caracteres (*strings*) e gráficos representando o desempenho dos métodos que devolvem listas de objetos. Os gráficos foram divididos em intervalos de forma a auxiliar sua visualização, pois há medidas de diferentes ordens de grandeza:

- cadeias de até 10.000 caracteres e cadeias de 10.000 a 100.000 caracteres para os cenários A e B; cadeias de até 1.000 caracteres e cadeias de 1.000 a 100.000 caracteres para o cenário C;
- listas de até 200 objetos e listas de 200 a 1.000 objetos.

4.2.1 Cenário A

No gráfico a seguir, é possível observar que para cadeias de até 10.000 caracteres, o desempenho de RMI foi melhor do que as demais tecnologias em quase todos os pontos, só sendo pior do que JacORB para cadeias de 50, 100, 200 e 500 caracteres. JacORB obteve o segundo melhor desempenho, ficando próximo ao desempenho de RMI até 1.000 caracteres. RMI-IIOP e JavaIDL tiveram um desempenho muito próximo, em um patamar bem acima de RMI e JacORB. JAX-RPC teve os piores resultados, com médias muito maiores do que as de RMI, ficando em um terceiro patamar acima das demais tecnologias. É interessante notar que todas as curvas possuem um traçado similar.

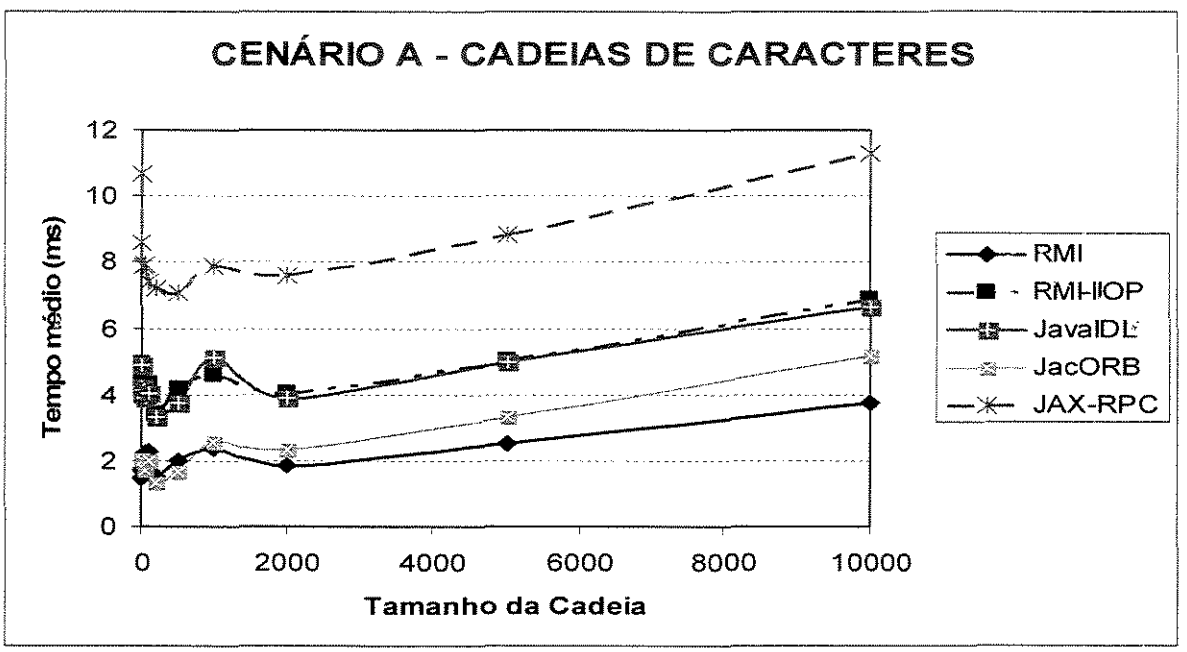

Gráfico 23 **Cenário A – cadeias de até 10.000 caracteres**

Até 20.000 caracteres o desempenho seguiu a mesma tendência apresentada no gráfico anterior. Entretanto, para cadeias de 50.000 e 100.000 caracteres o desempenho de RMI-IIOP e JavaIDL foi muito pior do que o das demais tecnologias. Para esses mesmos pontos JAX-RPC teve um desempenho mais próximo de JacORB, ambos piores do que RMI.

Para cadeias de até 20.000 caracteres os desvios padrão observados foram relativamente pequenos (menores do que 4,45), sendo que RMI apresentou o menor desvio

padrão na grande maioria dos pontos. Para cadeias de 50.000 caracteres, os menores desvios foram obtidos por RMI e JAX-RPC (em torno de 12), seguidos de JacORB (30,73) e num patamar mais alto, RMI-IIOP e JavaIDL (em torno de 55). Para cadeias de 100.000 caracteres, as tecnologias apresentaram desvios entre 43,39 e 49,12, com exceção de JAX-RPC (77,56).

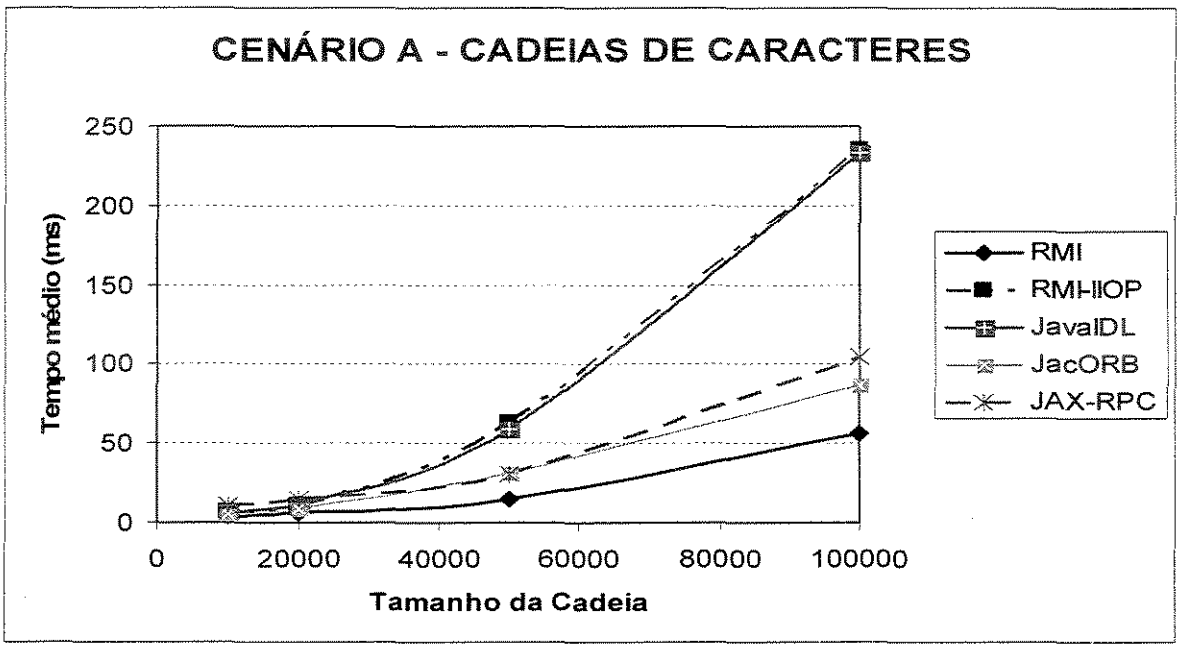

Gráfico 24 **Cenário A - cadeias de 10.000 a 100.000 caracteres**

Para listas de até 200 objetos, o desempenho de RMI foi melhor na maioria dos pontos, seguido de JacORB. JavaIDL e RMI-IIOP novamente tiveram um desempenho similar, acima de RMI e JacORB. O desempenho de JAX-RPC foi consideravelmente pior do que todas as outras tecnologias, com tempos médios crescendo a uma taxa muito maior.

Os desvios padrão para listas de até 200 objetos ficaram abaixo de 4,87 em todos os pontos, com exceção de listas de 200 objetos em JAX-RPC (7,56).

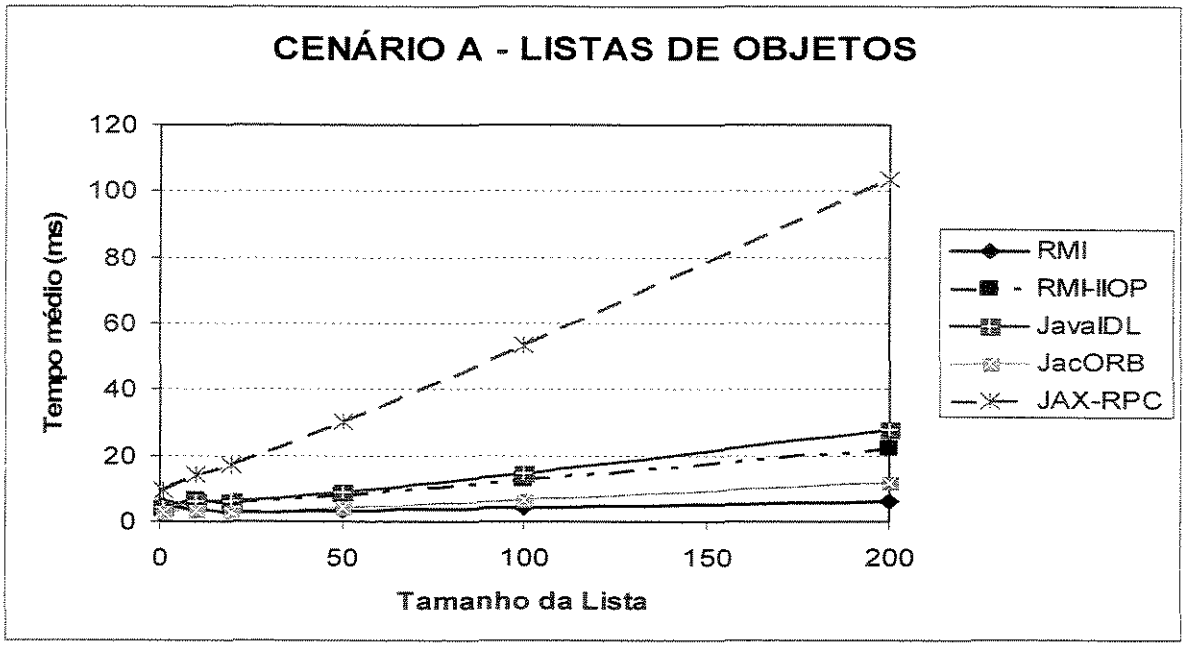

Gráfico 25 **Cenário A – listas de até 200 objetos**

Para listas de 500 a 1.000 objetos, os desempenhos de RMI e JacORB foram muito parecidos, sendo RMI melhor em alguns pontos e JacORB melhor em outros. RMI-IIOP e JavaIDL novamente tiveram um desempenho similar. Entretanto, conforme mencionado anteriormente, RMI-IIOP apresentou um erro de execução nas chamadas de métodos devolvendo listas de 850 e 1.000 de objetos. JAX-RPC manteve a tendência observada no gráfico anterior, chegando até a um desempenho 1225% pior do que RMI.

Na maioria dos pontos, RMI e JacORB apresentaram os menores desvios padrão. Desvios muito maiores foram observados para RMI-IIOP (700 objetos), JavaIDL (700 a 1.000 objetos) e JAX-RPC (500 e 700 objetos).

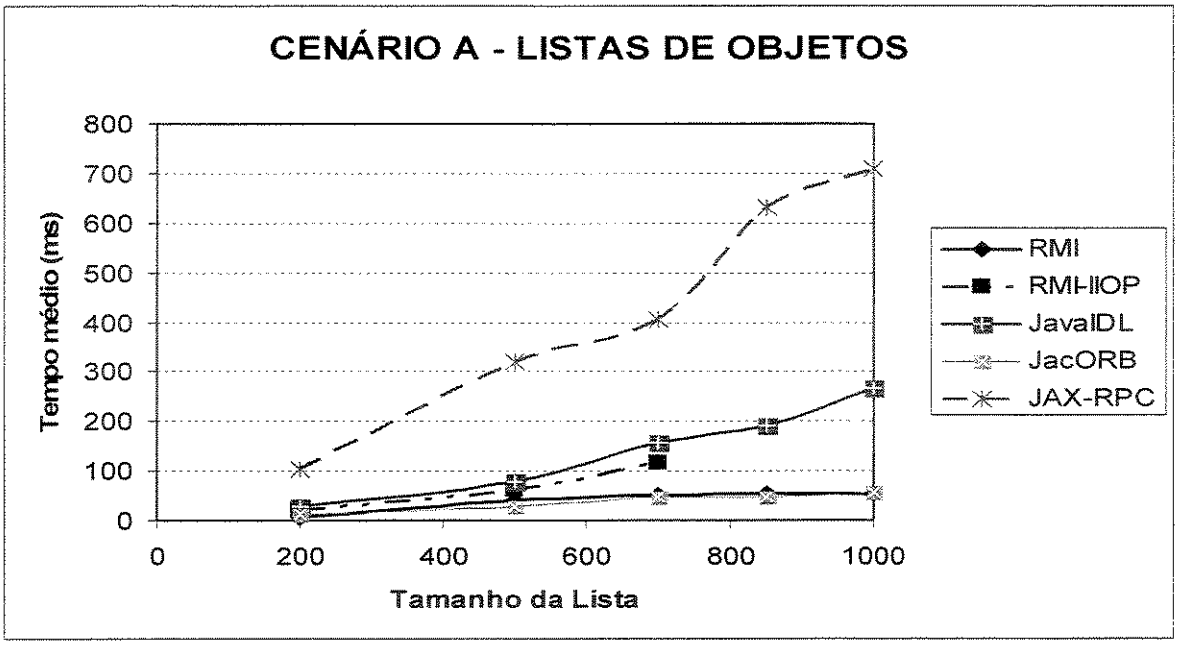

Gráfico 26 **Cenário A – listas de 200 a 1.000 objetos**

Em resumo, no cenário A RMI apresentou um desempenho melhor do que as demais tecnologias na grande maioria dos pontos. JacORB foi a tecnologia que conseguiu o desempenho mais próximo a RMI. JavaIDL e RMI-IIOP tiveram um desempenho similar, ficando em um patamar acima de RMI e JacORB na maioria dos casos. JAX-RPC teve o pior desempenho na maioria dos pontos, superando apenas JavaIDL e RMI-IIOP para cadeias de 50.000 e 100.000 caracteres.

4.2.2 Cenário B

Para cadeias de até 2.000 caracteres, RMI apresentou um desempenho melhor, seguido de JacORB, RMI-IIOP e JavaIDL que tiveram desempenhos próximos. Neste intervalo JAX-RPC apresentou um desempenho em um patamar acima dos demais. Para cadeias de 5.000 caracteres, RMI continuou com um desempenho melhor e os desempenhos de todas as demais tecnologias se aproximaram, como se observa no gráfico a seguir.

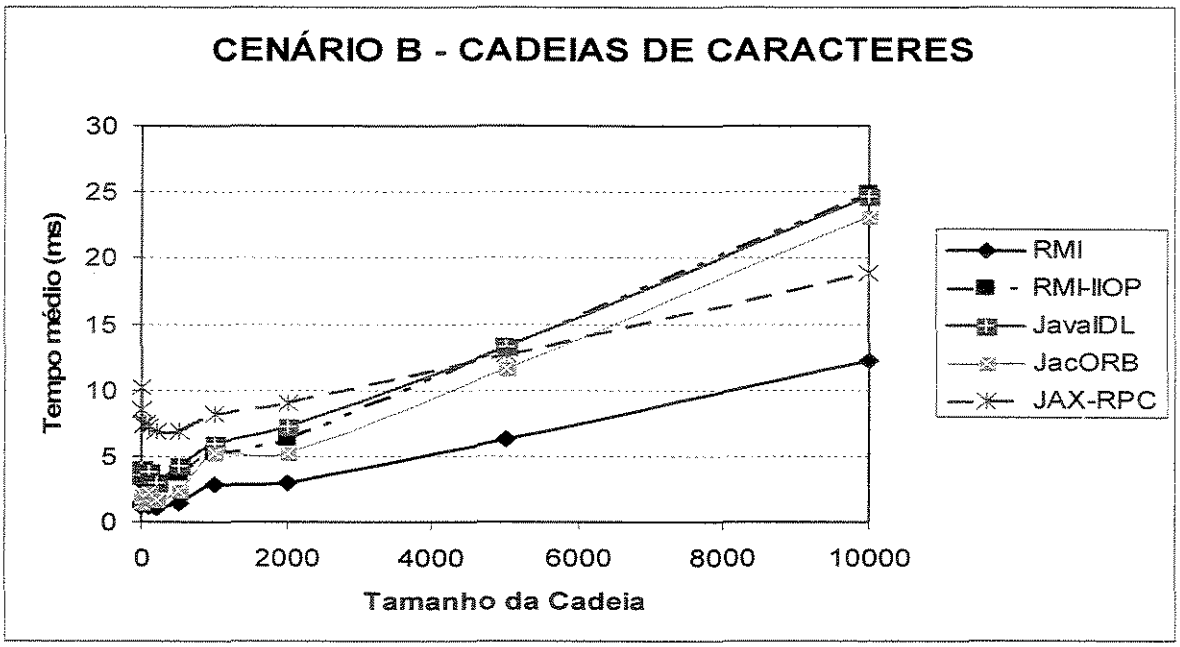

Gráfico 27 **Cenário B - cadeias de até 10.000 caracteres**

Para cadeias de 10.000 caracteres ou maiores, RMI manteve-se com o melhor desempenho, seguido de JAX-RPC. O desempenho das tecnologias baseadas em IIOP (RMI-IIOP, JavaIDL e JacORB) ficou em um patamar acima das demais, com JacORB superando JavaIDL e RMI-IIOP:

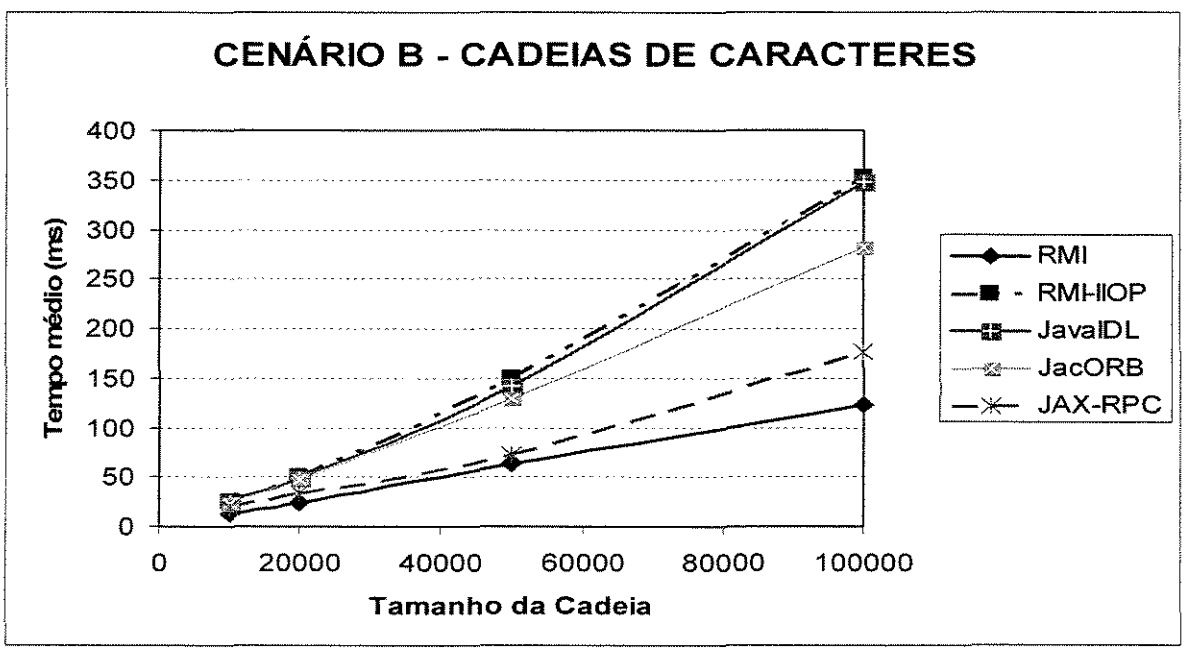

Gráfico 28 **Cenário B - cadeias de 10.000 a 100.000 caracteres**

Os desvios padrão obtidos para cadeias de até 10.000 caracteres foram relativamente pequenos (menores do que 4,95, com exceção de RMI para 10.000 caracteres: 6,47). A partir de 10.000 caracteres, os desvios observados foram maiores (entre 5,65 e 7,68 para 20.000 caracteres, entre 11,98 e 43,45 para 50.000 e entre 18,65 e 62,46 para 100.000 caracteres). Na grande maioria dos pontos, RMI apresentou o menor desvio padrão (exceção para 10.000 onde RMI obteve o maior desvio e para 50.000 onde RMI obteve o segundo menor desvio).

No cenário B, nas medidas dos métodos devolvendo listas de objetos, JAX-RPC teve um desempenho muito pior do que as demais tecnologias, como pode ser observado no gráfico seguinte:

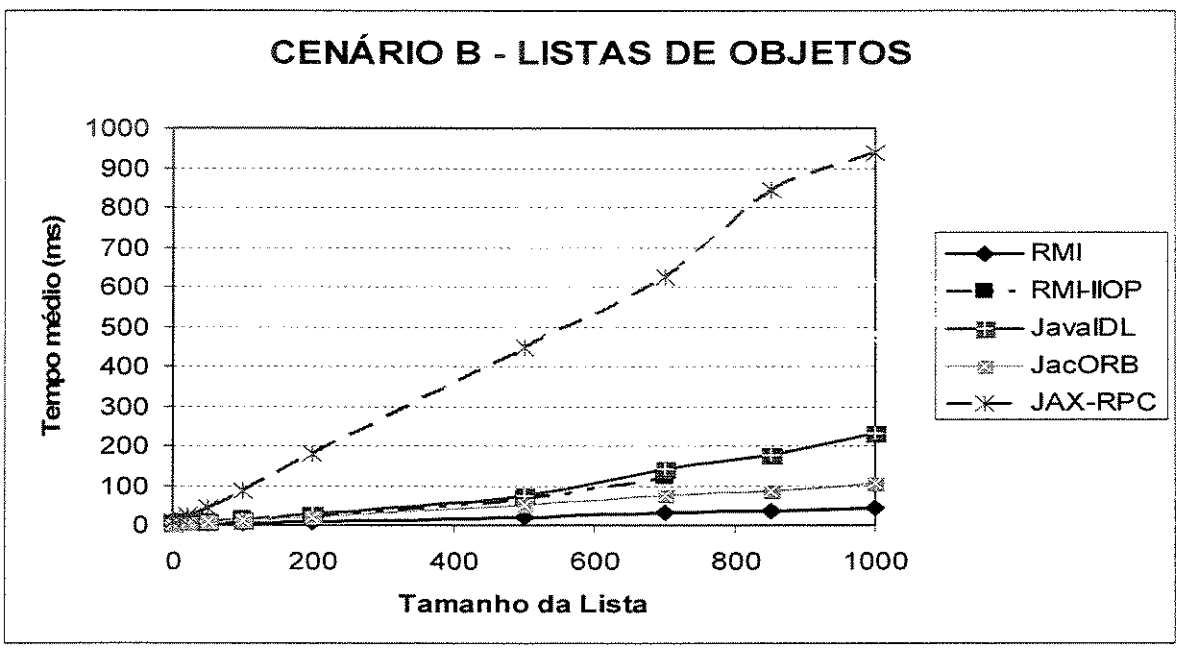

Gráfico 29 Cenário B – listas de até 1.000 objetos

De forma a facilitar a visualização do desempenho das demais tecnologias, a curva de JAX-RPC foi removida dos próximos gráficos.

Para listas de até 200 objetos, RMI obteve o melhor desempenho. JacORB teve o segundo melhor desempenho. JavaIDL e RMI-IIOP novamente tiveram desempenho similar, em um patamar acima dos demais. O desvio padrão observado para listas de até 200 objetos foi menor do que 4,5 para na grande maioria dos casos (exceção para JAX-RPC – listas de 100 e 200 objetos, respectivamente 10,25 e 46,68). RMI apresentou sempre um dos menores desvios (superado apenas por RMI-IIOP em alguns pontos).

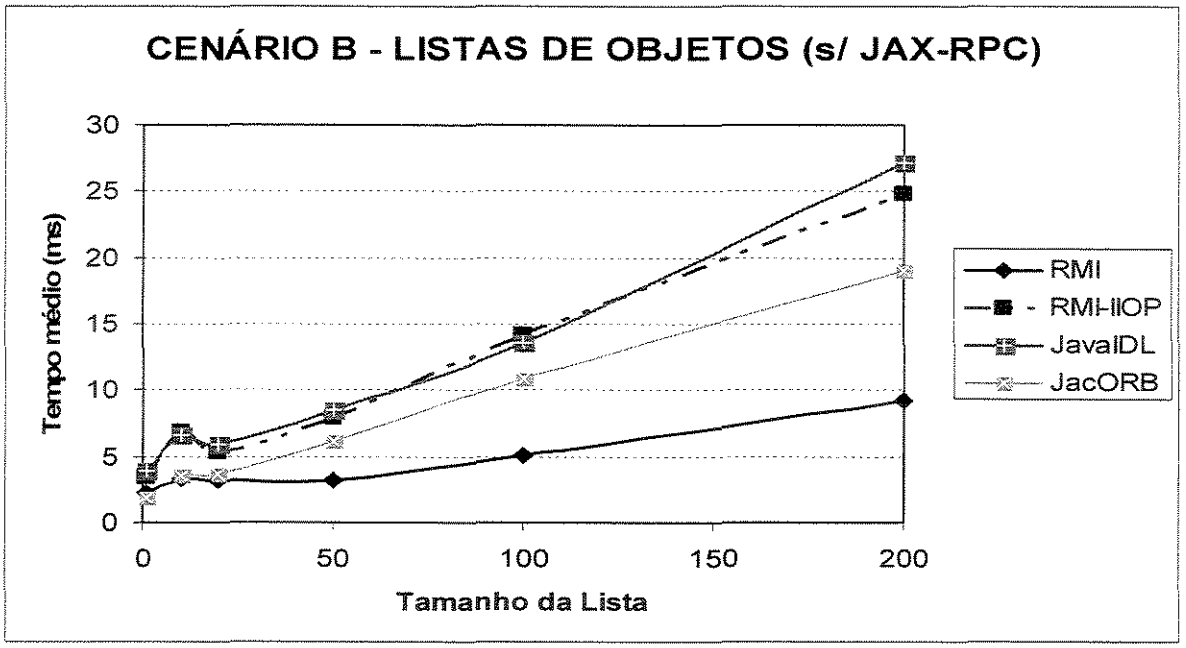

Gráfico 30 **Cenário B – listas de até 200 objetos sem JAX-RPC**

Para listas de 200 a 1.000 objetos, o desempenho das tecnologias estudadas seguiu o mesmo padrão observado no gráfico anterior. Novamente é importante notar que ocorreu um erro de execução com RMI-IIOP para listas de 850 e 1.000 objetos:

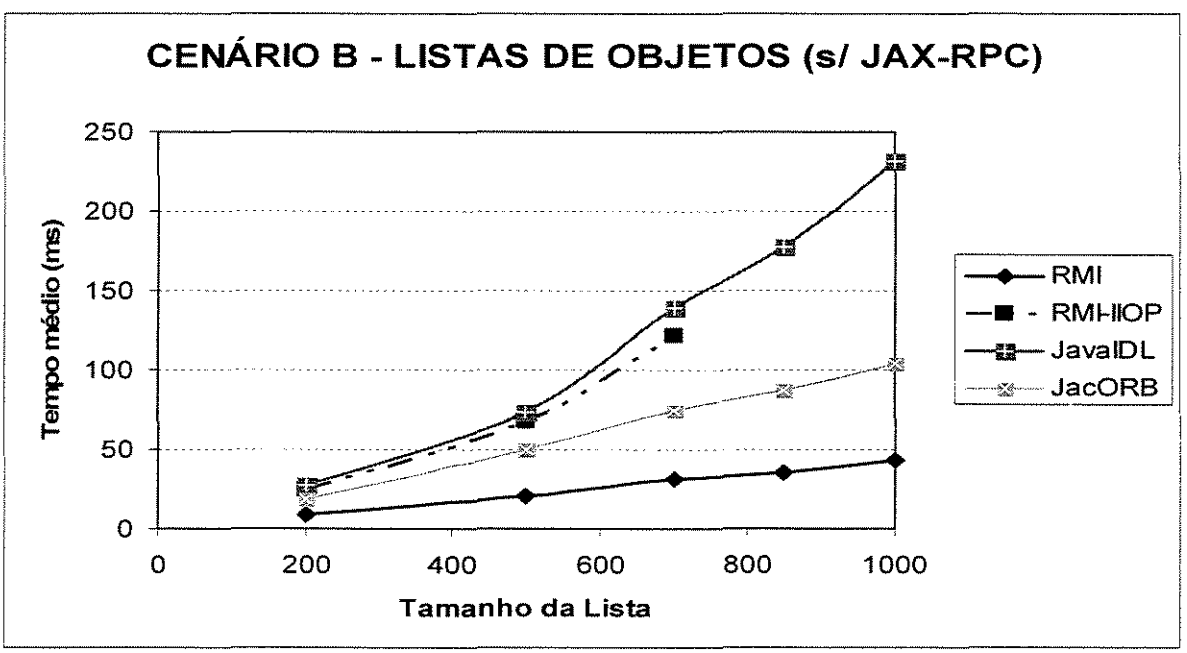

Gráfico 31 **Cenário B – listas de 200 a 1.000 objetos sem JAX-RPC**

RMI novamente obteve os menores desvios padrão, abaixo de 8,64 para listas de 200

a 1.000 objetos. JacORB obteve desvios abaixo de 17,70, enquanto que nas demais tecnologias obtiveram os desvios maiores ficaram entre 58,34 e 82,73).

Em resumo, da mesma forma que no cenário A, no cenário B RMI apresentou um desempenho melhor do que as demais tecnologias na grande maioria dos pontos. JacORB foi a tecnologia que conseguiu o desempenho mais próximo a RMI, com exceção dos métodos que devolvem cadeias de 10.000 caracteres ou maiores, onde JAX-RPC obteve um resultado melhor. JavaIDL e RMI-IIOP tiveram um desempenho similar, ficando acima de JacORB na maioria dos casos. JAX-RPC teve o pior desempenho na maioria dos pontos, superando apenas JavaIDL e RMI-IIOP para cadeias de 10.000 caracteres ou maiores.

4.2.3 Cenário C

O cenário C permite estudar o comportamento das tecnologias com a comunicação entre cliente e servidor através da Internet, um ambiente heterogêneo, sujeito a muita interferência de tráfego.

Para cadeias de até 1.000 caracteres, RMI obteve um desempenho melhor do que as demais tecnologias. As tecnologias baseadas em IIOP (RMI-IIOP, JavaIDL e JacORB) ficaram em um segundo patamar com um desempenho próximo entre si, com exceção de um ponto (200 caracteres), onde RMI-IIOP ficou bem próximo a RMI. É possível observar também no gráfico a seguir que JAX-RPC ficou em um terceiro patamar, acima das demais tecnologias, se aproximando das tecnologias baseadas em RMI-IIOP para cadeias de 1.000 caracteres.

Os desvios padrão observados para cadeias de até 10.000 caracteres no cenário C ficaram abaixo de 6,50 com exceção dos seguintes casos: RMI – 500 caracteres (11,65), RMI-IIOP – 200 caracteres (11,47), JavaIDL - 500 e 1.000 caracteres (respectivamente 8,31 e 26,53), JAX-RPC – 1 caractere (30,89) e JacCORB – todos os casos (entre 20,35 e 30,11). Na maioria dos casos JacORB apresentou o maior desvio padrão. Os menores desvios padrão foram obtidos por RMI, RMI-IIOP e JavaIDL, alternadamente.

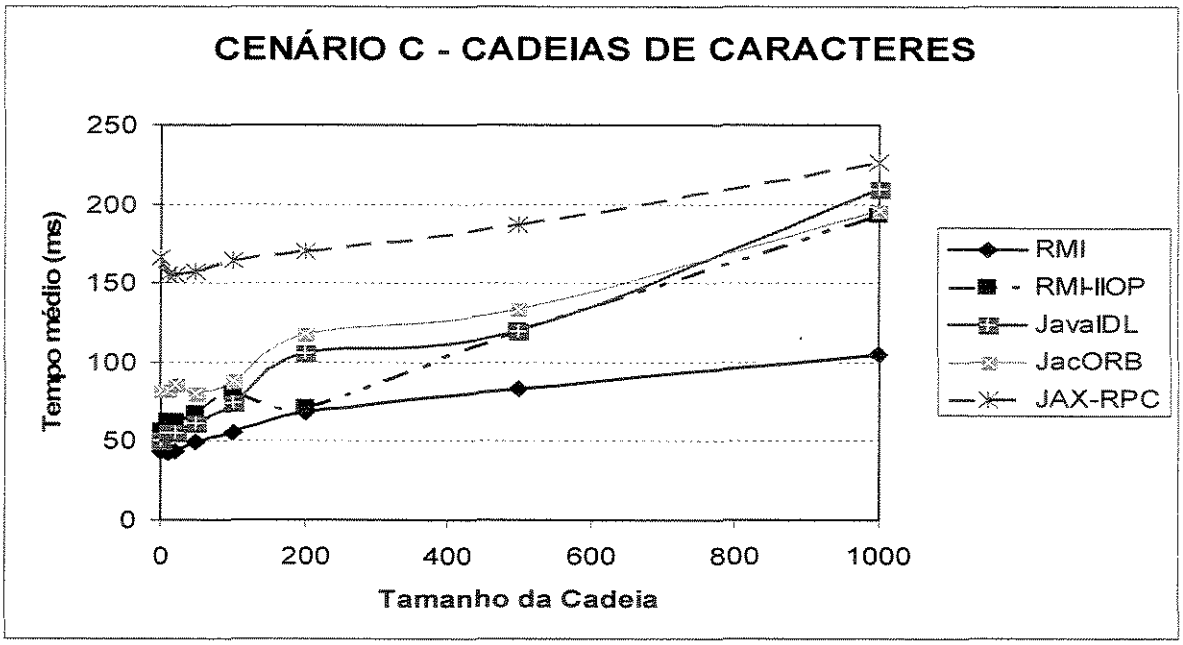

Gráfico 32 **Cenário C - cadeias de até 1.000 caracteres**

Para cadeias a partir de 2.000 caracteres, RMI e JAX-RPC tiveram desempenho similar, o mesmo acontecendo para as tecnologias baseadas em IIOP que ficaram em um patamar acima de RMI e JAX-RPC:

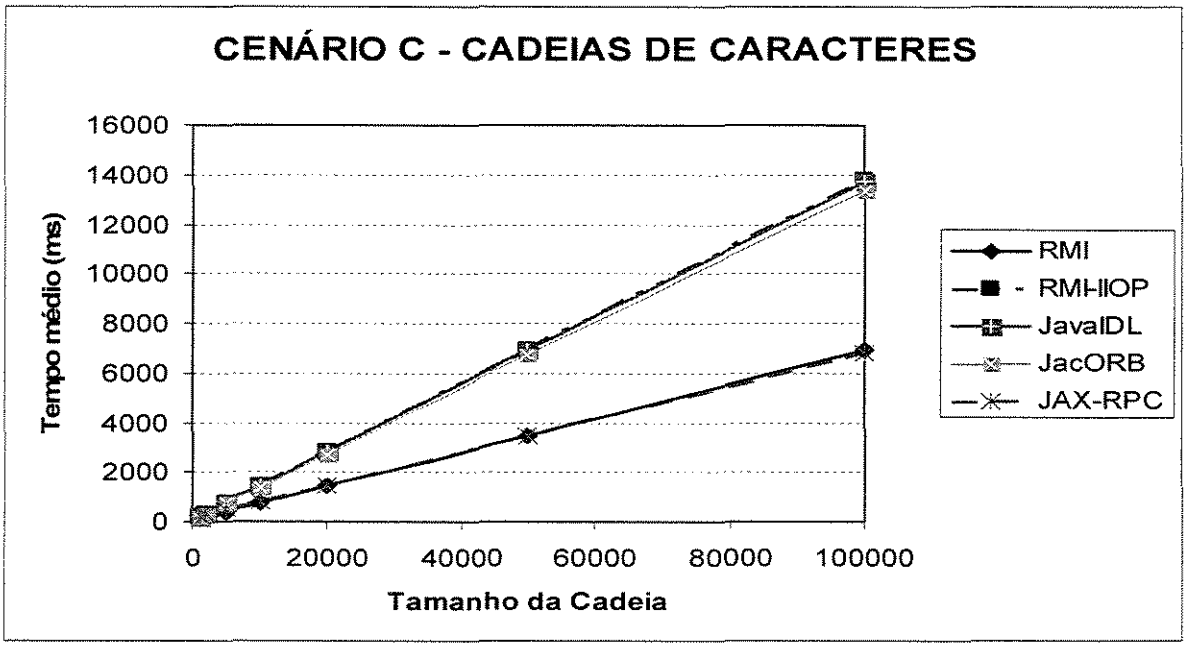

Gráfico 33 **Cenário C - cadeias de 1.000 a 100.000 caracteres**

Para cadeias a partir de 2.000 caracteres, os desvios padrão também variaram de

forma bem inconsistente. Uma boa parte dos desvios ficou abaixo de 10, principalmente para RMI e JAX-RPC. JacORB apresentou desvios maiores do que 20 em todos os pontos. RMI-IIOP e JavaIDL apresentaram desvios maiores do que 20 a partir de 10.000 caracteres.

Nos próximos gráficos é possível observar que RMI obteve o melhor desempenho para listas de objetos. JacORB obteve o segundo melhor desempenho, com tempos médios maiores do que RMI. Os desvios padrão de JacORB foram os menores observados na grande maioria dos pontos (exceto listas de 1 e 20 objetos).

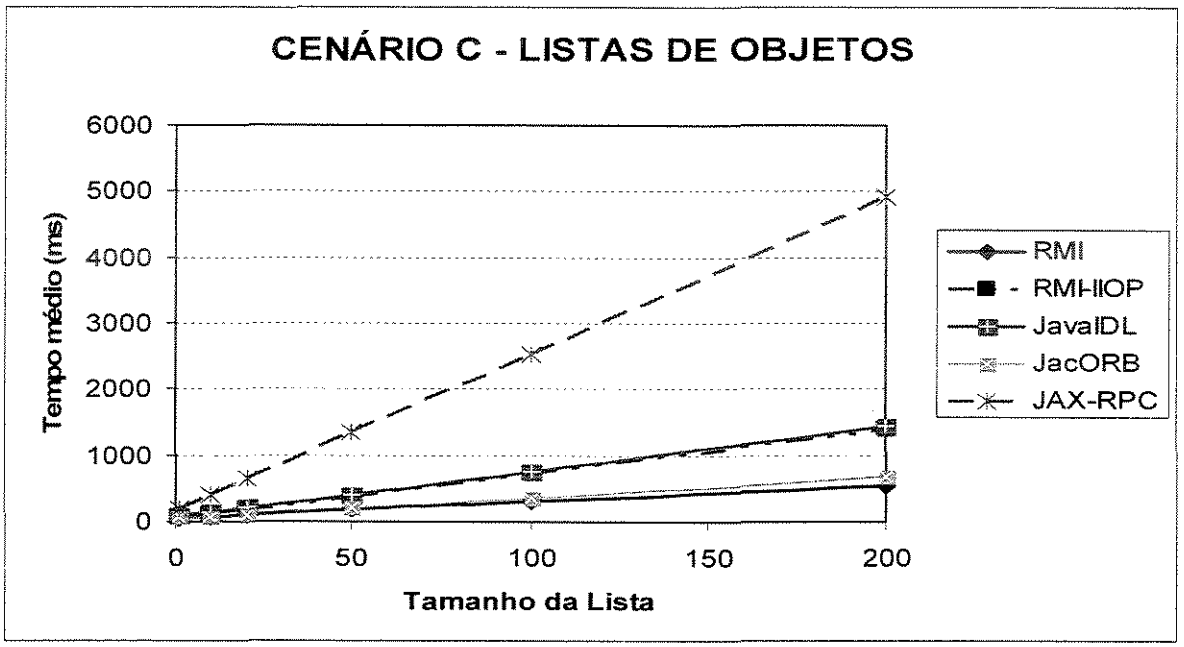

Gráfico 34 Cenário C – listas de até 200 objetos

JavaIDL e RMI-IIOP tiveram um desempenho similar, em um patamar acima de RMI e JacORB. RMI-IIOP novamente apresentou um erro de execução para listas de 850 e 1.000 objetos. JAX-RPC obteve o pior desempenho para listas de objetos, com uma curva de crescimento muito acentuada, diferente das curvas obtidas para as demais tecnologias. JavaIDL apresentou maiores desvios padrão do que as demais tecnologias para listas de até 100 objetos. A partir de 200 objetos, os maiores desvios padrão foram observados para JAX-RPC.

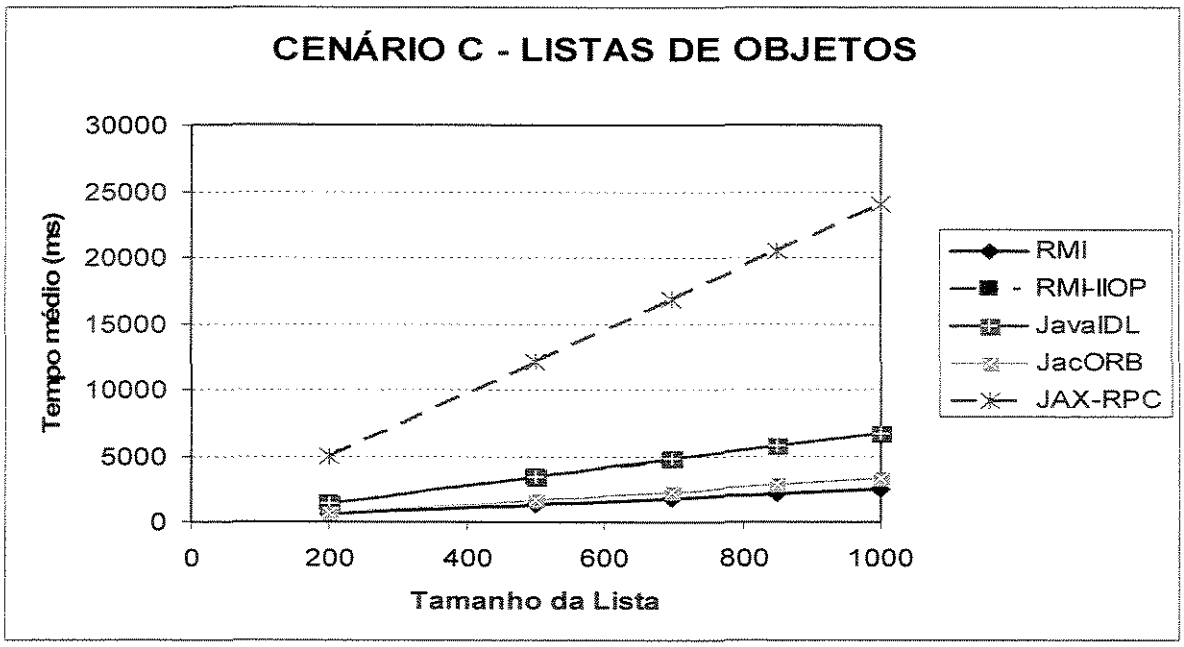

Gráfico 35 Cenário C – listas de 200 a 1.000 objetos

RMI apresentou um desempenho melhor do que as demais tecnologias na grande maioria dos pontos no cenário C. Para cadeias de caracteres as tecnologias baseadas em IIOP tiveram um desempenho similar, em um patamar acima de RMI. Para cadeias de até 1.000 caracteres JAX-RPC ficou em um patamar acima das tecnologias baseadas em IIOP. Entretanto, para cadeias maiores o desempenho de JAX-RPC se aproximou ao de RMI, superando as demais. Para listas de objetos JacORB teve um desempenho próximo ao de RMI e JavaIDL e RMI-IIOP ficaram em um patamar acima de JacORB. JAX-RPC novamente obteve o pior desempenho para métodos devolvendo listas de objetos, em um patamar muito acima dos demais.

4.3 Conclusões

Com os resultados apresentados na seção 4.1 foi possível concluir que a comunicação entre as máquinas do cliente e do servidor introduziu um custo adicional nos seguintes casos:

- RMI: cadeias com mais de 1.000 de caracteres;
- RMI-IIOP: para cadeias com mais de 500 caracteres e para listas de objetos de uma forma geral;
- JavaIDL: para cadeias com mais de 200 caracteres;

- JacORB: na grande maioria das chamadas de métodos testadas;
- JAX-RPC: para cadeias com mais de 500 caracteres e para listas contendo 10 objetos ou mais.

Em alguns casos o desempenho no cenário B foi melhor do que o desempenho no cenário A, ou seja, com as aplicações cliente e servidor rodando em máquinas distintas e comunicando-se através de uma rede local o desempenho do sistema foi melhor.

Com as comparações entre as diversas tecnologias nos cenários A, B e C foi possível concluir o seguinte:

- RMI é a tecnologia que conseguiu o melhor desempenho em todos os cenários, na grande maioria das chamadas de métodos exercitadas;
- as tecnologias baseadas em IIOP que fazem parte do Java SDK – RMI-IIOP e JavaIDL – obtiveram desempenho similar na maioria dos casos;
- dentre as tecnologias baseadas em IIOP, JacORB foi a que demonstrou melhor desempenho na maioria dos casos;
- JacORB foi a tecnologia que obteve o melhor desempenho depois de RMI na maioria dos casos;
- JAX-RPC obteve um desempenho melhor do que as tecnologias baseadas em IIOP (JavaIDL, RMI-IIOP e JacORB) apenas para cadeias de caracteres grandes (maiores ou iguais a 50.000 caracteres no cenário A, maiores ou iguais a 10.000 caracteres no cenário B e maiores ou iguais a 2.000 caracteres no cenário C);
- JAX-RPC obteve consistentemente o pior desempenho nos métodos devolvendo listas de objetos.

Além disso, a análise dos desvios padrão indicou variações maiores nos métodos devolvendo cadeias de caracteres e listas de objetos grandes. Nos cenários A e B os desvios foram maiores para cadeias com 50.000 caracteres ou mais e listas com 700 objetos ou mais. No cenário C o mesmo aconteceu para cadeias com mais de 5.000 caracteres e listas com mais de 200 objetos.

Quanto às diferenças de desvio padrão entre as tecnologias, é possível destacar o seguinte:

- JAX-RPC apresentou desvios maiores para listas com mais de 100 objetos em todos os cenários;

- JacORB apresentou desvios maiores nos métodos devolvendo cadeias de caracteres no cenário C.
- JavaIDL apresentou desvios maiores nos métodos retornando listas de até 100 objetos no cenário C,.
- nos demais casos, os desvios padrão apresentados pelas diferentes tecnologias foram similares, variando num intervalo pequeno, quando comparados métodos devolvendo os mesmos valores num determinado cenário.

Considerações Finais

Este trabalho apresentou diversas tecnologias de comunicação entre objetos distribuídos em Java: RMI, RMI-IIOP, CORBA (JavaIDL e JacORB) e JAX-RPC (SOAP). Suas arquiteturas e suas principais características foram descritas, assim como exemplos contendo os passos principais para a sua utilização.

Em seguida foram apresentadas comparações de desempenho dessas tecnologias em três cenários distintos: cliente e servidor na mesma máquina, cliente e servidor em máquinas distintas de uma mesma sub-rede e cliente e servidor se comunicando diretamente através da Internet. Para a comparação foram utilizadas chamadas de procedimentos remotos devolvendo cadeias de caracteres e listas de objetos de diversos tamanhos.

De uma forma geral, a tecnologia RMI apresentou o melhor desempenho, confirmando as expectativas. Entretanto as demais tecnologias são mais flexíveis e oferecem maior portabilidade.

Outra conclusão importante é que a tecnologia JAX-RPC, utilizando o padrão de mercado atual para *Web Services* (SOAP), é a menos eficiente entre as tecnologias comparadas.

Como trabalho futuro pode ser incluído mais um importante cenário: a comunicação entre cliente e servidor separados por *firewalls* na Internet.

Uma nova tecnologia está sendo desenvolvida pela Sun com a expectativa de melhorar a performance de JAX-RPC através da utilização de codificações binárias teoricamente mais eficientes como alternativas para representação das chamadas em XML (formato texto). Essa tecnologia é denominada "*Fast Web Services*" [51] e um outro trabalho futuro poderia ser a comparação de seu desempenho com as tecnologias consideradas neste trabalho.

Referências Bibliográficas

[1] Distributed Systems: Concepts and Design – G. Coulouris, J. Dollimore and T. Kindberg – Addison-Wesley, 2nd Edition, 1994

[2] Distributed Operating Systems – Andrew S. Tanenbaum – Printice Hall, 1994 – ISBN 0-13-219908-4

[3] A Note on Distributed Computing. - Samuel C. Kendall, Jim Waldo, Ann Wollrath and Geoff Wyant – Sun Microsystems Laboratories, Inc. - November 1994 – http://research.sun.com/research/techrep/1994/smli_tr-94-29.pdf

[4] Java Remote Method Invocation (RMI) – http://java.sun.com/products/jdk/rmi/

[5] Java – http://java.sun.com/

[6] Object Serialization – http://java.sun.com/j2se/1.4/docs/guide/serialization/

[7] Dynamic code downloading using RMI (Using the java.rmi.server.codebase Property) – http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html

[8] Getting Started Using RMI – Tutorial – http://java.sun.com/j2se/1.4.2/docs/guide/rmi/getstart.doc.html

[9] Java RMI Tutorial – http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html

[10] Fundamentals of RMI - Short Course – http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html

[11] Remote Object Activation – Tutorials – http://java.sun.com/j2se/1.4.2/docs/guide/rmi/activation.html

[12] OMG – *Object Management Group* – http://www.omg.org

[13] CORBA: Integrating diverse applications within distributed heterogeneous environments – Steve Vinoski – IEEE Communications Magazine, Volume 35, Issue 2 (Feb. 1997).

[14] New Features for CORBA 3.0 – Steve Vinoski – Communications of the ACM, Volume 41, Issue 10 (Oct. 1998)

[15] Distributed Programming with Java, Chapter 11: Overview of CORBA – Qusay H. Mahmoud – January 2001 – http://developer.java.sun.com/developer/Books/corba/

[16] Introduction to CORBA – http://java.sun.com/developer/onlineTraining/corba/corba.html

[17] IONA Products – Orbix – http://www.iona.com/products/orbix.htm

[18] Borland Enterprise Server, VisiBroker Edition – http://www.borland.com/besvisibroker

[19] Java IDL – http://java.sun.com/products/jdk/idl

[20] JacORB – http://www.jacorb.org

[21] History of CORBA – http://www.omg.org/gettingstarted/history_of_corba.htm

[22] CORBA Programming with J2SE 1.4 – Programming Transient and Persistent Servers – http://java.sun.com/developer/technicalArticles/releases/corba/

[23] OMG Security – http://www.omg.org/technology/documents/formal/omg_security.htm

[24] Getting Started with Java IDL – http://java.sun.com/j2se/1.4.2/docs/guide/idl/GShome.html

[25] Java IDL Technology Documentation – http://java.sun.com/j2se/1.4.2/docs/guide/idl/index.html

[26] Java RMI over IIOP Documentation – http://java.sun.com/products/rmi-iiop/index.html

[27] Java to IDL Language Mapping Specification – September 2003 – Version 1.3 – formal/03-09-04 – OMG – http://www.omg.org/docs/formal/03-09-04.pdf

[28] RMI over IIOP – Java World – Akira Andoh and Simon Nash – December 1999 – http://www.javaworld.com/javaworld/jw-12-1999/jw-12-iiop_p.html

[29] Java IDL and RMI-IIOP: Naming Services – http://java.sun.com/j2se/1.4.2/docs/guide/idl/jidlNaming.html

[30] RMI-IIOP Programmer's Guide – http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/rmi_iiop_pg.html

[31] Tutorial: Getting Started Using RMI-IIOP – http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/tutorial.html

[32] Java API for XML-based RPC 1.1 Specification – http://java.sun.com/xml/downloads/jaxrpc.html

[33] XML – eXtensible Markup Language – http://www.w3.org/TR/xml11

[34] SOAP –Simple Object Access Protocol – http://www.w3.org/TR/SOAP/

[35] JCP – Java Community Process – http://jcp.org/

[36] JSR-101 – Java APIs for XML based RPC – http://jcp.org/jsr/detail/101.jsp

[37] WSDL – Web Services Description Language - http://www.w3.org/TR/wsdl

[38] W3C – World Wide Web Consortium – http://www.w3.org

[39] Java API for XML Registries (JAXR) – http://java.sun.com/xml/jaxr/index.jsp

[40] Java API for XML-Based RPC (JAX-RPC) Overview – http://java.sun.com/xml/jaxrpc/overview.html

[41] Java API for XML-based RPC (JAX-RPC): A Primer – http://java.sun.com/xml/jaxrpc/primerarticle.html

[42] J2SE – Java 2 Standard Edition – http://java.sun.com/j2se/index.jsp

[43] Articles: Servlets – http://java.sun.com/developer/technicalArticles/Servlets

[44] J2EE – Java 2 Enterprise Edition – http://java.sun.com/j2ee/index.jsp

[45] SOAP Messages with Attachments – http://www.w3.org/TR/SOAP-attachments

[46] JAX-RPC Brings Portability to Web Services – http://java.sun.com/features/2003/01/jax_rpc.html

[47] Java Web Services Developer Pack (Java WSDP) – http://java.sun.com/webservices/jwsdp/index.jsp

[48] The Java Web Services Tutorial for JWSDP v1.3 – http://java.sun.com/webservices/docs/1.3/tutorial/doc/index.html

[49] Comparison of CORBA and Java RMI Based on Performance Analysis – http://citeseer.nj.nec.com/juric98comparison.html

[50] Estudo comparativo detalhado do desempenho de Java/CORBA, Java/RMI, Java/ICE e Java/SOAP – http://gsd.ime.usp.br/~kon/MAC5755/projetos/Emilio/RelatorioFinal/

[51] Fast Web Services – http://java.sun.com/developer/technicalArticles/WebServices/fastWS/index.html

[52] Understanding SOAP – K. Scribner, M. Stiver – SAMS Publishing – ISBN 0-672-31922-5

[53] Notas de Aula - MAC 5759 - Sistemas de Objetos Distribuídos – Fábio Kon – http://www.ime.usp.br/~kon/MAC5759/aulas/Aula25.html

[54] Universal Description, Discovery, and Integration (UDDI) project – http://www.uddi.org/

[55] ebXML – http://www.ebxml.org/

Observação: todos os endereços HTTP referenciados nesta seção foram acessados pela última vez em 27 de Março de 2004.