

**Pedro Otávio Alves Neto**

*Universidade para o Desenvolvimento  
do Estado e da Região do Pantanal*

robsonsoares.silva@gmail.com

**Robson Soares Silva**

*Universidade para o Desenvolvimento  
do Estado e da Região do Pantanal*

robsonsoares.silva@gmail.com

**Anhanguera Educacional S.A.**

Correspondência/Contato  
Alameda Maria Tereza, 2000  
Valinhos, São Paulo  
CEP 13.278-181  
rc.ipade@unianhanguera.edu.br

Coordenação  
Instituto de Pesquisas Aplicadas e  
Desenvolvimento Educacional - IPADE

Artigo Original  
Recebido em: 8/10/2008  
Avaliado em: 13/2/2009

Publicação: 13 de março de 2009

## **TECNOLOGIAS DE SISTEMAS DISTRIBUÍDOS IMPLEMENTADAS EM JAVA: SOCKETS, RMI, RMI-IIOP E CORBA**

---

### **RESUMO**

Este trabalho descreve as principais características no desenvolvimento de Sistemas Distribuídos usando a linguagem Java, seus conceitos, desafios e implementações. Os Sistemas Distribuídos proporcionam diferentes tecnologias: SOCKETS, RMI (*Remote Method Invocation*), RMI-IIOP (*Internet Inter-Orb Protocol*) e CORBA (*Common Object Request Brokerage Architecture*). O objetivo do trabalho corresponde ao desenvolvimento de uma mesma aplicação em cada uma das tecnologias para poder conseguir vivenciar as diferentes necessidades de cada uma delas. Suas características e desenvolvimentos são distintos, mas o modelo de comunicação é o mesmo, no qual, esse tipo de comunicação é chamada de Cliente/Servidor. O desenvolvimento das aplicações tem como base um projeto lógico através de um estudo de caso utilizando diagrama de caso de uso, com objetivo de obter o acesso um banco de dados. Por fim apresenta-se uma tabela comparativa referente às principais características de cada aplicação desenvolvida em cada uma das tecnologias.

**Palavras-Chave:** Sistemas Distribuídos; Java; comparação; tecnologias; banco de dados.

---

### **ABSTRACT**

This work describes the main features in the development of Distributed Systems using the Java language, its concepts, challenges and implementations. The Distributed Systems provide different technologies: SOCKETS, RMI, RMI-IIOP and CORBA. The objective was to develop a single application in each one of the technologies to be able to obtain to live deeply the different necessities of each one of them. Its characteristics and developments are different but the communication model is the same, in which, this type of communication is called Client/Server. The development of the applications a logical project through a case study had had as base using diagram of use case, with objective to get the access a data base. Finally, a referring comparative table to the main features of each application developed in each one of the technologies is presented.

**Keywords:** Distributed systems; Java; comparison; technologies; data base.

## 1. INTRODUÇÃO

Sistemas distribuídos têm se desenvolvido muito nos últimos anos, e sua utilização vem apresentando um crescimento constante. Os principais fatores que colaboram para esse crescimento são a evolução da capacidade de processamento dos computadores e as crescentes taxas de transmissão das redes de comunicação, mas a motivação maior para contribuir e usar sistemas distribuídos é proveniente do desejo de compartilhar recursos.

O trabalho descreve as principais tecnologias de programação distribuída disponíveis na linguagem de programação Java. A linguagem Java foi escolhida, pois é uma linguagem de programação orientada a objeto que suporta interação entre objetos remotos e vem apresentando um grande crescimento em utilização e importância nos últimos anos, principalmente para as aplicações desenvolvidas para sistemas distribuídos.

O objetivo geral deste trabalho é apresentar as principais tecnologias de programação para sistemas distribuídos disponíveis em Java e realizar uma tabela comparativa entre as mesmas.

As tecnologias apresentadas por este trabalho são: Sockets, RMI, RMI-IIOP e CORBA. Existem várias diferenças entre essas tecnologias, onde as diferenças são fatores cruciais em relação ao desempenho e na escolha de qual das mesmas utilizar quando for efetuar a implementação de um sistema distribuído.

## 2. SISTEMAS DISTRIBUÍDOS

Sistemas Distribuídos consistem em uma coleção de computadores independentes que se apresenta ao usuário como um sistema único coerente. Esta definição tem dois aspectos. Primeiro trata do *hardware*: as máquinas são autônomas. Segundo trata do *software*: os usuários pensam do sistema como um único computador. Ambos são essenciais. (TANENBAUM, 1995, p.2).

Pode-se dizer também que um Sistema Distribuído é um sistema onde os componentes de *hardware* e *software* estão localizados em computadores interligados por uma rede, comunicam e coordenam suas ações somente através de troca de mensagens. (COULOURIS, 2001, p.1)

A motivação para construir e usar sistemas distribuídos é derivada da necessidade de compartilhar recursos. O termo “recurso” é bastante abstrato, mas caracteriza bem o conjunto de coisas que podem ser compartilhadas de maneira útil em um sistema de computadores interligados em rede.

O conceito “recurso” abrange desde componentes de *hardware*, como discos e impressoras, até entidades definidas pelo *software*, como arquivos, bancos de dados e objetos de dados de todos os tipos, isso inclui o fluxo de quadros de vídeo conferência de uma câmera de vídeo digital ou a conexão de áudio que uma chamada de telefone móvel representa. (COULOURIS, 2001, p.02).

Para Colouris (2001, p. 25), a construção de sistemas distribuídos gera muitos desafios:

**Heterogeneidade:** eles devem ser construídos a partir de uma variedade de redes, sistemas operacionais, *hardware* e linguagens de programação diferentes. Os protocolos de comunicação da Internet mascaram a diferença existente nas redes e o *middleware* pode cuidar das outras diferenças.

**Sistemas abertos:** os sistemas distribuídos devem ser extensíveis, no qual, o primeiro passo é publicar as interfaces dos componentes, mas a integração de componentes escritos por diferentes programadores é um desafio real.

**Segurança:** a criptografia pode ser usada para proporcionar proteção adequada para os recursos compartilhados e para manter informações sigilosas em segredo, quando são transmitidas em mensagens por uma rede. Os ataques de negação de serviço ainda são um problema.

**Escalabilidade:** um sistema distribuído é considerado escalável, isto é, se permanece eficiente quando há um aumento significativo no número de recursos e no número de usuários.

**Tratamento de falhas:** qualquer processo, computador ou rede pode falhar, independentemente dos outros. As falhas em um sistema distribuído são parciais, isto é, alguns componentes falham, enquanto outros continuam funcionando. Portanto, o tratamento de falhas é particularmente difícil, mas existem técnicas para tratar essas falhas.

**Concorrência:** a presença de múltiplos usuários em um sistema distribuído é uma fonte de pedidos concorrentes para seus recursos, ou seja, possibilita que vários usuários acessem um recurso compartilhado ao mesmo tempo.

**Transparência:** o objetivo é tornar certos aspectos da distribuição invisíveis para o programador de aplicativos, para que este se preocupe apenas com o projeto de seu aplicativo em particular. Por exemplo, ele não precisa estar preocupado com sua localização ou com os detalhes sobre como suas operações serão acessadas por outros componentes, nem se será replicado ou migrado. As falhas de rede e de processos podem ser apresentadas aos programadores de aplicativos na forma de exceções - mas elas devem ser tratadas. (COULOURIS, 2001, p.26).

### 3. TECNOLOGIAS PARA DESENVOLVIMENTO DE SISTEMAS DISTRIBUÍDOS EM JAVA

A linguagem Java possibilita o desenvolvimento de aplicações distribuídas de diferentes formas. As formas possíveis em Java são: Sockets, RMI, RMI-IIOP e CORBA.

#### 3.1. Sockets

O modelo mais utilizado para aplicações distribuídas em redes de computadores é chamado cliente/servidor, no qual, a comunicação costuma se dar através de uma mensagem de solicitação do cliente enviada para o Servidor, pedindo para que alguma tarefa seja executada. Em seguida, o servidor executa a tarefa e envia a resposta. O mecanismo mais utilizado atualmente para esse modelo que possibilita a comunicação entre aplicações é chamado de *Socket*.

Através de um *socket* é possível realizar diversas operações, como exemplo:

- Estabelecer conexões entre máquinas.
- Enviar e receber dados.
- Encerrar conexões.
- Esperar por conexões em determinada porta.

##### 3.1.1. Modos de utilização de Sockets

Os seguintes modos de utilização de *sockets* que Java oferece é o modo orientado à conexão, que funciona sobre o protocolo TCP (*Transmission Control Protocol*, ou protocolo de controle de transmissão), e o modo orientado a datagrama, que funciona sobre o protocolo UDP (*User Datagram Protocol*, ou protocolo de datagrama de usuários). Os dois modos funcionam sobre o protocolo IP (*Internet Protocol*).

### Modo orientado à conexão TCP/IP

O processo de comunicação no modo orientado à conexão ocorre quando o servidor escolhe uma determinada porta e fica aguardando conexões nesta porta. O cliente deve saber previamente qual a máquina servidora (*host*) e a porta que o servidor está aguardando conexões. Então o cliente solicita uma conexão em um *host/porta*, de acordo com a Figura 1.

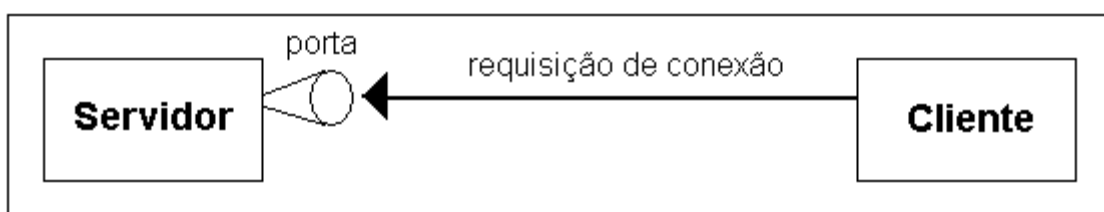


Figura 1. Requisição de conexão.

Se tudo ocorrer conforme o esperado, ou seja, se não existir nenhum problema, o servidor aceita a conexão gerando um *socket* em uma porta qualquer do lado servidor, criando assim um canal de comunicação entre o cliente e o servidor, como demonstrado na Figura 2.

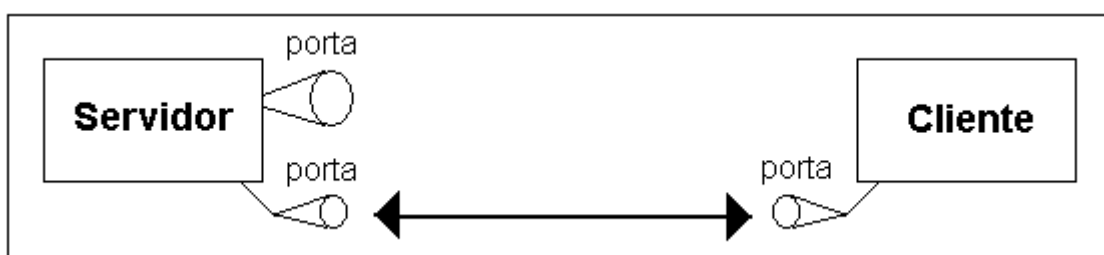


Figura 2. Comunicação entre Servidor e Cliente.

Tipicamente o comportamento do servidor é ficar em um *loop* aguardando novas conexões e gerando *sockets* para atender as solicitações de clientes.

### Modo orientado a datagrama UDP/IP

O UDP é um acrônimo do termo inglês *User Datagram Protocol* que significa protocolo de datagramas de utilizador (ou usuário) é um protocolo de transporte que presta um serviço de comunicação não orientado à conexão e sem garantia de entrega. As aplicações que usam o UDP devem tomar medidas para garantir a recuperação dos dados perdidos, a organização dos dados recebidos fora de ordem, a eliminação dos dados recebidos em duplicidade e o controle de fluxo. (ALBUQUERQUE, 2001, p.11).

*Sockets UDP/IP* são muito mais rápidos e mais simples que *sockets TCP/IP*, porém menos confiáveis. Em UDP não existe o estabelecimento de conexão, sendo que a comunicação ocorre apenas com o envio de mensagens.

Uma mensagem é um datagrama, pelo qual é composto de um remetente (*sender*), um destinatário ou receptor (*receiver*), e a mensagem (*content*). Em UDP, caso o destinatário não esteja aguardando uma mensagem, ela é perdida.

A Figura 3 apresenta o envio de um datagrama de uma suposta Máquina1 para outra Máquina2 em uma rede.

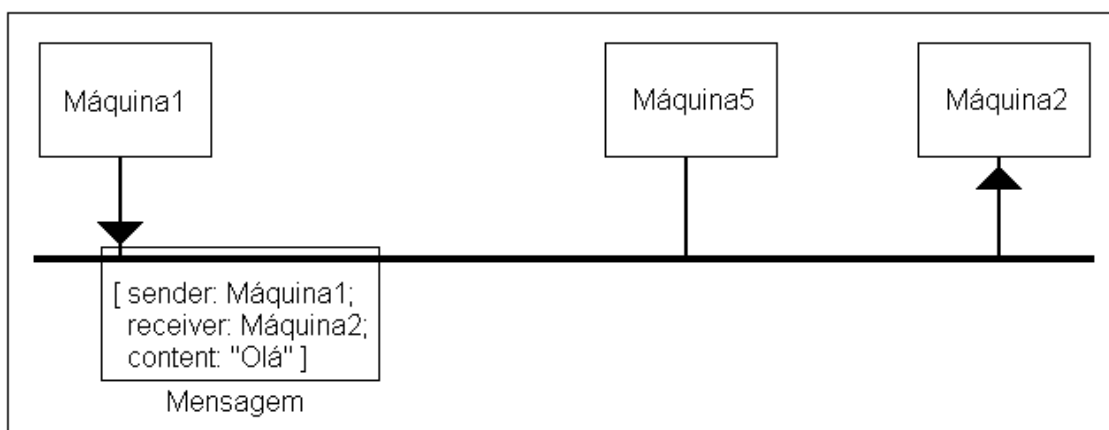


Figura 3. Envio de um Datagrama.

### 3.2. RMI

O mecanismo RMI (*Remote Method Invocation*) é uma tecnologia de programação de objetos distribuídos que faz parte 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. (RMI, 2009).

Em outras palavras, essa é uma aplicação cliente/servidor na qual o lado cliente submete uma tarefa a ser executada pelo objeto remoto localizado no lado servidor. Tal objeto, por sua vez, executa a tarefa e devolve o resultado da execução para o cliente que o chamou.

O RMI usa um mecanismo padrão aplicado em sistemas RPC (*Remote Procedure Call*), baseado no modelo de chamada de procedimento remota que tem por objetivo

simplificar a programação distribuída, tornando-a semelhante à programação convencional, para se comunicar com objetos remotos, que são *stubs* e *skeletons*. O *stub* funciona semelhante a um *proxy* para o objeto remoto. Quando um objeto local invoca um método num objeto remoto, o *stub* fica responsável por enviar a chamada ao método para o objeto remoto.

Para gerar *stubs* e *skeletons* é necessário utilizar o comando `rmic`. A partir da versão 1.2 do Java SDK, *skeletons* não são mais necessários, agora utiliza um protocolo adicional de *stubs* que 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.

### 3.2.1. Arquitetura RMI

A arquitetura do sistema RMI é dividida em camadas, como mostra a Figura 4, de forma a facilitar sua expansão e inclusão de novas funcionalidades:

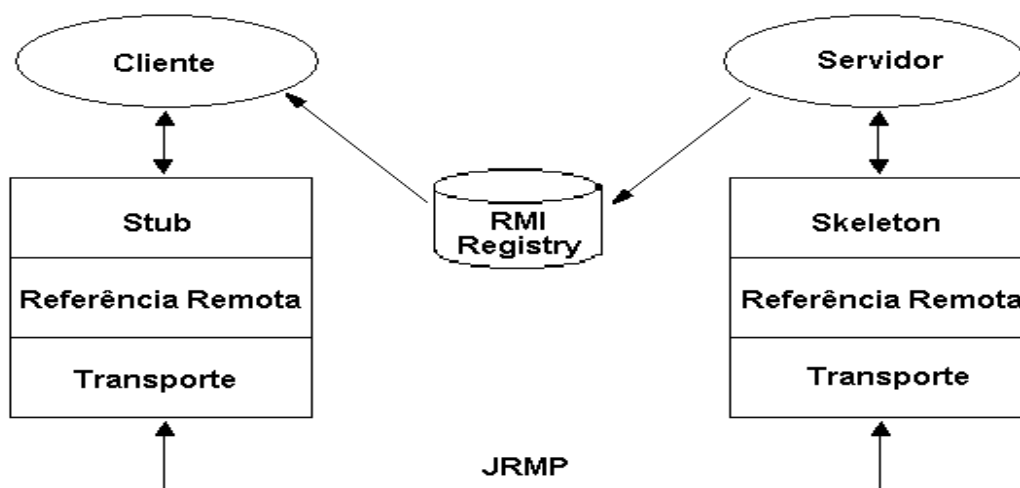


Figura 4. Arquitetura RMI.

**Camada de Stubs & Skeletons:** a camada mais próxima do programador, ou seja, da aplicação cliente e do objeto remoto, é a camada *stub* do cliente e o *skeleton* do servidor, essa camada reside logo abaixo da camada da aplicação. 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 (*unmarshal*) e chamar o método remoto propriamente dito. Após a execução do método remoto, o *skeleton* empacota os dados a serem devolvidos (argumen-

tos de saída, código de retorno do método ou exceção levantada durante a execução) e os envia para o cliente. (SARMENTO, 2003).

**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, que funciona como um *router* entre o cliente e os (eventuais) vários objetos remotos, tais como ativação de objetos remotos dormientes, gerenciamento de tentativas de conexão (*connection retry*), etc. (SARMENTO, 2003).

**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. (SARMENTO, 2003).

### 3.2.2. Características RMI

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

- Serialização de objetos (*Object Serialization*);
- Passagem de parâmetros;
- Polimorfismo distribuído;
- Carga Dinâmica de classes/*stubs* (*dynamic class/stub download*);
- Coleta de lixo distribuída;
- Segurança;
- Ativação de objetos remotos.

### 3.3. RMI-IIOP

RMI-IIOP 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 objeti-



vo 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. (JAVA RMI OVER IIOP DOCUMENTATION, 2009).

As aplicações distribuídas realizadas com a tecnologia RMI-IIOP 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.

A Tabela 1 mostra a interoperabilidade entre as diferentes tecnologias de programação de objetos distribuídos.

Tabela 1. Matriz de interoperabilidade entre as tecnologias RMI, RMI-IIOP e CORBA.

<b>Cliente \ Servidor</b>	<b>RMI (JRMP)</b>	<b>RMI (IIOP)</b>	<b>CORBA</b>
<b>RMI (JRMP)</b>	OK	OK	X
<b>RMI (IIOP)</b>	OK	OK	OK
<b>CORBA</b>	X	OK	OK

Além disso, aplicações desenvolvidas em RMI-IIOP se beneficiam da interoperabilidade entre diferentes tecnologias de programação distribuída 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. (JAVA RMI OVER IIOP DOCUMENTATION, 2009).

### 3.3.1. Arquitetura RMI-IIOP

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, 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 apli-

cações CORBA cliente e servidor em diferentes linguagens de programação. Além disso, a especificação também é utilizada para o mapeamento de chamadas Java RMI para o protocolo de comunicação IIOP. (JAVA TO IDL LANGUAGE MAPPING SPECIFICATION, 2003).

A Figura 5 mostra como funciona a arquitetura RMII-IIOP.

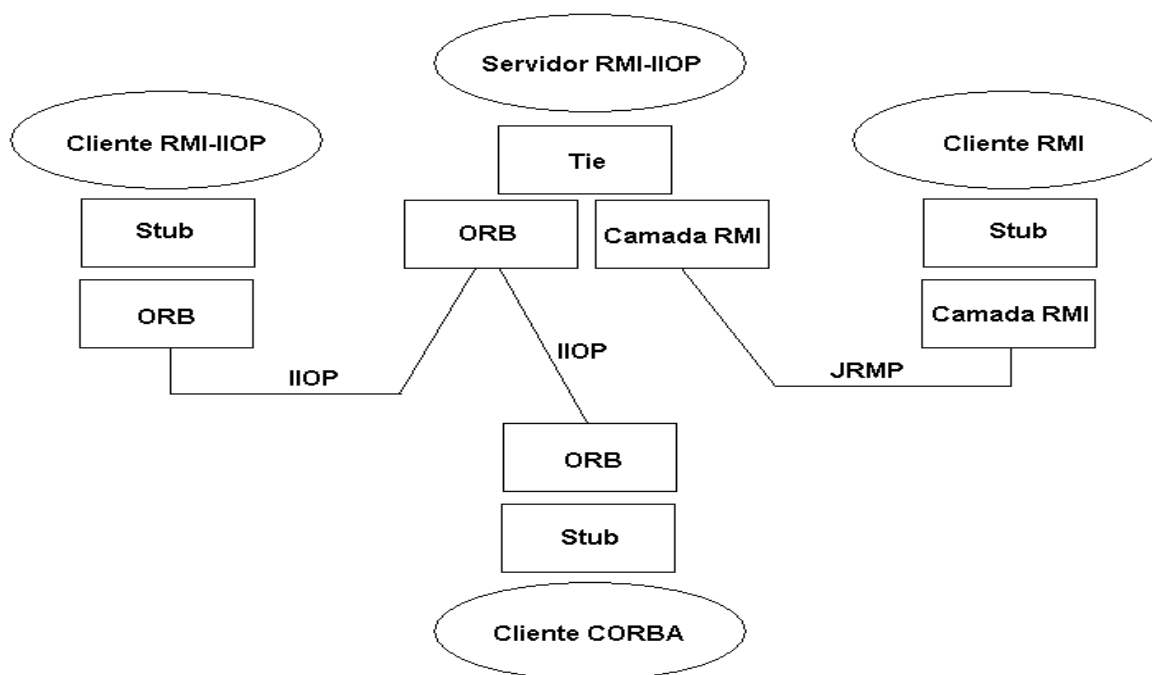


Figura 5. Arquitetura RMI-IIOP.

Em RMI-IIOP um servidor pode disponibilizar a interface de um objeto remoto através dos protocolos JRMP, IIOP ou ambos (*dual export*). A utilização de IIOP por um cliente ou servidor RMI-IIOP implica na utilização de uma ORB para a comunicação entre as partes. (ANDOH; NASH, 1999).

Assim como RMI, a arquitetura RMI-IIOP é baseada na definição da *interface* do serviço remoto onde essa *interface* é interpretada tanto pelo o objeto servidor quanto o *stub* do cliente. 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.

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 acessíveis 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. (ANDOH; NASH, 1999).

### 3.3.2. Características RMI-IIOP

Como características principais de RMI-IIOP são possíveis destacar:

**Interoperabilidade com CORBA:** um dos principais objetivos da implementação de RMI sobre IIOP propriamente dita. Entretanto, conforme descrito anteriormente, 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*. (ANDOH e NASH, 1999).

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:** a coleta de lixo distribuída não é implementada em CORBA e também não pôde ser implementada em RMI-IIOP.

**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: as implementações de passagem de parâmetros do RMI-IIOP têm as mesmas formas de passagem de parâmetros disponíveis em RMI.

**Carga de código (code downloading):** RMI-IIOP disponibiliza carga dinâmica de código para os objetos de Java emitidos pelo valor através de uma conexão de IIOP na mesma maneira que o RMI faz através de uma conexão de JRMP. Entretanto essa funcionalidade só é possível quando as aplicações cliente e servidor são desenvolvidas em Java. (MCMANUS, 2001).

**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. (JAVA TO IDL LANGUAGE MAPPING SPECIFICATION, 2003).

### 3.4. CORBA

Em 1991, um grupo de empresas estabeleceu uma especificação para uma arquitetura de agente de requisição de objeto, conhecida como CORBA (*Common Object Request Broker Architecture*). Em seguida, em 1996, surgiu a especificação CORBA 2.0, que definiu padrões permitindo que implementações feitas por diferentes desenvolvedores se comunicassem. Esses padrões são chamados de *General Inter-ORB Protocol* ou GOIP. Pretendia-se que o protocolo GIOP pudesse ser implementado sobre qualquer camada de transporte orientada a conexões. A implementação do GIOP para a Internet usa o protocolo TCP e é chamada de *Internet Inter-ORB Protocol*, ou IIOP. O CORBA 3 apareceu no final de 1999 e um modelo de componente foi adicionado recentemente. (COULORIS, 2007, p. 670).

O CORBA trabalha com uma linguagem denominada OMG IDL (*Interface Definition Language*), para a definição de *interfaces*, e mapeamentos dessa linguagem para determinadas linguagens de programação, tais como C, C++, Java, Smalltalk e COBOL. (TEIXEIRA JR.; MORAES; TEIXEIRA, 2000, p.14).

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

Em outras palavras, o padrão CORBA, do OMG, é um mecanismo que permite aos objetos de sistemas distribuídos comunicarem-se entre si de forma transparente, não importando onde eles estejam, em que plataforma ou sistema operacional estejam rodando.

## 4. DESENVOLVIMENTO DO PROTÓTIPO

Este capítulo descreve toda a parte de desenvolvimento do protótipo, desde a parte lógica inicial até a sua implementação e testes finais.

### 4.1. Análise de requisitos

Para o desenvolvimento deste projeto objetiva-se desenvolver um sistema de agenda de nomes onde devem ser armazenados os dados referentes ao nome da pessoa, seu telefone e endereço. Com esses dados o sistema deve permitir efetuar consultas pelo no-

me completo ou parte do nome. O diferencial da aplicação desenvolvida é que o sistema pode ser usado local ou de forma distribuída (remota) em qualquer máquina com as configurações necessárias efetuadas. O acesso remoto é transparente para o usuário, que pensa que o acesso é local.

## 4.2. Modelo de Caso de Uso

O Diagrama de Caso de Uso descreve a funcionalidade proposta para o sistema (protótipo), representando uma unidade discreta da interação entre um usuário e o sistema.

O caso do sistema corresponde a um ator nomeado usuário, que tem o papel de solicitar ações, eventos do sistema e receber resposta, e três casos de uso: Cadastrar Pessoa, Consultar Pessoa e Deletar Pessoa, no qual, correspondem a um documento narrativo que descreve a seqüência de eventos feitos por um ator (usuário) no uso do sistema. O diagrama pode ser visualizado na Figura 6.

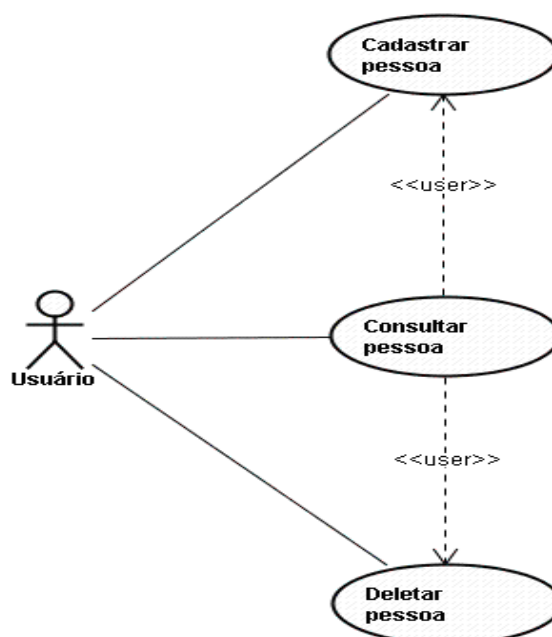


Figura 6. Modelo de Caso de Uso do Projeto Desenvolvido.

A janela principal das aplicações desenvolvidas são todas iguais, o que muda é apenas a barra de título, no qual cada aplicação tem em sua barra de título o nome das suas respectivas tecnologias, como mostra a janela da programação com uso de *Sockets*, mostrada na Figura 7.

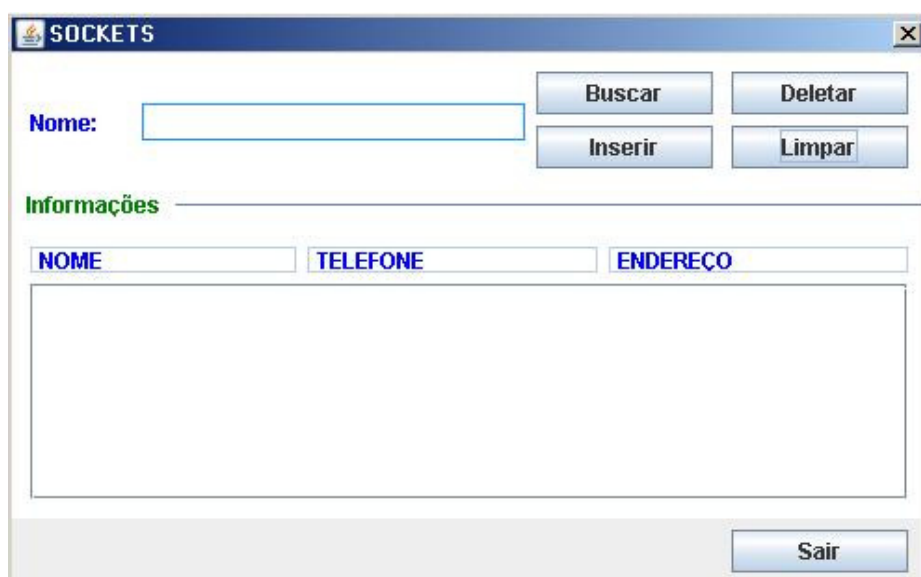


Figura 7. Tela principal da aplicação Sockets.

### 4.3. Comparando as Tecnologias

A Tabela 2 mostrada a seguir contém uma lista das principais características da aplicação desenvolvida e a relação com cada uma das tecnologias implementadas.

Tabela 2. Comparação entre as tecnologias.

Características	SOCKETS	RMI	RMI-IIOP	CORBA
Quantidade de arquivos .java	2	4	4	9
Número total de linhas de código	502	425	468	837
Interoperabilidade	Não	Não	Somente com o CORBA e RMI	Sim
Coleta de lixo	Sim	Sim	Não	Não
Dificuldade de implementação	Baixa	Média	Média	Alta
Orientado a objeto	Não	Sim	Sim	Sim
Qualidade de Segurança	Baixa	Alta	Alta	Alta
Protocolo de comunicação	TCP e UDP	RJMP	RJMP e IIOP	IIOP e GIOP

A coleta de lixo distribuída é realizada apenas nas tecnologias SOCKETS e RMI que é desempenhado automaticamente pela Máquina Virtual do Java (JVM), já a tecnologia CORBA e a tecnologia RMI-IIOP não realizam o mecanismo de coleta de lixo distribuída.

As tecnologias orientadas a objeto são as tecnologias RMI, RMI-IIOP e CORBA nas quais suas conexões são orientadas a objetos, ao contrário da tecnologia SOCKETS que é orientada a *sockets*.

A tecnologia SOCKETS é a única que não apresenta uma política de segurança como nas outras tecnologias citadas nesse trabalho.

## 5. CONCLUSÃO

Devido ao grande aumento da importância das redes de computadores, bem como a popularização da *Internet*, novas mudanças no desenvolvimento de aplicações foram introduzidas, nas quais as aplicações distribuídas se estabelecem quase como um padrão devido às necessidades atuais dos usuários.

A tecnologia SOCKETS é o mecanismo mais utilizado atualmente que possibilita a comunicação entre aplicações, ou seja, cliente/servidor, e também a mais simples e prática, ideal para iniciantes que querem entender como funciona a comunicação entre aplicativos.

A tecnologia RMI e a RMI-IIOP apresentam aplicações semelhantes, nas quais as *interfaces* de objetos remotos são escritas em Java, porém possuem algumas diferenças, como o protocolo de comunicação e o serviço de nomeação que na tecnologia RMI utiliza o protocolo JRMP e o serviço RMI *Registry* e a tecnologia RMI-IIOP utiliza o tanto o protocolo JRMP como o IIOP que é um dos protocolos da tecnologia CORBA. Através desta versatilidade o RMI-IIOP apresenta uma interoperabilidade com as aplicações implementadas nas tecnologias RMI ou CORBA, seja de cliente/servidor ou servidor/cliente.

Outra tecnologia de comunicação estudada neste trabalho que foi citada anteriormente é o CORBA, que permite desenvolver aplicações ou componentes distribuídos em Java que podem se comunicar com aplicações ou objetos distribuídos escritos em outras linguagens que utiliza os protocolos de comunicação IIOP/GIOP, mas não executa tarefas de coleta de lixo como as tecnologias SOCKETS, RMI e RMI-IIOP.

Uma das dificuldades encontradas para o desenvolvimento desse trabalho foi encontrar conteúdo relacionado à tecnologia RMI-IIOP, que muitos confundem como uma implementação CORBA, mas que na verdade é uma combinação de RMI e CORBA, e é apenas implementado na linguagem Java.

A tecnologia que apresentou o maior grau de dificuldade na parte de implementação foi a tecnologia CORBA, que utiliza uma definição da *interface* dos objetos remotos a partir da IDL, no qual, foi necessário entender os comandos da IDL que apresenta uma sintaxe muito parecida com da linguagem C/C++, e entender seu funcionamento que a partir dessa IDL são gerados a *interface*, *stub*, *skeleton* e entre outros arquivos necessários para que a aplicação implementada em CORBA funcione corretamente.

Como propostas para trabalhos futuros têm-se os seguintes itens:

- as aplicações que foram implementadas na linguagem Java para *Desktop* podem ser desenvolvidas para *Web* utilizando JavaEE.
- nas implementações usando CORBA, poderia ser usada a linguagem C/C++ para desenvolver a aplicação cliente e a linguagem Java para desenvolver o servidor, ou vice-versa.

## REFERÊNCIAS

ANDOH, Akira; NASH, Simon. **RMI over IIOP** – Java World. 1999. Disponível em: <[http://www.javaworld.com/javaworld/jw-12-1999/jw-12-iiop\\_p.html](http://www.javaworld.com/javaworld/jw-12-1999/jw-12-iiop_p.html)>. Acesso em: 27 mar. 2007.

ALBUQUERQUE, Fernando. **TCP/IP INTERNET**: programação de sistemas distribuídos HTML, JavaScript e JAVA. Rio de Janeiro: Axcel Books, 2001.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed systems**: concepts and design. 3. ed. Harlow: Addison-Wesley, 2001.

**Java to IDL Language Mapping Specification**. 2003. Disponível em: <<http://www.omg.org/docs/formal/03-09-04.pdf>>. Acesso em: 25 mar. 2007.

**Java RMI over IIOP Documentation**. 2007. Disponível em: <<http://java.sun.com/products/rmi-iiop/index.html>>. Acesso em: 01 mar. 2009.

MCMANUS, Alex. **Java**: API: EJB. 2001. Disponível em: <<http://www.jguru.com/faq/view.jsp?EID=320155>>. Acesso em: 09 abr. 2007.

**RMI (Remote Method Invocation)**. Disponível em: <<http://java.sun.com/javase/6/docs/platform/rmi/spec/rmi-objmodel5.html>>. Acesso em: 01 mar. 2009.

SARMENTO, Luís. **Introdução ao RMI**. 2003. Disponível em: <<http://paginas.fe.up.pt/~eol/AIAD/aulas/JINIdocs/rmi1.html>>. Acesso em: 10 abr. 2007.

TANENBAUM, Andrew S. **Distributed operating systems**. Englewood Cliffs: Prentice Hall, 1995.



TEIXEIRA JR., José Helvécio; MORAES, Luís Felipe M. de; TEIXEIRA, Suzana Ramos. **Uma abordagem para o desenvolvimento de aplicações distribuídas em CORBA usando C++ e orbix.** Universidade Federal do Rio de Janeiro COPPE/UFRJ, 2000. Disponível em: <[http://www.ravel.ufrj.br/arquivosPublicacoes/helvecio\\_rel\\_corba.pdf](http://www.ravel.ufrj.br/arquivosPublicacoes/helvecio_rel_corba.pdf)>. Acesso em: 06 maio 2007.