Helder Lima Santos da Rocha

Comparação entre tecnologias distribuídas Java™: estudos de casos

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal da Paraíba – Campus II como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação e Sistemas Distribuídos

Orientador: José Antão Beltrão Moura – UFPB – CCT – DSC

Campina Grande, Paraíba, Brasil

Maio de 1999

R679c Rocha, Helder Lima Santos da
Comparacao entre tecnologias distribuidas Java : estudos de casos / Helder Lima Santos da Rocha. - Campina Grande, 1999.
109 f.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Java - Linguagem de Programacao 2. Computacao Distribuida - Rede de Computadores 3. Word Wide Web (Servico de Internet) 4. Dissertacao - Informatica I. Moura, Jose Antao Beltrao II. Universidade Federal da Paraiba - Campina Grande (PB) III. Título

CDU 004.43(043)

COMPARAÇÕES ENTRE TECNOLOGIAS DISTRIBUÍDAS JAVA™: ESTUDOS DE CASOS

HELDER LIMA SANTOS DA ROCHA

DISSERTAÇÃO APROVADA EM 26.02.1999

PROF. JOSÉ ANTÃO BELTRÃO MOURA, Ph.D
Orientador

PROF. JACQUES PHILIPPE SAUVÉ, Ph.D
Examinador

PROFª MARIA IZABEL C. CABRAL, D.Sc
Examinadora

PROF. MARCOS ANTÔNIO G. BRASILEIRO, D.Sc
Examinador

CAMPINA GRANDE – PB

"Sempre que um trabalho científico apresenta alguma informação,
ela vem acompanhada de uma margem de erro – um silencioso
mas insistente lembrete que conhecimento algum é completo ou perfeito."
Carl Sagan (1934 – 1996) *em* "The Demon-Haunted World"

"Eu posso estar errado e você pode estar certo,
e com esforço poderemos chegar mais perto da verdade."
Sir Karl Popper (1902 – 1994)

Aos meus pais.

Agradecimentos

A conclusão desta dissertação não teria sido possível sem o apoio e incentivo constante de meu orientador José Antão Beltrão Moura, que esteve presente até quando se encontrava do outro lado da Terra. A orientação de Antão foi essencial para que eu pudesse, a partir do universo de informações acumuladas durante todo o trabalho de pesquisa, definir e concentrar nas idéias essenciais do trabalho, o que permitiu a sua conclusão.

Agradeço também a todos os meus professores, Jacques Sauvé, Joberto Martins, M. Agamemnon Lopes, Ulrich Schiel, William Giozza, Marcelo Alencar e Maria Helena dos quais tive o prazer de ser aluno. Agradeço em especial a Mário T. Hattori (*in memoriam*), cuja motivação, dedicação e poder de realização em mim permanecem como um modelo a ser seguido.

A Ana, Vera e todo o pessoal da Coordenação de Pós-Graduação em Informática, agradeço principalmente pela paciência e dedicação.

Ao prof. Fábio Marinho e Adriana Guerra, da Ibpinet, agradeço pelo apoio e possibilidade de utilizar suas instalações em São Paulo para a realização dos experimentos analisados nesta dissertação. Agradeço a Gilberto Pereira da Costa, da TWA-Conhecer, pelo incentivo e oportunidades que deram origem a pesquisas que tiveram grande influência sobre este trabalho. À Ibpinet, Itelcon, TWA-Conhecer e Sun Microsystems do Brasil devo também a oportunidade de coordenar, elaborar e ministrar programas de treinamento que contribuíram para que eu não perdesse contato com a pesquisa, e adquirisse maior experiência de didática e redação, o que se mostrou indispensável para a realização e conclusão desta dissertação.

Agradeço imensamente à minha família, e a Luciana, por todo o amor, apoio e incentivo constante, e também por terem suportado as minhas longas ausências.

Este trabalho contou com o apoio financeiro do CNPq.

Resumo

Este trabalho apresenta vários estudos de casos que comparam aplicações distribuídas desenvolvidas em Java, executando em ambiente *Windows*. São investigados diversos cenários diferentes, comparando tecnologias nativas da linguagem entre si, e discutindo também a adequação das tecnologias a tarefas comuns em computação distribuída.

Comparamos entre si aplicações Java construídas com APIs de suporte a objetos distribuídos medindo suas taxas de transferência de dados. Analisamos aplicações Web construídas com a Java Servlet API, comparando-as, com programas CGI escritos em C e em Perl. Finalmente, ilustramos o uso de todas as tecnologias analisadas em uma única aplicação que pode ser utilizada sob a forma de um programa executável do *Windows*, de um applet acessível via browser Web, ou de uma página HTML interativa, gerada por um servlet.

As aplicações desenvolvidas rodam, sem necessitar de recompilação, em qualquer plataforma com suporte a Java, permitindo que sejam usadas para medir o desempenho em outros ambientes não explorados no trabalho.

Palavras-chave: Java, computação distribuída, objetos remotos, JDBC, RMI, CORBA, IIOP, CGI, TCP/IP, HTTP, servlets, applets.

Abstract

This text presents several case studies that compare distributed applications developed in Java running under *Windows* environment. A number of different scenarios are investigated, comparing several native Java technologies and discussing the suitability of each technology to common distributed computing tasks.

We compare Java applications developed with APIs that support distributed objects, measuring their data transfer rates. Later, we examine Web applications developed with the Java Servlet API, which are compared to CGI programs written in C and Perl. Finally, we provide a working example which makes use of all technologies within a single application that runs either as a *Windows* executable, as an applet within a Web browser, or via an interactive JavaScript-powered HTML page, generated by a servlet.

The applications will run, without need for recompilation, in any Java-enabled platform. They can thus be used for performance analysis of distributed technologies in other environments not explored in this dissertation.

Keywords: Java, distributed computing, remote objects, JDBC, RMI, CORBA, IIOP, CGI, TCP/IP, HTTP, servlets, applets.

Siglas

Várias siglas comuns na literatura técnica são utilizadas ao longo deste trabalho. Elas estão apresentadas na lista abaixo.

API – Application Programmer's Interface
ASP – Active Server Pages
CGI – Common Gateway Interface
CORBA – Common ORB Architecture
CSS – Cascading Style Sheets
HTML – Hypertext Markup Language
HTTP – Hypertext Transfer Protocol
IIOP – Internet Inter-ORB Protocol
IDL – Interface Definition Language
IP – Internet Protocol
ISAPI – Internet Server API
JDBC – Java Database Connectivity
JDK – Java Development Kit
JIT – Just In Time (compiler)
JRE – Java Runtime Environment
JRMP – Java Remote Method Protocol
JSP – Java Server Pages
JVM – Java Virtual Machine
NSAPI – Netscape Server API
ODBC – Open Database Connectivity
ORB – Object Request Broker
RMI – Remote Method Invocation
RTT – Round Trip Time
SQL – Structured Query Language
TCP – Transport Control Protocol
UML – Unified Modelling Language

Conteúdo

Lista de figuras

Lista de tabelas

Capítulo 1
Introdução

1.1 O uso de Java em ambientes distribuídos

Escolhemos Java como linguagem central deste trabalho por vários motivos. Primeiro, por ser a nossa linguagem preferencial, com a qual estamos familiarizados. Em segundo lugar, entre as linguagens mais populares, nenhuma possui um suporte tão amplo a aplicações distribuídas como Java. É essencial que uma linguagem para computação distribuída tenha características como suporte a operações em rede, suporte a padrões Internet, adesão a padrões abertos, ampla aceitação no mercado, independência de plataforma e suporte a tecnologias de objetos distribuídos. Java oferece tudo isso, nos dando mais uma justificativa à escolha da linguagem.

Java é uma linguagem orientada a objetos desenvolvida em 1995 pela *Sun Microsystems*. Surgiu inicialmente como uma linguagem voltada ao desenvolvimento de aplicações para a Internet. Por este motivo, provavelmente, foi construída com um amplo suporte a todo tipo de operação em rede. Com a popularidade alcançada através da Web, a linguagem se tornou em pouco tempo, um padrão de mercado e está em processo de se tornar também um padrão aberto ISO (International Standards Organization). Suas APIs nativas suportam vários outros padrões abertos como ODBC, HTTP e CORBA.

Uma das principais características de Java é o fato de ser uma linguagem independente de plataforma. Seu código-fonte, após ser compilado em uma linguagem de máquina *virtual*, roda sem modificações em um amplo universo de plataformas e sistemas operacionais diferentes. Essa característica é muito importante, principalmente para aplicações distribuídas.

Java oferece suporte à computação distribuída através de sua API, que consiste de diversas classes e interfaces organizadas em módulos denominados de *pacotes*. Utilizando as classes e interfaces dos pacotes que dão suporte a operações de rede, é possível desenvolver

aplicações de rede com uma facilidade que linguagens populares como *C*, *Delphi* ou *Perl* não oferecem.

Os principais pacotes[1] Java que oferecem suporte a operações distribuídas e que serão explorados neste trabalho são:

- `java.io`. Este pacote oferece suporte a operações de entrada e saída. Suas classes e interfaces são usadas na construção e filtragem de fluxos (*streams*) de dados que podem consistir de tipos primitivos ou objetos, além de classes que representam arquivos e permitem operações no sistema de arquivos.
- `java.net`. Este pacote oferece suporte a aplicações de rede em geral. Contém classes e interfaces que permitem a implementação de clientes, servidores e protocolos TCP/IP utilizando soquetes TCP, datagramas UDP, *multicasting*, endereços Internet, conexões HTTP e URLs.
- `java.rmi`. Suporta a arquitetura de objetos remotos RMI – *Remote Method Invocation*, que permite o desenvolvimento de aplicações que chamam métodos em objetos localizados em máquinas virtuais diferentes. Baseia-se no protocolo JRMP – *Java Remote Method Protocol* que permite a comunicação entre máquinas virtuais Java.
- `java.sql`. Pacote que suporta JDBC – *Java Database Connectivity*. Com essas classes e interfaces é possível desenvolver programas em Java que se comunicam com qualquer banco de dados relacional que suporte as operações mínimas do SQL92. Também oferece suporte ao desenvolvimento de *drivers* JDBC.
- `javax.servlet`. API que oferece suporte ao desenvolvimento de componentes de servidor – os *servlets*. Servlets são componentes que rodam dentro de uma aplicação servidora. Servlets HTTP podem ser usados como alternativa eficiente, aberta e independente de plataforma a tecnologias atualmente utilizadas nos servidores HTTP como CGI[2] ou ISAPI[3].

---

[1] Os pacotes da API Java seguem uma hierarquia baseada em estruturas de diretórios e subdiretórios, geralmente comprimidos e colecionados em arquivos de biblioteca ZIP ou JAR (Java Archive). O pacote java.awt.image, por exemplo, corresponde ao conteúdo do diretório java/awt/image dentro do arquivo de biblioteca rt.jar (distribuição Java 2).

[2] *Common Gateway Interface*. Especificação aberta para aplicações *gateway* no servidor chamadas pelo browser.

- `org.omg.CORBA` e `org.omg.cosNaming`. Estes pacotes oferecem classes, interfaces e subpacotes que permitem o desenvolvimento de aplicações Java que usam a tecnologia de objetos remotos CORBA.
- `javax.rmi`. Oferece suporte a RMI usando o protocolo IIOP (Internet Inter-ORB Protocol) e o serviço de nomes da plataforma Java permitindo que objetos RMI se comuniquem com objetos CORBA.
- `java.naming`. Pacote que oferece suporte a serviços de nomes e diretório. Utilizado neste trabalho para registrar nomes de objetos RMI quando utilizado via IIOP.

Todos os pacotes com prefixo `java` fazem parte do que se chama de "Core API". Os pacotes com prefixo `javax` pertencem a extensões da API suportadas pela *JavaSoft*. São chamadas de "*Standard extensions*" (extensões padrão) e geralmente não são distribuídas com a API núcleo (parte do JRE – *Java Runtime Environment*). Os pacotes CORBA, embora não tenham prefixo `java`, são distribuídos com a API núcleo. A documentação Java os chama de "Core extensions" (extensões ao núcleo).

1.2 Desempenho de aplicações Java

As maiores críticas em relação à linguagem Java ocorrem em relação ao desempenho. O problema não é da linguagem Java em si, mas da *plataforma* Java, que emula uma máquina virtual. Os programas em Java são compilados de forma a gerar código binário para essa máquina virtual, que roda como uma aplicação sobre o sistema operacional nativo.

O desempenho, porém, tem se mostrado na prática, bastante razoável. Trabalhos independentes como [KERN98] e [GROT98] mostram resultados de testes de *benchmark* nas recentes implementações da máquina virtual Java (*Windows*, *Solaris*) onde programas em Java levam 20 a 50% mais tempo que aplicações em C/C++ para executar uma tarefa, e não mais 10 a 20 vezes o mesmo tempo, como ocorria quando a linguagem foi lançada. A maior parte do ganho em desempenho deve-se aos interpretadores com compilador *Just-In-Time* (JIT). Esses interpretadores traduzem o código binário Java para a linguagem nativa durante a execução. Mas o próprio interpretador tem melhorado bastante, independente do JIT. [GROT98] mostra que enquanto o interpretador JIT (*default*) do JDK1.1.7 no *Windows95* é

---

[3] *Internet Server Application Programmer's Interface*. Alternativa da Microsoft ao CGI através de uma API que permite desenvolver DLLs que estendem o servidor com novos métodos e implementações.

cinco vezes mais veloz que o interpretador convencional (não-JIT) do mesmo pacote, o interpretador *default* do JDK1.02 (não-JIT) leva quase 12 vezes mais tempo para rodar o mesmo programa.

A maior parte das análises de desempenho relacionadas com a linguagem Java refere-se ao desempenho da *plataforma* Java. Vários documentos da *Sun* concentram-se na análise do desempenho dos interpretadores Java. Os mais recentes exploram as vantagens do novo compilador-interpretador adaptativo *HotSpot*, ainda instável e não tão eficiente quanto prometido. *Benchmarks* como [SUN97] e [CAFF98] mostram o desempenho (resumido em um único número) de máquinas virtuais Java de diversos fabricantes em diversas plataformas (inclusive browsers), dando uma ligeira vantagem às máquinas virtuais da *Microsoft*, e uma larga desvantagem às máquinas virtuais em sistemas menos populares, que não possuem interpretadores *Just-In-Time* (JIT).

Benchmarks, porém, costumam apresentar resultados controversos, baseados em critérios nem sempre muito claros. [KERN98] faz uma comparação mais ampla e mais clara. Compara não só Java, mas várias linguagens de programação modernas: *C*, *Awk*, *Tcl*, *Java*, *VB*, *Limbo* e *Scheme* em tarefas selecionadas que utilizam os recursos do sistema de forma diferente.

Encontramos poucas discussões que se fundamentassem em *experimentos* para comparar o desempenho de tecnologias de objetos distribuídos. A maior parte discute outros aspectos como integração, facilidade de desenvolvimento, etc. e usa *argumentos* para concluir que uma tecnologia tem um desempenho inferior a outra [FARL97][CURT97][BARR99].

Em [BARR99], porém, o autor apresenta uma aplicação Java para medir a taxa de transferência de dados em aplicações usando CORBA (API nativa da *Sun*). Nós testamos a aplicação do autor, mas não a consideramos adequada para a comparação com outras tecnologias de rede como RMI e TCP/IP (nosso objetivo). Nos nossos experimentos, desenvolvemos um conjunto de aplicações para medir a taxa de transferência de bytes entre um cliente e um servidor, utilizando o mesmo princípio empregado em [BARR99] (de enviar vetores de dados e o instante de envio). A aplicação apresentada em [BARR99] servia apenas para comparar o uso de tecnologias CORBA. Projetamos uma estrutura básica que pudesse ser estendida facilmente, para criar aplicações que servissem para analisar outras tecnologias de rede. Separamos, o quanto possível, a lógica das aplicações das partes envolvidas exclusivamente com a transferência de dados. Assim acreditamos ter obtido aplicações mais

simples e maior facilidade no desenvolvimento de aplicações "equivalentes" para medir a taxa de transferência usando TCP/IP, RMI e RMI/IIOP, além de CORBA.

A taxa de transferência de dados não é o único fator que influencia o melhor ou pior desempenho de uma aplicação distribuída. Decidimos medi-la, em primeiro lugar, para manter a consistência com a literatura pesquisada ([BARR99]). Em segundo lugar, por tratar-se de uma medida cujos fatores de imprecisão poderiam ser melhor controlados na infraestrutura que tínhamos à disposição, sendo, portanto, uma medida mais robusta.

Em uma aplicação real, os dados brutos geralmente precisam ser transformados em informação útil no cliente ou no servidor e isto implica uma carga adicional que poderá ser realizada de uma maneira mais eficiente por uma certa tecnologia que por outra. O consumo de recursos também é um fator importante, já que a necessidade de alocar mais memória ou realizar mais cálculos poderá afetar o tempo que o programa requer para decodificar a informação. Apesar desses aspectos serem importantes em um estudo *completo* acerca do desempenho de uma aplicação distribuída, não são explorados neste trabalho. O nosso estudo concentra-se em medir a taxa de transferência que, embora possa ser considerada uma medida insuficiente, permite que se tenha uma visão do desempenho da aplicação e oferece um ponto de partida que poderá orientar trabalhos futuros.

Algumas das tecnologias que analisamos têm maior aplicação na Web, onde estudamos o desempenho de agentes (cliente e servidor), que podem ser otimizados com a linguagem Java. Vários estudos existem sobre as limitações do protocolo HTTP [SLOT97] [SPERO95] [EDWA99], que não serão resolvidas no cliente ou servidor. Mas a ineficiência dos servidores tem grande peso nos problemas de desempenho da rede. Em [MURP99], o autor cita um trecho de um artigo de Andrew Odlyzko (AT&T) que afirma, em estudo realizado para verificar origens de um congestionamento na Internet, no qual "(...) 42% dos atrasos foram causados por transmissões de rede, 13% devido ao serviço DNS e os restantes 45% devido aos *servidores*". Há uma necessidade, portanto, em melhorar a eficiência dos servidores. Publicações como [SUN97] e [MICR97] argumentam que a tecnologia CGI – principal tecnologia de extensão do servidor – faz exatamente o contrário: torna o servidor menos eficiente. Em cada publicação, propõem alternativas ao CGI (servlets e ISAPI, respectivamente).

Por tratar-se de tecnologia Java, procuramos comparações entre servlets e CGI. Novamente, a grande maioria das publicações discute outras vantagens como programação e

argumenta em favor do desempenho dos servlets, mas não apresenta dados comparativos, resultantes de experimentos facilmente reproduzíveis. Os argumentos são insuficientes. Pode-se argumentar, por exemplo, que uma certa aplicação CGI escrita em C seja mais eficiente que um servlet escrito em uma linguagem interpretada como Java. No nosso trabalho, portanto, procuramos medir os tempos de resposta em aplicações usando servlets, com outras, equivalentes, usando CGI, complementando essas medições com uma discussão sobre as vantagens e desvantagens de se utilizar esta ou aquela tecnologia.

1.3 Objetivos

O objetivo do trabalho é analisar o comportamento de algumas aplicações desenvolvidas usando tecnologias disponíveis como APIs Java nativas[4]. A análise consiste de medidas de desempenho, discussão de vantagens, desvantagens e benefícios de cada tecnologia. No final é apresentada uma tabela comparativa que poderá fornecer subsídios e auxiliar na escolha de uma ou outra tecnologia.

As medições são obtidas através de experimentos que podem ser facilmente repetidos em outros ambientes. Complementamos a análise dos resultados medidos apresentando diferentes visões dos dados e uma discussão acerca de características como portabilidade, reutilização, facilidade de manutenção e programação.

1.4 Metodologia, plataforma e limitações

1.4.1 Metodologia utilizada nos experimentos

Os experimentos, independentemente do seu objetivo, levam em consideração três aspectos: 1) a tarefa a ser realizada pelo programa, 2) a arquitetura e tecnologia de rede utilizada, e 3) a linguagem de programação utilizada. A plataforma e o ambiente de rede são sempre os mesmos em cada experimento. O seu desempenho só é importante na medida em que possa provocar interferências heterogêneas nos resultados que visam comparar as arquiteturas, linguagens ou tarefas entre si. A linguagem de programação é Java em todos os experimentos exceto no capítulo 3, onde são realizadas comparações entre arquiteturas usando linguagens diferentes.

---

[4] Inclui extensões padrão distribuídas pela Sun. Não inclui APIs de outros fabricantes.

Os experimentos consistem da medição de valores, principalmente instantes e intervalos de tempo durante a execução de aplicações distribuídas, geralmente em uma rede local formada por dois computadores. Em alguns casos, essa rede foi simulada em um único computador, com o objetivo de atenuar erros devido a interferências e limitações da plataforma de rede (veja seção 1.4.3).

Os valores medidos foram organizados em planilhas através das quais se obteve valores de interesse (geralmente valores médios de medições repetidas) e intervalos de confiança desses valores. Para expor visões alternativas dos dados, foram também produzidas tabelas com valores calculados. Essas tabelas permitem que certos aspectos do comportamento das aplicações durante os experimentos sejam colocados em destaque.

Todos os valores apresentados na forma de tabelas também foram expostos na forma de gráficos. A vantagem dessa exposição é a possibilidade de destacar padrões e tendências, permitindo a compreensão mais rápida do comportamento observado, que as tabelas que mostram valores individuais.

Os resultados observados muitas vezes levaram à ampliação do problema analisado, provocando novas medições, experimentos ou cálculos, para analisar efeitos que consideramos relevantes. Alguns desses resultados secundários são menos precisos e menos confiáveis devido a limitações discutidas na seção 1.4.3.

As tarefas realizadas foram escolhidas de acordo com o objetivo proposto. Nos testes envolvendo diferentes linguagens, tentamos ao máximo garantir a equivalência entre as aplicações, mantendo-as simples. Procedemos de forma semelhante nos testes entre tecnologias de objetos distribuídos, utilizando os recursos de orientação a objeto de Java para manter as partes (classes e interfaces) da aplicação que nada têm a ver com os protocolos de transferência, idênticas. A aplicação que ilustra diferentes interfaces do usuário é mais complexa (devido aos objetivos aos quais se destina), mas ainda assim a maior parte das classes e interfaces utilizadas é reutilizada em todos os cenários analisados.

1.4.2 Plataforma de testes

Em todos os experimentos, utilizamos máquinas *Pentium* 200, com 32 ou 64MB de memória, rodando *Windows 95* (OSR2 ou OSR4). Nos experimentos relatados no capítulo 2 utilizamos duas máquinas (*Pentium* 200/64 com *Windows95* OSR4) conectadas a uma rede 10Base-T com taxa de transferência nominal de 10Mbps.

Utilizamos quatro máquinas virtuais Java (JVM): a máquina *default* da *Sun* (Java 2 JVM with JIT compiler), a máquina original da *Sun* (sem o JIT compiler) e as JVMs da *Netscape* ou *Microsoft* (não foi possível usar ambas, por problemas de compatibilidade), embutidas nos seus respectivos browsers. Essas duas últimas foram utilizadas apenas pelos applets.

Utilizamos o interpretador *GNU Perl 5 for Windows 95* (build 110) para executar as aplicações Perl. Os servidores Web *Netscape FastTrack 2.0 for Windows 95* para servir páginas e executar aplicações CGI (em Perl e em C) e servlets (em Java) através do módulo *Live Software JRun Servlet Engine 2.3*, acoplado ao servidor. Em um dos experimentos não foi possível usar o *JRun* e utilizamos o *Servletrunner* da *Sun*.

1.4.3 Limitações deste trabalho

Vários fatores limitam confiabilidade das medidas de desempenho obtidas neste trabalho. Os recursos de medição de tempo e consumo de recursos fornecidos pelas linguagens utilizadas e pelo sistema operacional *Windows* são pouco adequados à medição de tempos de execução, levando a resultados muitas vezes inconsistentes, com variações durante a execução da aplicação. O gerenciamento de memória nunca ocorre da mesma forma em máquinas diferentes, mesmo quando a sua configuração é idêntica. Muitas vezes, até na mesma máquina um experimento pode apresentar um resultado bastante diferente devido à coleta automática de lixo e outros efeitos do gerenciamento de memória pelo sistema operacional. A maior parte desses efeitos é inacessível ao usuário e nem sempre pode ser prevista, já que não há recursos nativos no *Windows* para prevê-los.

Essas limitações não são exclusividade deste trabalho. Já foram observadas em experimentos como [KERN 98], que compara o desempenho de linguagens Java, C e outras em mais de uma plataforma. Nessa publicação, os autores concluem que "aparenta haver pouca esperança em prever desempenho de outra maneira que não seja bastante genérica; se há uma única clara conclusão, é que nenhum resultado de *benchmark* deve ser levado em consideração apenas por seus resultados".

No nosso trabalho, valores absolutos não têm relevância, pois o foco do trabalho é comparar aplicações rodando em uma mesma plataforma, utilizando interfaces do usuário ou protocolos diferentes. Nós tentamos reduzir ao máximo a influência dos problemas causados pela plataforma e sistema operacional, realizando os mesmos experimentos mais de uma vez, em ocasiões diferentes, eliminando um ou outro que tenha apresentado comportamento

irregular, causado, na maior parte das vezes, por operações de gerenciamento de memória do sistema ou devido a interferências na rede[5].

Apesar de todas as limitações, os resultados obtidos permitem que diferenças significativas sejam percebidas e que algumas conclusões possam ser alcançadas. Depois, como já afirmamos, na maior parte das medições os valores absolutos não têm grande relevância, sendo mais importante o comportamento relativo entre as aplicações.

1.4.4 Contribuições

As principais contribuições que este trabalho espera oferecer são:

- Apresentar estudos de casos com uma análise comparativa de tecnologias recentes, populares, mas pouco estudadas (uma delas foi lançada em versão beta em janeiro/1999).
- Apresentar aplicações multiplataforma escritas em Java utilizando cada uma dessas tecnologias.
- Apresentar uma tabela comparativa com subsídios que poderão nortear a opção por uma ou outra tecnologia analisada.

1.5 Organização

Este trabalho consiste de duas partes: esta dissertação impressa, e um disquete, que contém todo o código-fonte e código compilado (Java ou *Windows*) utilizado nos experimentos, além de documentação em HTML (padrão Java) sobre todas as aplicações.

A dissertação está organizada em cinco capítulos (esta introdução e mais quatro) e possui ainda três apêndices.

O capítulo 2 analisa as diferenças e semelhanças entre tecnologias de objetos distribuídos (RMI e CORBA) e as compara com uma solução nativa usando os recursos TCP/IP oferecidos pelo pacote `java.net`.

O capítulo 3 compara as tecnologias CGI e Servlet API. São testadas oito aplicações Web diferentes: 5 programas CGI e 3 servlets. Com os dados obtidos nas medições, as duas tecnologias são comparadas.

---

[5] Veja maiores detalhes sobre as limitações nos capítulos onde cada experimento é descrito.

No capítulo 4, são apresentados exemplos de aplicações Java que usam todas as tecnologias até então apresentadas. A aplicação é então usada para medir instantes de tempo durante a execução de aplicações rodando como servlets, applets e programas do *Windows*.

O capítulo 5 apresenta conclusões baseadas nos resultados obtidos nos capítulos 2 a 4 e resume as informações em vários quadros comparativos.

Os apêndices A, B e C contêm, respectivamente, detalhes sobre a estrutura das aplicações e implementação em Java das aplicações utilizadas nos experimentos dos capítulos 2, 3 e 4.

Famílias e estilos de fontes diferentes são usadas em várias partes do texto para realçar ou destacar palavras que possuem significado especial. A fonte `Courier` é usada principalmente em listagens de programas. Quando dentro de um parágrafo, representa nomes de classes Java, URLs e nomes de arquivos. *Garamond Italic* é usada geralmente para destacar nomes de aplicativos.

Referências bibliográficas, que aparecem no texto da forma [ABCD99], estão listadas no final da dissertação.

Capítulo 2
Comparação entre tecnologias de objetos distribuídos

2.1 Introdução

O objetivo deste capítulo é comparar tecnologias de objetos distribuídos em Java. Realizaremos a medição do tempo e taxa de transferência de dados entre aplicações usando tecnologias de objetos distribuídos na comunicação em rede. Serão medidos tempos de transferência de dados *brutos*, sem considerar o aproveitamento da informação que eles possam conter. De posse dessa informação, discutiremos a adequação de cada tecnologia em diferentes cenários.

Para a realização dos experimentos, desenvolvemos quatro aplicações distribuídas. Três delas utilizam tecnologias de *objetos distribuídos*[1], combinando duas arquiteturas diferentes (RMI ou CORBA) com dois protocolos de transporte (IIOP ou JRMP):

- CORBA (*Common ORB Architecture*) sobre IIOP (*Internet Inter-ORB Protocol*).
- RMI (*Remote Method Invocation*) sobre JRMP (*Java Remote Method Protocol*).
- RMI sobre IIOP

A quarta aplicação foi desenvolvida usando apenas o pacote `java.net` – que oferece o suporte TCP/IP nativo da linguagem Java – permitindo a representação de soquetes, endereços Internet, portas de serviços, etc. A aplicação utiliza-se de um protocolo TCP/IP proprietário projetado especificamente para transferir *bytes* entre o cliente e servidor da forma mais eficiente possível. A finalidade dessa última aplicação é traçar um cenário de referência com o qual poderemos comparar as outras tecnologias. Como todas as tecnologias de objetos distribuídos fazem uso do pacote `java.net` (de forma transparente ao programador), espera-se a melhor taxa de transferência desta última aplicação.

---

[1] Tecnologias de objetos distribuídos são aquelas que permitem a manipulação de objetos em máquinas remotas.

Conhecendo-se a quantidade de *bytes* transmitidos em cada requisição do cliente, poderemos calcular a taxa média de transferência de dados e usar esse valor para comparar o desempenho das quatro tecnologias.

2.2 Experimento para medir a *taxa de transferência de dados*

As quatro aplicações utilizadas neste experimento estão localizadas no disquete que acompanha esta dissertação e podem ser executadas a partir do subdiretório `/jad/apps3/`. É preciso que um ambiente de execução Java 2 esteja disponível (em qualquer plataforma).

Cada aplicação possui uma parte cliente e uma parte servidora. Ao total são oito programas executáveis que devem ser usados em pares. Os programas podem ser executados em uma mesma máquina ou em máquinas diferentes ligadas em rede TCP/IP. É preciso instalar o pacote completo (descompactar os arquivos do disquete) em *todas* as máquinas onde um dos programas (cliente ou servidor) irá executar.

2.2.1 Funcionamento das aplicações e descrição do experimento

O objetivo das aplicações é medir o tempo entre o momento em que um cliente envia uma certa quantidade de bytes a um servidor, e o momento em que este o recebe (tempo fim-a-fim). As informações recebidas não são aproveitadas e podem ser descartadas (o tempo de transferência de *informação* não é registrado).

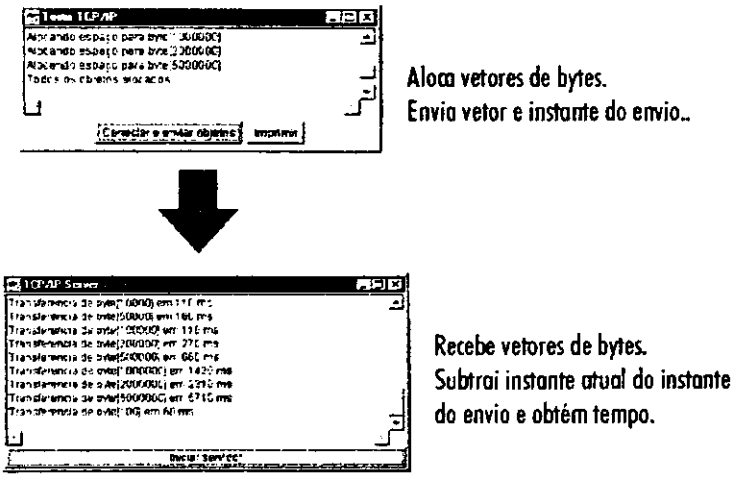


Figura 2-1 Aplicação cliente cria vetores (coleções) de bytes que em seguida são enviados para a aplicação servidora juntamente com um inteiro longo contendo o instante do envio.

Já que o tempo de partida será registrado em uma máquina e o tempo de chegada recuperado em outra, é importante que os relógios das duas máquinas estejam sincronizados (e que se mantenham sincronizados no decorrer do experimento).

Uma limitação que enfrentamos foi a dificuldade de sincronização dos relógios de máquinas *Pentium/Windows* conectadas em rede, e a capacidade de manter os mesmos sincronizados pelo menos durante a execução das aplicações de uma etapa dos experimentos. Tínhamos

10 máquinas à disposição e as primeiras duas que selecionamos mostraram uma diferença de 330ms entre si uma hora após a sincronização. No final, conseguimos realizar os experimentos (que duravam de 15 a 30 minutos) em uma máquina que atrasava 50ms em uma hora.

A precisão dos relógios e a capacidade de medir tempo no *Windows* e nas linguagens Java, C, JavaScript e Perl, utilizadas nos experimentos, introduziram outra limitação, que prejudicou a confiabilidade dos valores medidos próximos ou inferiores a 110ms. Não foi possível medir valores com intervalos menores que 50 ou 60ms. Um valor inferior a 50ms poderia ser registrado como 50ms ou 0. Um valor maior que 50ms e menor que 200ms, poderia assumir apenas um dos seguintes valores: 60ms, 100ms, 110ms, 150ms, 160ms, 170ms ou 180ms. A sincronização entre máquinas, dependente desses fatores, pode apresentar uma diferença de até ±110ms. Nos testes que envolviam linguagens diferentes, tivemos também a limitação imposta pela inexistência de uma maneira padrão de medir tempo durante a execução.

Para evitar a sincronização, poderíamos ter optado por medir o tempo de requisição e resposta (RTT - *Round Trip Time*) em vez do tempo fim-a-fim. Um dos motivos que nos levou a escolher este último foi o maior controle sobre os dados medidos. A medição do RTT inclui, além do tempo fim-a-fim, tempo de processamento no servidor que está sujeito a variações fora do nosso controle (principalmente gerenciamento de memória do sistema).

Os resultados, portanto, incluem efeitos do gargalo de rede que, embora afetem todas as aplicações por igual, não nos permitirá tirar conclusões muito precisas sobre *o quanto* uma tecnologia é melhor que outra.

Estando as aplicações cliente e servidor executando, e o servidor iniciado, o cliente poderá iniciar o envio de objetos. Qualquer uma das 4 aplicações cliente (CORBA, RMI/IIOP, RMI/JRMP ou TCP/IP) cria um determinado número de vetores contendo inteiros de 8 bits (tipo *byte*). No *nosso* teste criamos 14 vetores com tamanhos entre 10^5 e $5 \cdot 10^6$ bytes. O usuário do programa cliente pode escolher quantas vezes esses vetores serão enviados.

Depois que todos os vetores são criados, eles são enviados ao servidor em seqüência, uma ou mais vezes. Junto com os dados é enviado um número inteiro longo (mais 8 bytes) que contém o instante[2] (no relógio da máquina cliente) em que os dados foram enviados.

O servidor aguarda conexões do cliente. Quando recebe todos os elementos de um vetor, registra o instante em que isto ocorre (baseado no relógio da máquina servidora) e guarda a di-

---

[2] O instante corresponde ao tempo em milissegundos desde 01/Jan/1970 às 0:00:00 GMT (padrão UTC).

ferença entre esse instante e o instante armazenado pelo cliente. Quando todos os dados tiverem sido enviados, o cliente envia mais um comando que imprime um relatório com todos os tempos classificados por tamanho do vetor.

2.2.2 Realização do experimento

Os programas foram executados em duas máquinas *Intel Pentium* 200 MMX com 64MB de memória interligados em rede TCP/IP com arquitetura 10-Base-T e velocidade nominal de 10Mbps. As duas máquinas rodavam *Windows 95* (OSR4) e tinham a plataforma Java 2 instalada (JDK1.2) com a extensão RMI-IIOP beta 0.2.

Como o teste foi realizado em máquinas diferentes, seus relógios foram sincronizados. Não conseguimos, porém, uma precisão absoluta. Diferenças inferiores a 110ms não puderam ser detectadas. Depois da sincronização, a diferença ainda aumentava 50ms (aproximadamente) em cada hora. Como os experimentos duravam sempre menos que 30 minutos e as máquinas sempre eram sincronizadas antes de cada experimento, essas imprecisões afetaram todos os experimentos praticamente por igual. Pode ter prejudicado as tecnologias mais lentas, mas de forma imperceptível já que os computadores eram incapazes de registrar intervalos de tempo menores que 50ms. Como o objetivo do experimento é comparar as quatro tecnologias, imprecisões que afetam todas as tecnologias por igual têm pouca relevância.

Para executar a aplicação, é necessário que cada cliente e servidor seja iniciado a partir de uma janela de comando no subdiretório `/jad/apps3/` através do interpretador Java. A sintaxe geral é semelhante para servidores e clientes.

```
java nome.completo.da.Classe [nome da maquina do servidor]
```

O nome do servidor é obrigatório apenas nos programas cliente que rodam em máquina diferente daquela onde roda o servidor. Os nomes completos de cada aplicação são:

- Cliente e servidor TCP/IP (usando soquetes TCP) com protocolo proprietário:

  ```
  bench.server.tcpip.Server
  bench.client.tcpip.Client
  ```

- Cliente e servidor CORBA (usando IIOP):

  ```
  bench.server.corba.Server
  bench.client.corba.Client
  ```

- Cliente e servidor RMI (usando JRMP):

```
bench.server.rmi.Server
bench.client.rmi.Client
```

- Cliente e servidor RMI (usando IIOP):

```
bench.server.riiop.Server
bench.client.riiop.Client
```

Para iniciar uma aplicação cliente (fig.2-2), executamos a partir do diretório `/jad/apps3/`:

```
java bench.client.tipo.Client nome_da_maquina
```

onde *tipo* é `tcpip`, `corba`, `rmi` ou `riiop`. Se o nome da máquina for omitido, a aplicação procurará por um servidor local. A aplicação servidora, na máquina "`nome_da_maquina`" foi iniciada de forma semelhante:

```
java bench.server.tipo.Server
```

Ao serem executados, o servidor e o cliente devem aparecer na tela como aplicações gráficas do *Windows* (ou do sistema operacional onde estão sendo executadas). Antes de iniciar a transferência de dados, é necessário *iniciar* o servidor. As aplicações que realizam a comunicação usando objetos remotos (todas menos a aplicação TCP/IP) necessitam ainda que um servidor de nomes esteja rodando *antes* que o servidor seja iniciado. Para RMI usando JRMP, executamos o `rmiregistry` na máquina do servidor:

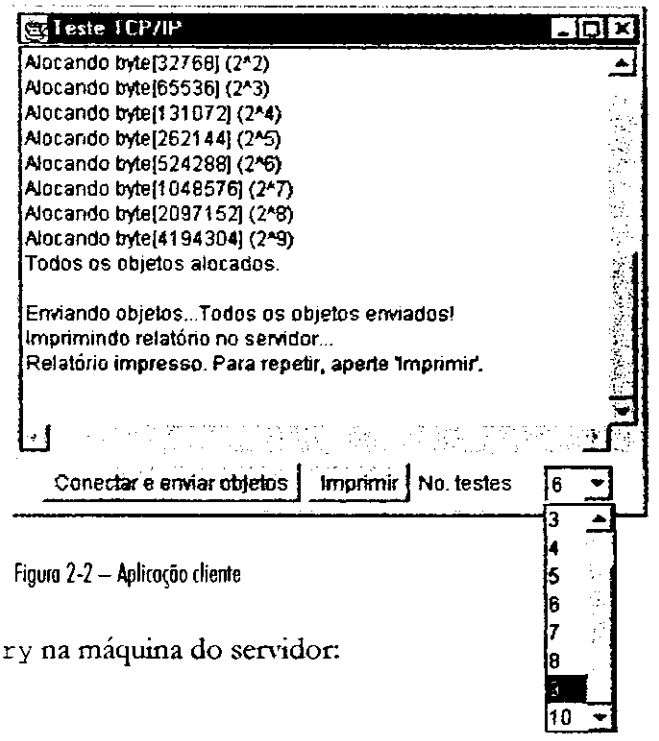


Figura 2-2 – Aplicação cliente

```
start rmiregistry
```

Para usar as tecnologias baseadas em IIOP (CORBA ou RMI), iniciamos o servidor de nomes do JRE[3] em *ambas* as máquinas do cliente e do servidor:

```
start tnameserv
```

[3] Java Runtime Environment

Figura 2-3 – Aplicação servidora

Com as duas aplicações rodando, damos partida no servidor apertando o botão "Iniciar Servidor" situado na parte inferior da janela da aplicação (veja figura 2-3). Os servidores de objetos distribuídos só tentarão estabelecer uma conexão com o `rmiregistry` ou `tnameserv` depois que o botão for apertado. Quando o servidor conseguir registrar o objeto no servidor de nomes e estiver pronto para receber conexões aparecerá uma mensagem informando o sucesso na sua janela principal.

Com o servidor no ar, esperando clientes, dirigimo-nos à máquina cliente e escolhemos o número de vezes que cada vetor será enviado ao servidor, na parte inferior da aplicação (figura 2-2). No nosso experimento, enviamos cada vetor 10 vezes (e repetimos este experimento 5 vezes, totalizando 50 valores). Após a seleção, apertamos o botão "Conectar e enviar objetos" fazendo o cliente construir cada coleção de bytes e os enviar ao servidor.

O botão imprimir (na aplicação cliente) envia uma instrução para imprimir uma nova cópia do relatório (uma já deve ter sido impressa automaticamente). Para salvar os resultados em arquivo, selecionamos os dados do relatório, copiamos para o *clipboard* do *Windows*, e depois os colamos em um arquivo de textos posteriormente importado por uma planilha.

2.2.3 Análise dos Resultados

Executamos cada uma das aplicações da forma descrita na seção anterior e enviamos um grupo de 14 vetores, com coleções contendo entre 100000 a 5000000 bytes[4] cada, do

[4] A faixa de valores pode ser alterada no código-fonte da classe bench.ClientFrame. Escolhemos esta faixa de valores por apresentar dados dentro de um intervalo de confiança aceitável (com nível de confiança de 95%). Na plataforma utilizada, transferências menores que 50kB produziram valores irregulares devido à baixa resolução do

cliente para o servidor. No final de cada transferência coletamos os dados organizados pela aplicação e com eles montamos a tabela 2-1, que relaciona os tempos de transferência médios de cada uma das quatro aplicações. O intervalo de confiança refere-se a um nível de confiança de 95%.

Tabela 2-1 – Tempo de transferência em milissegundos versus quantidade de bytes enviados em cada transferência.

bytes transferidos	*RMI sobre JRMP*		*RMI sobre IIOP*		*CORBA sobre IIOP*		*TCP/IP*	
	média	int. conf.	**média**	int. conf.	**média**	int. conf.	**média**	int. conf.
100 000	**239,8**	8,67%	**283,4**	25,53%	**259,8**	7,18%	**238,8**	28,02%
200 000	**347,8**	4,32%	**395,4**	3,43%	**400,2**	4,92%	**306,2**	3,31%
300 000	**484,6**	3,80%	**565,0**	2,73%	**559,8**	1,82%	**421,6**	2,81%
400 000	**617,2**	3,59%	**711,6**	3,28%	**715,6**	2,43%	**535,2**	2,18%
500 000	**702,4**	1,58%	**836,4**	2,55%	**835,8**	2,00%	**640,2**	1,52%
600 000	**806,8**	1,36%	**975,6**	1,31%	**977,4**	1,18%	**763,6**	1,51%
700 000	**922,2**	0,94%	**1110,4**	1,59%	**1088,8**	1,12%	**861,4**	1,38%
800 000	**1045,0**	1,46%	**1240,2**	2,00%	**1234,8**	1,07%	**982,4**	1,52%
900 000	**1183,0**	1,32%	**1360,4**	1,72%	**1378,2**	2,51%	**1086,8**	0,89%
1 000 000	**1288,6**	1,13%	**1478,4**	1,30%	**1493,6**	1,65%	**1209,6**	1,12%
2 000 000	**2480,6**	0,57%	**2947,4**	2,06%	**2924,0**	1,60%	**2327,2**	0,95%
3 000 000	**3682,8**	0,64%	**4432,6**	1,35%	**4404,4**	1,48%	**3498,8**	0,88%
4 000 000	**4901,6**	0,90%	**5746,6**	1,60%	**5823,0**	1,42%	**4612,2**	1,11%
5 000 000	**6145,0**	1,48%	**7951,6**	1,21%	**7899,4**	2,04%	**5761,4**	1,48%

Os dados da tabela podem ser melhor visualizados através da figura 2-4. É importante ressaltar que o que foi medido neste experimento foi apenas o *tempo de transferência de dados brutos*, em uma direção, de um cliente para um servidor.

O resultado das medições reflete valores esperados[FARL98], isto é, a transferência via RMI (JRMP) mais eficiente que via CORBA (IIOP)[5]. É natural que usando programação em baixo nível com soquetes TCP se possa obter um desempenho superior, uma vez que o cliente e servidor poderão se comunicar usando um protocolo criado *especificamente* para a tarefa a ser realizada, sem qualquer *overhead* adicional introduzido pelas outras arquiteturas.

---

intervalo de tempo medido (a menor unidade de tempo mensurável é 50ms). Valores superiores a 5MB apresentaram grandes variações devido a atrasos provocados pelo sistema para gerenciar memória e cache de disco (o que às vezes causava o travamento do cliente).

[5] Na seção seguinte discutiremos os motivos que levam a essas diferenças.

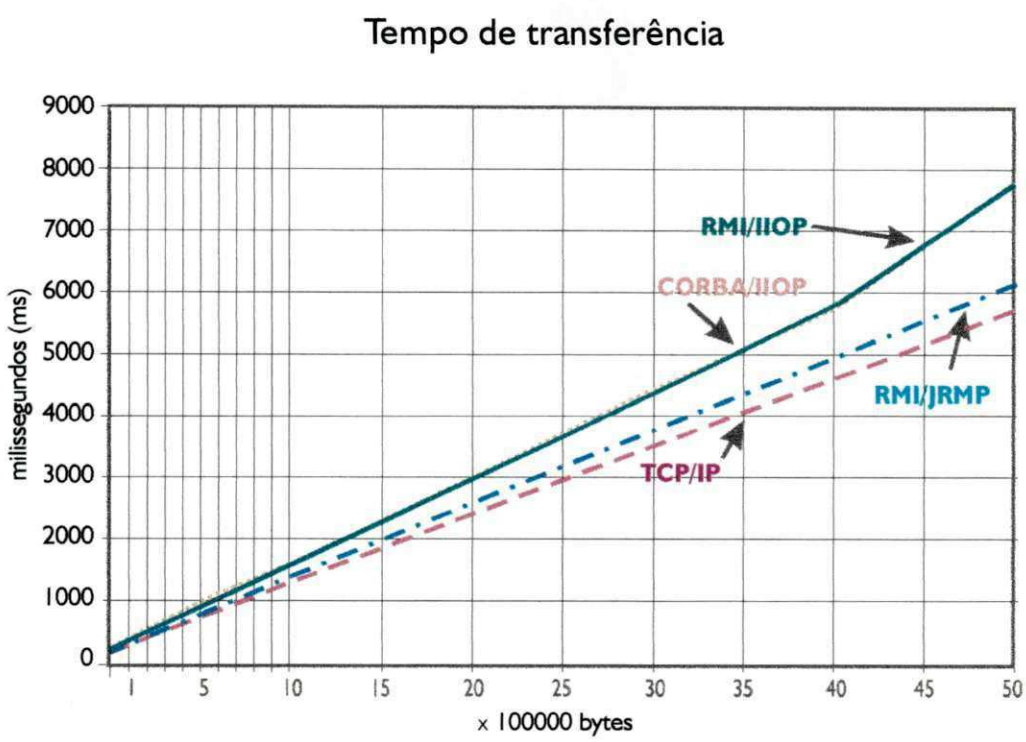


Figura 2-4 - Tempo de transferência em milissegundos ***versus*** quantidade de ***bytes*** enviados em cada transferência.

Não podemos deixar de observar o "empate" entre os tempos de transferência observados na aplicação CORBA e na aplicação RMI/IIOP. Como estamos analisando apenas a transferência de *bytes* (dados brutos), tem maior destaque o efeito devido ao protocolo de transporte, que é o mesmo: IIOP. A medição da taxa de transferência de *informações*, poderia gerar um gráfico diferente, uma vez que iria expor outras diferenças entre as duas tecnologias.

Como sabemos quantos *bytes* foram enviados, e o tempo transcorrido na transferência, podemos calcular a taxa de transferência de *bytes* entre o cliente e o servidor (que é uma medida que tem maior utilidade comparativa que o tempo absoluto). A tabela 2-2 e as figuras 2-5, 2-6 e 2-8 mostram os mesmos dados apresentados anteriormente organizados de acordo com a taxa de transferência de *bytes*. Os valores foram calculados usando a relação:

$$taxa = \frac{bytes\ transferidos}{tempo}.$$

Tabela 2-2 – Taxa de transferência de bytes. Intervalo de confiança com nível de confiança de 95%

bytes transferidos	*RMI sobre JRMP*		*RMI sobre IIOP*		*CORBA sobre IIOP*		*TCP/IP*	
	média	int. conf.	**média**	int. conf.	**média**	int. conf.	**média**	int. conf.
100 000	**417,0142**	8,67%	**352,8582**	25,53%	**384,9115**	7,18%	**418,7605**	28,02%
200 000	**575,0431**	4,32%	**505,8169**	3,43%	**499,7501**	4,92%	**653,1679**	3,31%
300 000	**619,0673**	3,80%	**530,9735**	2,73%	**535,9057**	1,82%	**711,5750**	2,81%
400 000	**648,0881**	3,59%	**562,1135**	3,28%	**558,9715**	2,43%	**747,3842**	2,18%
500 000	**711,8451**	1,58%	**597,8001**	2,55%	**598,2292**	2,00%	**781,0059**	1,52%
600 000	**743,6787**	1,36%	**615,0062**	1,31%	**613,8735**	1,18%	**785,7517**	1,51%
700 000	**759,0544**	0,94%	**630,4035**	1,59%	**642,9096**	1,12%	**812,6306**	1,38%
800 000	**765,5502**	1,46%	**645,0572**	2,00%	**647,8782**	1,07%	**814,3322**	1,52%
900 000	**760,7777**	1,32%	**661,5701**	1,72%	**653,0257**	2,51%	**828,1192**	0,89%
1 000 000	**776,0360**	1,13%	**676,4069**	1,30%	**669,5233**	1,65%	**826,7196**	1,12%
2 000 000	**806,2566**	0,57%	**678,5642**	2,06%	**683,9945**	1,60%	**859,4019**	0,95%
3 000 000	**814,5976**	0,64%	**676,8037**	1,35%	**681,1370**	1,48%	**857,4368**	0,88%
4 000 000	**816,0601**	0,90%	**696,0638**	1,60%	**686,9311**	1,42%	**867,2651**	1,11%
5 000 000	**813,6697**	1,48%	**628,8043**	1,21%	**632,9595**	2,04%	**867,8446**	1,48%

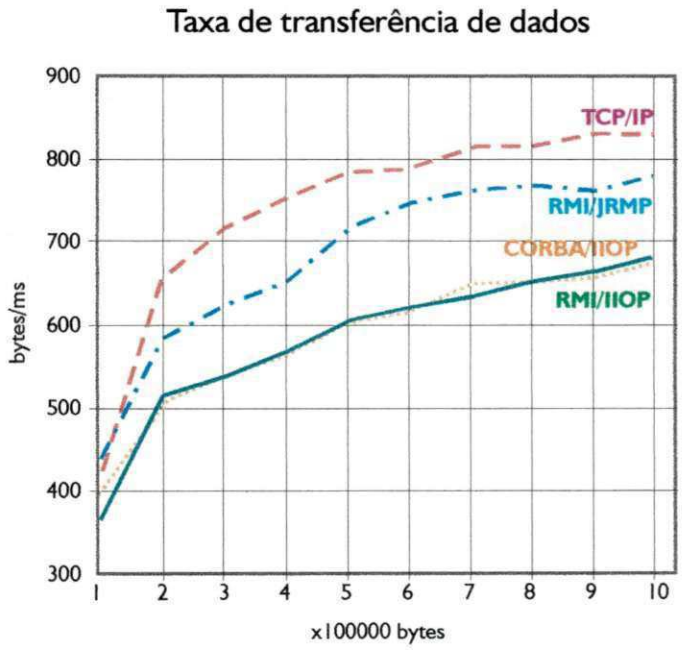


Figura 2-5 – Taxa de transferência de dados para coleções de 0,1 a 1 · 10^6 bytes

Nesta tabela (e gráficos das figuras 2-5 e 2-6), observamos que as taxas de transferência são melhores e tendem a estabilizar-se quando quantidades maiores de *bytes* são transferidos de uma vez. Acreditamos que as quedas de desempenho no final da escala (nas tecnologias IIOP) refletem os efeitos do gerenciamento de memória da máquina utilizada devido à quantidade limitada de memória RAM. A figura 2-7, mostra um gráfico com a alocação de memória no cliente durante o teste usando RMI. Não foi possível obter um gráfico semelhante durante a execução das aplicações CORBA e RMI/IIOP por causa do esgotamento total dos recursos do sistema.

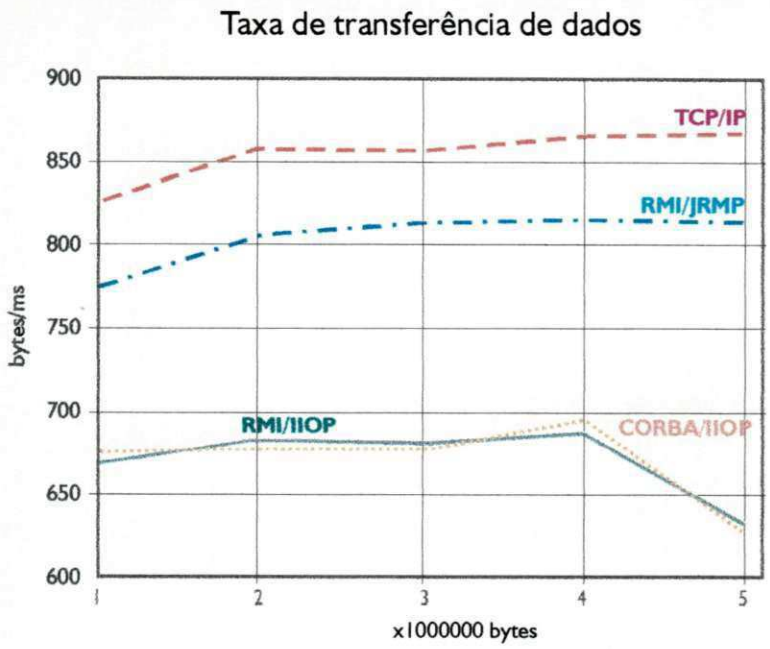


Figura 2-6– Taxa de transferência de dados para coleções de 1 a 5 · 10^6 bytes

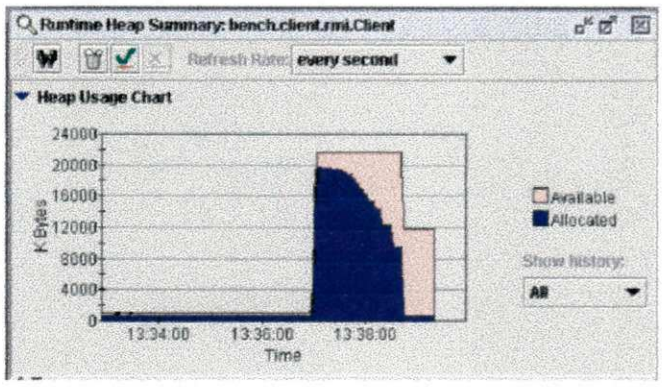


Figura 2-7 – Alocação de espaço na aplicação cliente durante o teste RMI. Gráfico gerado pelo JProbe - "profiler" da KL Group (www.klg.com/jprobe)

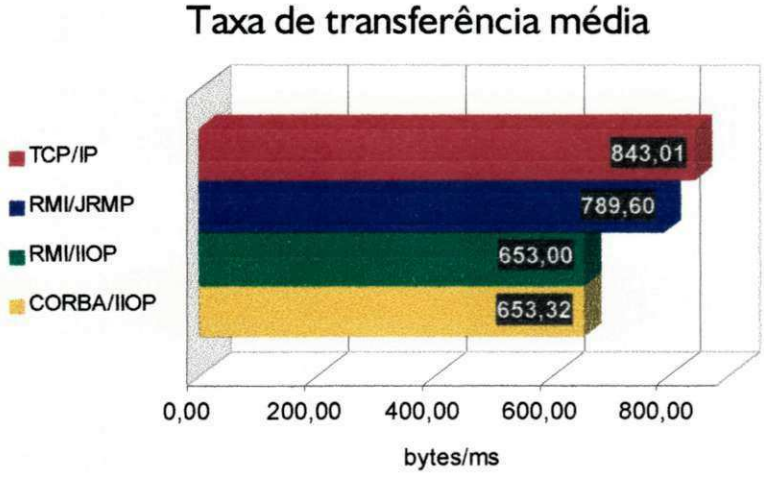


Figura 2-8 – Taxa de transferência média para cada tecnologia testada.

A figura 2-9 ilustra o uso do processador[6] (que está relacionado com a complexidade da aplicação e atividades paralelas do sistema, como gerenciamento de memória) durante a execução das duas aplicações em uma única máquina com 32MB de memória. Pelo gráfico, podemos prever que as aplicações CORBA e RMI/IIOP apresentarão um desempenho inferior causados, principalmente pelo maior consumo de recursos dessas tecnologias, que causa o esgotamento dos recursos do sistema mais rapidamente que as outras.

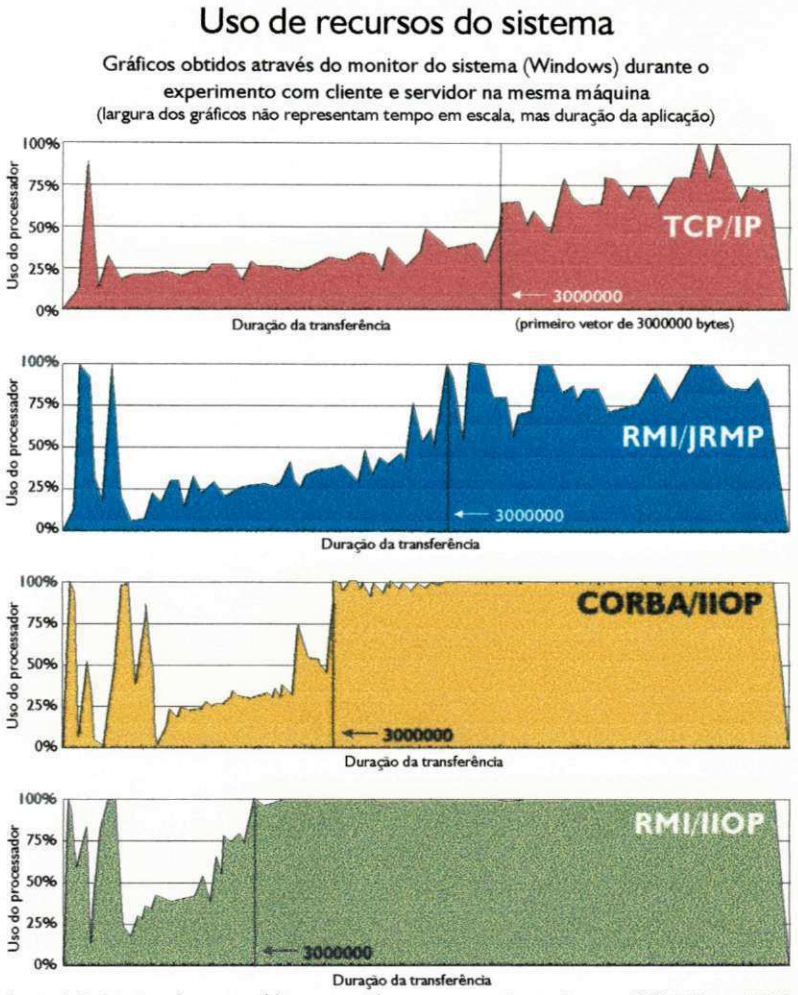


Figura 2-9: Consumo de recursos (cliente e servidor na mesma máquina Pentium MMX 200 com 32MB de memória). A escala reflete a duração da transferência de dados e não uma diferença de tempo.

2.3 Aplicação das tecnologias estudadas em diferentes cenários

Esta seção discute a utilização de aplicações baseadas nas tecnologias estudadas, levando em conta os resultados obtidos e cenários onde cada tecnologia pode ser aplicada.

2.3.1 Objetos remotos (RMI/CORBA) *versus* Sockets TCP/IP

Toda a comunicação em rede TCP/IP entre uma aplicação Java e outra aplicação ou serviço pode ser realizada através do pacote `java.net`. Soquetes (classe `java.net.Socket`) implementam uma conexão confiável usando o protocolo TCP. Datagramas UDP (classe `java`

---

[6] Os gráficos são gerados pelo limitado *Monitor do Sistema* do *Windows95* e se referem a um único experimento de cada par de aplicações. Os valores não são significativos. O objetivo é ilustrar o consumo de recursos apenas.

`.net.DatagramSocket`) implementam conexões mais eficientes, porém não confiáveis. Utilizando essas e outras classes do pacote `java.net`, pode-se implementar qualquer tipo de interoperabilidade entre aplicações TCP/IP. Por que então utilizar tecnologias de objetos remotos, que observamos ser menos eficientes?

Se o objetivo da aplicação é simplesmente transferir comandos ou informações de um lugar para outro, usar o pacote `java.net` pode ser a melhor opção. Usar Java em rede é quase tão simples quanto usar Java para operações locais de entrada e saída. Talvez seja necessário usar *threads*. Uma típica aplicação cliente-servidor terá provavelmente alguns serviços em *threads* diferentes (por exemplo, para que um servidor possa aceitar múltiplos clientes).

Mas se a aplicação precisa criar objetos na aplicação remota ou invocar métodos remotos que retornam objetos, a programação usando `java.net` poderá se tornar bastante complexa. "Os requisitos essenciais em um sistema de objetos distribuídos são a capacidade de criar ou invocar objetos em um processo ou máquina remota, e interagir com eles como se fossem objetos do processo e máquina correntes"[FARL98]. Para criar um objeto remoto, o cliente precisa saber qual a sintaxe usada no servidor para realizar tal tarefa. O servidor também precisará saber como retornar uma informação para o cliente. Portanto, será necessária a existência de algum protocolo de mensagens para "enviar requisições para agentes remotos para que criem novos objetos, para invocar métodos nesses objetos e para eliminar os objetos quando não mais precisarmos deles"[FARL98].

Se a linguagem é Java e um cliente desejar criar um objeto remoto, ele precisará instruir o servidor a carregar a classe correspondente (pelo nome), precisará passar os argumentos do construtor do objeto correspondente, realizar uma operação de criação de objeto (usando `new`) e retornar para o cliente uma referência remota ao objeto criado. Com esta referência, o cliente poderá invocar métodos remotos e quando não precisar mais do objeto, informar ao servidor que o mesmo pode ser destruído.

O servidor também precisará manter um controle sobre os objetos criados remotamente. Se um outro cliente tenta criar um objeto já criado, o servidor deverá retornar-lhe uma referência do objeto existente. Se um dos clientes solicitar a remoção de um objeto, o servidor precisará descobrir se ainda há outras referências remotas para aquele objeto. O servidor precisa manter mais que uma tabela de objetos criados. Precisa também mapear cada um dos métodos daqueles objetos e seus argumentos, que podem, por sua vez, ser também objetos remotos.

É possível desenvolver um mecanismo desses para uma determinada aplicação distribuída usando somente TCP/IP (`java.net`). Ela provavelmente será mais eficiente que uma solução usando tecnologias de objetos distribuídos. O preço pela eficiência será cobrado quando for necessário alterar alguma característica da aplicação. Por utilizar um protocolo proprietário, quase não há reutilização de componentes e é provável que vários procedimentos tenham que ser rescritos. Para, por exemplo, acrescentar um método em um objeto remoto, será necessário mexer em todas as estruturas que usam o objeto remoto em questão, e também naquelas que possuem métodos ou construtores que recebem ou retornam o objeto remoto.

Poderíamos melhorar o protocolo proprietário, acrescentando operações genéricas, independentes dos detalhes da implementação. Isto certamente diminuiria a sua eficiência, mas permitiria um desenvolvimento mais simples, possibilitando talvez uma maior reutilização. Continuando a desenvolver o protocolo dessa maneira, terminaríamos por reinventar um mecanismo como o RMI: mais fácil de desenvolver e manter, porém menos eficiente.

As tecnologias de objetos remotos, como RMI, e CORBA oferecem uma infraestrutura que permite invocações de operações em objetos localizados em máquinas remotas como se fossem locais à aplicação que os usa [VOGE98]. Tanto RMI e CORBA são implementadas em Java usando esses recursos de baixo nível (`java.net`), mas escondem a complexidade tornando o desenvolvimento em rede quase tão simples quanto o desenvolvimento local. O preço é a menor eficiência, uma vez que a tecnologia tem que lidar com uma gama de possibilidades nem sempre presentes na aplicação que as utiliza. Tanto RMI e CORBA geralmente requerem uma infraestrutura adicional, externa à aplicação, como sistemas de nomes e gerentes de objetos, para manter o controle sobre os objetos remotos. Essas estruturas, além de ocuparem mais recursos e memória do sistema, impõem obstáculos adicionais que os dados precisam transpor, antes de serem enviados pela rede através de conexões TCP ou UDP.

Portanto, embora o uso direto de TCP/IP em aplicações Java permita obter o melhor desempenho, a complexidade da aplicação e a necessidade de manutenção futura podem induzir à opção por uma tecnologia de objetos distribuídos, como RMI.

2.3.2 RMI/JRMP versus RMI/IIOP e CORBA

A tecnologia RMI, nativa do Java 2, utiliza para a comunicação um protocolo chamado *Java Remote Method Protocol* (JRMP), que permite a criação de objetos remotos e a chamada de métodos remotos, passando parâmetros e retornando valores e referências de forma transpa-

rente ao programador. Através do RMI, o programador invoca métodos remotos da forma como invoca métodos locais. A comunicação entre os clientes e os objetos remotos é centralizada no *RMI Registry* – programa rodando no servidor que atua tanto como servidor de nomes como gerente de objetos Java [SUN96]. O *RMI Registry* roda sobre a máquina virtual Java local e permite que ela se comunique com uma máquina virtual remota.

O suporte à tecnologia CORBA foi introduzido na plataforma Java 2 através de um conjunto de pacotes de prefixo `org.omg`. Embora parecidos na superfície, RMI e CORBA são bastante diferentes quando sua estrutura interna é analisada. RMI é uma tecnologia nativa a Java e é, em essência, uma extensão ao núcleo da linguagem. CORBA é uma tecnologia de *integração*, independente de Java. Seu objetivo é unir tecnologias de *programação* incompatíveis entre si [CURT97]. A comunicação em CORBA não ocorre diretamente entre o cliente e o servidor, ou entre máquinas virtuais, mas entre ORBs (*Object Request Brokers*) – gerentes de objetos presentes nas máquinas que possuem clientes ou servidores. Em redes TCP/IP, CORBA geralmente utiliza o protocolo IIOP – *Internet Inter-ORB Protocol*, que é responsável pela comunicação entre ORBs. As comunicações não são centralizadas (como são no *RMI Registry*) e os objetos podem estar distribuídos pela rede [FARL97][OMG98].

O "denominador comum" nas comunicações JRMP é uma interface escrita em Java. Nas comunicações IIOP, o denominador comum é uma interface IDL (neutra com respeito à linguagem), que permite a possibilidade de comunicação com objetos escritos em outras linguagens.

Ainda em versão beta (como extensão do Java 2), a nova versão de RMI suporta a comunicação através de IIOP – *Internet Inter-ORB Protocol*, que é o protocolo utilizado na comunicação entre objetos remotos CORBA. Na prática o novo RMI *é* CORBA pois só se parece com RMI na superfície. Usa a tecnologia de *programação* Java, mas está de acordo com a tecnologia de *integração* CORBA. O programador pode desenvolver em Java, como já fazia com o RMI antigo. Pode usar suas classes para criar objetos remotos que serão utilizados via JRMP ou IIOP. O pacote contém ainda ferramentas que geram IDL a partir de Java, facilitando a comunicação entre objetos de linguagens diferentes. Esta nova solução permite que programas Java escritos em RMI possam se comunicar com objetos remotos CORBA e vice-versa.

Agora com três opções de objetos distribuídos, qual delas devemos usar? RMI sobre JRMP, CORBA ou RMI sobre IIOP? Novamente, não temos a intenção de apontar esta ou aquela tecnologia como a melhor. RMI (JRMP) é uma alternativa viável para um conjunto de

aplicações. CORBA (incluindo RMI sobre IIOP), é uma tecnologia apropriada para outro conjunto de aplicações. Existem aplicações que podem ser desenvolvidas usando qualquer uma das duas tecnologias, mas há muito mais aplicações onde uma das duas possui vantagens distintas sobre a outra [CURT97].

Usar RMI/JRMP em vez de CORBA/IIOP é o ideal quando todas as partes da aplicação distribuída são escritas em Java. O desempenho da aplicação provavelmente será melhor (com base nos resultados dos experimentos que realizamos) e é possível utilizar recursos da linguagem Java que não são disponíveis via CORBA, como carregamento de classes remotas (para serem instanciadas localmente) e objetos serializados [FARL98]. Além do desempenho, uma vantagem do RMI/JRMP sobre CORBA/IIOP é o desenvolvimento mais simples. Isto, antes do advento do RMI sobre IIOP:

- Com RMI os tipos dos objetos são preservados. É preciso realizar conversões adicionais em CORBA. Usando RMI/IIOP as conversões são feitas de forma transparente.
- RMI é Java. Não é necessário aprender uma nova linguagem. É preciso usar IDL para definir as interfaces dos objetos remotos em CORBA. Usando RMI/IIOP o IDL é gerado automaticamente, não sendo, portanto, necessário conhecer a linguagem.
- Uma única linha de código é necessária para registrar ou localizar um objeto no *RMI Registry*. CORBA possui um serviço de registro de nomes mais sofisticado que exige a obtenção de várias referências para registrar até mesmo um único objeto. RMI sobre IIOP exige um esforço menor que CORBA, mas ainda bem maior que RMI sobre JRMP (o registro de objetos é a "parte CORBA" da tecnologia).

Usar CORBA é a melhor opção quando partes da aplicação estão escritas em outra linguagem. RMI não suporta a comunicação com objetos que não sejam objetos Java. Para aplicações totalmente escritas em Java, observamos uma taxa de transferência menor nas aplicações CORBA. Muitas transformações extras são necessárias para atingir o "denominador comum" IDL, que uma interface Java. Mas uma solução CORBA pode melhorar o desempenho de uma aplicação Java utilizando objetos escritos em linguagens mais eficientes (como C) para tarefas críticas. Objetos especificados em IDL podem ser substituídos por quaisquer outros escritos em outras linguagens sem que o restante da aplicação seja afetada [VOGE 98].

Há ainda, vantagens que podem levar à escolha de CORBA/IIOP sobre RMI/JRMP mesmo em ambientes somente Java:

- Os clientes RMI precisam saber *em que máquina* estão os objetos. O *RMI Registry* deve estar executando na máquina onde estarão os objetos remotos. Os objetos CORBA, por sua vez, podem estar em *qualquer lugar* da rede e o cliente não precisa saber da sua localização. A comunicação em redes TCP/IP é realizada entre ORBs de qualquer plataforma ou linguagem usando o protocolo IIOP.

- Enquanto RMI/JRMP suporta apenas nomes de objetos definidos na mesma raiz, o servidor de nomes Java usado em aplicações CORBA suporta a organização hierárquica de nomes de objetos, dentro de contextos que se assemelham a diretórios em um sistema de arquivos, evitando o risco de conflito de nomes. Isto é útil quando várias aplicações usam o mesmo registro de nomes ou registram muitos objetos.

Se uma aplicação Java distribuída não pretende interagir com código em outras linguagens e se a localização dos objetos em um local específico não for um problema, RMI pode ser a solução ideal de objetos remotos para essa aplicação. Mas é preciso ter cuidado, pois RMI é parte de Java e não uma tecnologia de integração independente. No futuro, certamente surgirão novas linguagens e os sistemas Java serão os próximos sistemas legados que exigirão integração com os novos sistemas, e RMI não será capaz de oferecê-la. CORBA, por ser uma tecnologia de integração independente de linguagem de programação é uma melhor defesa contra a obsolescência [CURT97].

Mas com a opção de usar tanto JRMP como IIOP (ainda em beta), RMI pode vir a ser a melhor opção para a implementação de objetos distribuídos usando Java, oferecendo ao mesmo tempo facilidade de desenvolvimento e portabilidade com independência de plataforma e linguagem, além de proteção contra a obsolescência futura. A troca de JRMP por IIOP é direta. A implementação dos objetos remotos só precisam mudar uma linha de código e todo o restante do código é aproveitado, sem alterações. A única alteração maior ocorre no servidor que irá registrar os objetos remotos, já que não mais usará o *RMI Registry*, mas um ORB.

2.4 Conclusão

Este capítulo contribuiu com uma discussão acerca da aplicabilidade de tecnologias de objetos distribuídos e apresentou um experimento onde pôde-se comparar taxas de transferência de aplicações usando tecnologias diferentes. Várias tecnologias estudadas são novas e têm sido objeto de poucos estudos. Os resultados obtidos, embora limitados e insuficientes para

fornecer conclusões gerais, servem, junto com as aplicações desenvolvidas, de ponto de partida para estudos posteriores.

Os resultados do capítulo também oferecem subsídios para nortear a escolha de tecnologias de objetos distribuídos. Conhecendo a taxa de transferência relativa entre diferentes tecnologias de rede, o programador Java terá mais um parâmetro que poderá influir na sua escolha. A aplicabilidade à tarefa desejada e a necessidade de integração presente ou futura, que também foram discutidos, são outros fatores que podem predominar nessa escolha.

Os resultados são limitados. Variações nas configurações dos sistemas envolvidos (principalmente memória e cache de disco), podem alterar de forma significativa os resultados dos experimentos. Peculiaridades das implementações das tecnologias utilizadas em plataformas diferentes de *Windows 95* também podem influenciar bastante os resultados.

As limitações devido ao gargalo da rede (10Mbps) não permitem que se tire conclusões definitivas quanto ao desempenho entre as diferentes aplicações. Não é possível afirmar, por exemplo, *o quanto* RMI/IIOP é pior que RMI/JRMP. Para atenuar o problema da rede seria necessário realizar os experimentos em uma rede veloz ou em uma mesma máquina. Realizando-o na mesma máquina, outros gargalos (gerenciamento de memória) serão expostos.

Os experimentos apresentados são baseados em aplicações escritas em Java que podem ser executadas em outras plataformas. O usuário poderá, portanto, repetir os experimentos na plataforma onde pretende desenvolver suas aplicações, e ter resultados mais confiáveis.

Embora seja uma medida importante, que de fato reflete um aspecto de cada tecnologia, a taxa de transferência de dados não é suficiente para caracterizar de forma *definitiva* o desempenho relativo entre tecnologias de objetos distribuídos. Para uma análise mais completa seria preciso levar em conta também fatores como a transferência de *informação* (transferência de dados mais a decodificação dos dados em informação útil) e o tempo de uma transação completa (RTT – *round trip time*), que não foram medidos pelas aplicações utilizadas. Um estudo mais abrangente poderá ser realizado em um trabalho futuro.

Capítulo 3
Comparação entre aplicações Web usando CGI e servlets

Este capítulo discute e compara, quanto à aplicabilidade e aspectos de desempenho, duas tecnologias abertas que são usadas para estender as funções básicas de servidores HTTP: CGI – *Common Gateway Interface* e Java *Servlet* API. Existem várias outras tecnologias de extensão para servidores HTTP. A grande maioria, porém, é proprietária e tem seu uso restrito a determinadas plataformas e fabricantes. As duas aqui analisadas são suportadas em diversos servidores, de fabricantes e plataformas diversas.

3.1 Introdução: Aplicações Web

Aplicações Web são aplicações cujo ambiente de execução é a Web. Suas funções e recursos podem estar distribuídos por várias localidades, e disponibilizados através de um ou mais *servidores* HTTP. A interface do usuário é construída por um browser (ou outro *cliente* HTTP), após interpretar uma página escrita em HTML[1], fornecida pelo servidor. HTML não possui recursos de programação nem recursos de apresentação visual. A linguagem apenas fornece instruções de *marcação* de texto que são utilizadas pelo browser para:

- formatar uma página de informação (usando uma determinada *folha de estilos*, geralmente definida pelo browser), com texto, tabelas e componentes de formulário como botões, caixas de texto, etc.
- vincular imagens à página, formatando-as juntamente com o texto (se possível),
- vincular recursos multimídia como som, vídeo, applets e *plug-ins*, e
- habilitar eventos em vínculos de hipertexto e botões para permitir que o browser carregue outras páginas ou recursos cujos endereços estão embutidos no código.

---

[1] *HyperText Markup Language* – linguagem padrão utilizada para estruturar páginas interligadas por hipertexto.

Com HTML *apenas*, a única aplicação Web que se pode construir é aquela que permite a navegação dentro de um banco de informações em hipertexto. Esta aplicação é básica e fornecida por qualquer browser.

Para ir além da navegação, é preciso estender as capacidades básicas do HTML através de programas de roteiro (*scripts*) ou componentes (*plug-ins*). Esses recursos não estendem o *protocolo* HTTP mas podem estender as capacidades de um cliente ou servidor. Desta forma, as extensões podem ser classificadas em relação à sua localização dentro da arquitetura cliente-servidor. Recursos *lado-cliente* executam no browser, e só dependem do browser para funcionar. Recursos *lado-servidor* executam no servidor e só precisam do suporte do servidor. Um browser, porém, pode iniciar uma aplicação que executará no servidor e o servidor poderá enviar páginas ou componentes que só serão interpretados ou executados no browser. Freqüentemente, uma aplicação Web utiliza tanto recursos *lado-cliente* quanto *lado-servidor*.

3.1.1 Aplicações Web com recursos *lado-cliente*

Existem dois tipos de extensões *lado-cliente*: os *componentes*, que funcionam como extensões *executadas* como se fossem parte do browser; e os *scripts* (roteiros), que são *interpretados* pelo browser juntamente com o HTML.

Os componentes, como os Java *Applets*, controles *ActiveX* ou *plug-ins*, podem acrescentar novos recursos ao browser, permitir que o mesmo suporte outros formatos de multimídia, ou realizar tarefas bem específicas. São aplicações completas e geralmente interagem pouco com a página HTML, utilizando-a somente como contexto gráfico para exibição da sua própria interface. Componentes são objetos externos à página, e, como qualquer objeto independente do texto, são carregados através de uma requisição à parte (como é feito com as imagens) [2].

Os *scripts* estendem a linguagem HTML. Geralmente são embutidos dentro do próprio código HTML. São interpretados enquanto o browser carrega a página. O próprio código HTML é um *script* que é interpretado pelo browser para definir a estrutura da página. Um possível bloco CSS (*Cascading Style Sheets*) embutido no HTML é outro *script* que estende o

---

[2] O browser se comunica com o servidor através de *requisições* HTTP, que consistem de métodos (GET, POST, HEAD) seguidos de uma URL informando o recurso desejado. A *resposta* do servidor, consiste essencialmente de um código de status (200, 404, 500) e um conjunto de cabeçalhos (formato RFC822) contendo informações sobre o tamanho e tipo MIME [RFC2045] dos dados retornados, que seguem o cabeçalho. HTTP não mantém uma sessão aberta. Cada imagem de uma página provoca uma nova requisição ao servidor. [RFC2068]

código HTML para definir a apresentação e *layout* de uma página. Estruturas de programação podem ser embutidas em uma página usando JavaScript, que introduz no HTML a capacidade de manipular eventos, realizar controle de fluxo, suporte a operações matemáticas, acesso a variáveis do browser entre outras possibilidades. JavaScript não é um padrão da W3C, mas é suportado pela recomendação ECMA-262[3].

Enquanto as tecnologias *lado-cliente* são ótimas para realizar operações locais como validação, geração de gráficos, etc. não servem para a maior parte das operações que exigem persistência de dados. Operações de acesso a rede que utilizam disco local geralmente são desabilitadas ou bastante restritas no cliente, por questões de segurança. Também podem ser pouco eficientes. Nesses casos, a melhor alternativa é apelar às tecnologias *lado-servidor.*

3.1.2 Aplicações Web com recursos *lado-servidor*

Existem várias arquiteturas diferentes que implementam suporte a extensões em servidores Web. A mais popular é a tecnologia CGI – *Common Gateway Protocol* [GUND96], que fornece uma especificação que permite o desenvolvimento de aplicações *gateway* que servem como ponte para que o browser possa realizar tarefas no servidor. CGI não é linguagem. É apenas uma especificação que pode ser implementada usando *qualquer* linguagem. Aplicações CGI podem ter sua execução solicitada por uma requisição do browser e podem servir de ponte para qualquer aplicação ou dados localizados no servidor. A figura 3-1 ilustra duas requisições e respostas HTTP. A segunda requisição usa CGI.

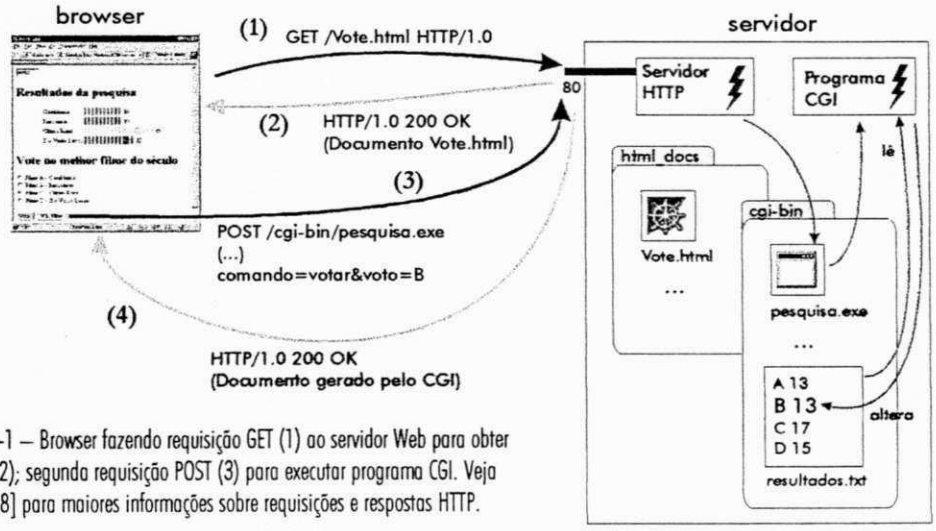


Figura 3-1 – Browser fazendo requisição GET (1) ao servidor Web para obter página (2); segunda requisição POST (3) para executar programa CGI. Veja [RFC2068] para maiores informações sobre requisições e respostas HTTP.

[3] ECMA é um consórcio europeu que define padrões abertos para comunicações. URL: http://www.ecma.org.

O preço da portabilidade do CGI está no baixo desempenho [EDWA99]. Um programa CGI não faz parte do servidor. É um processo externo, separado do servidor, portanto, não reutiliza recursos alocados ao mesmo. Essa e outras limitações têm incentivado o mercado a procurar alternativas a esta tecnologia, mais fortemente integradas ao servidor. O resultado foi um coquetel de soluções incompatíveis: desde APIs que permitem o desenvolvimento de componentes que interagem diretamente com as funções do servidor (ISAPI, NSAPI[4]), até rotinas (*scripts*) embutidas em páginas HTML para serem interpretadas pelo servidor antes de uma resposta (ASP, JSP, LiveWire, Cold Fusion[4]). A maior parte das soluções é pouco ou nada portátil. Entre as que oferecem maior portabilidade, através de uma API independente de servidor e linguagem de programação, está a Java Servlet API [SUN982], que permite que aplicações em Java possam ser usadas internamente por qualquer servidor. Por ser Java, roda em qualquer plataforma onde há uma máquina virtual Java. Por ser uma tecnologia que não depende do servidor, roda, através de um *plug-in* (quando não está embutida no servidor) em todos os servidores mais populares.

3.1.3 CGI vs. Servlets

Neste trabalho desenvolvemos aplicações Web usando a Java Servlet API e CGI. São aplicações simples de funcionamento equivalente, adequadas para que possamos comparar as duas tecnologias quanto ao desempenho relativo a tempos de resposta. As duas tecnologias, porém, possuem arquiteturas diferentes, o que nos permite prever, antecipadamente, os resultados.

CGI é uma especificação que determina parâmetros para que uma aplicação *externa* possa ser executada a partir de uma requisição enviada ao servidor Web [GUND96]. A figura 3-2 (a) ilustra um servidor processando uma aplicação através de CGI. Cada requisição do browser (usando o método POST, neste exemplo), provoca a execução de um novo processo na máquina servidora. Este novo processo é externo ao servidor Web (aplicação), não havendo compartilhamento de recursos alocados pelo sistema entre eles. Se houver 5 requisições em processamento pelo servidor, haverá 5 processos extras no sistema, além do processo do próprio servidor Web.

---

[4] Marcas registradas.

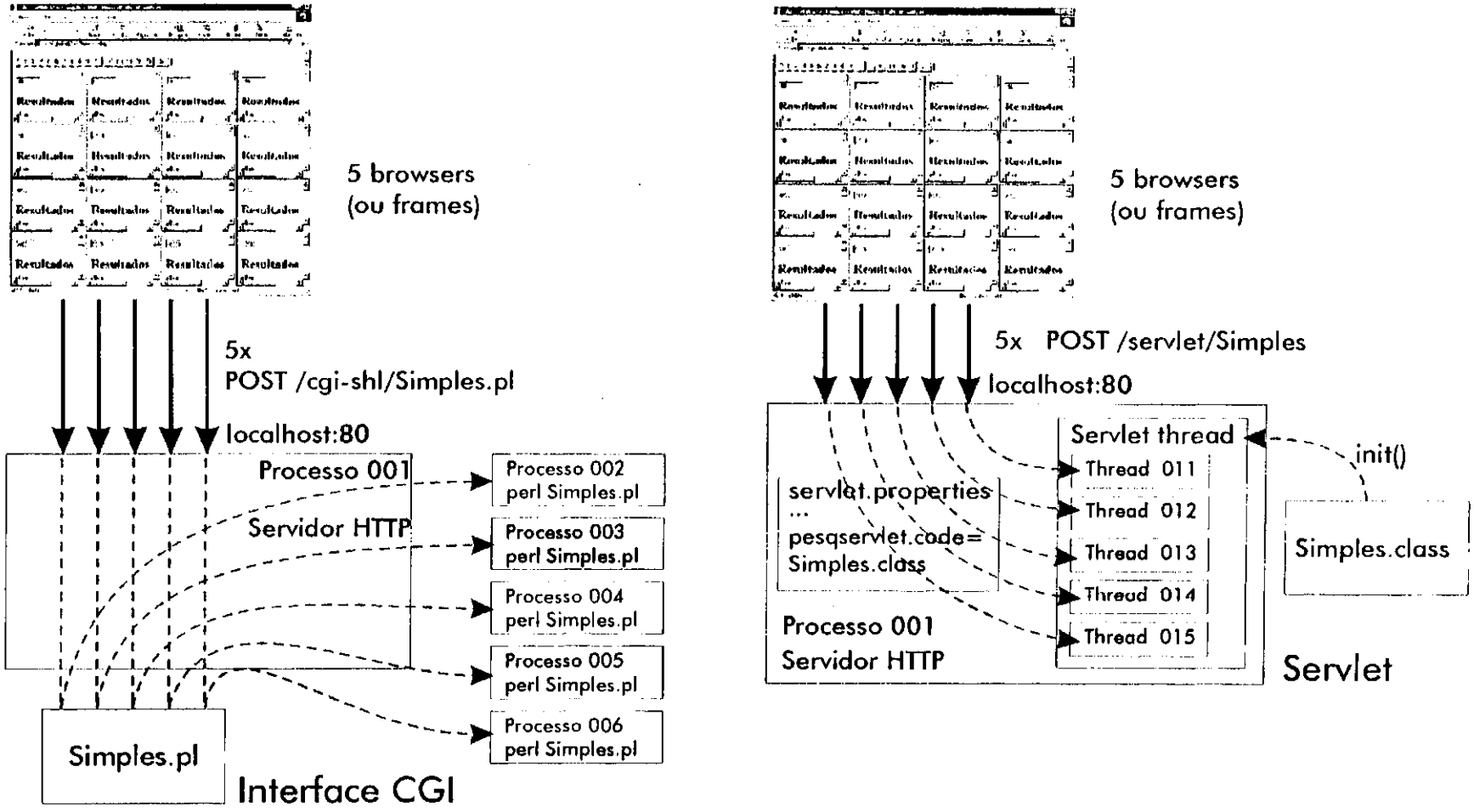


Figura 3-2 – (a) Figura do lado esquerdo: servidor Web processando requisição para executar 5 programas CGI. (b) Figura do lado direito: servidor processando requisição para executar 5 servlets.

Servlets são programas em Java construídos a partir de uma API específica implementada em servidores Web compatíveis. Uma requisição direcionada a um servlet é executada pelo próprio servidor, e não por um programa externo. A figura 3-2 (b) mostra o comportamento de um servidor usando servlets. O programa que contém o servlet é carregado pelo servidor, possivelmente na sua inicialização, e passa a fazer parte do mesmo. Quando o servidor recebe uma requisição destinada a um servlet, um novo *thread* é iniciado. No exemplo citado anteriormente, se usássemos servlets não teríamos 5 processos mas 5 *threads* rodando como parte do mesmo processo que corresponde ao servidor Web. *Threads* consomem recursos assim como processos, mas consomem bem menos. Sendo *threads*, servlets poderão compartilhar dados, utilizar os mesmos recursos e, enfim, realizar a mesma coisa com um desempenho e eficiência bem melhores que programas CGI em processos separados.

Apesar dos argumentos que apontam em favor dos servlets, mencionados em várias publicações, encontramos poucas informações que nos oferecessem medidas do seu desempenho em relação a CGI. Certamente que a criação de um processo no sistema operacional influenciará negativamente sobre o desempenho de uma aplicação. Mas será que essa diferença é tão significativa a ponto de tornar um servlet escrito em uma linguagem interpretada como Java mais eficiente que um programa CGI escrito em uma linguagem eficiente como C?

3.2 Experimento para comparar o desempenho de CGI e *servlets*

Para responder à pergunta da última seção, decidimos realizar um experimento com o objetivo de medir o desempenho de aplicações Web usando servlets e usando CGI em um servidor Web. O experimento realizará a execução de três conjuntos de aplicações como extensões do servidor. As aplicações refletem níveis diferentes de utilização dos recursos da máquina servidora. Elas precisam ser instaladas em um servidor Web com suporte a servlets e CGI.

As aplicações estão localizadas no subdiretório `/jad/apps4/` (resultante da expansão dos arquivos do disquete que acompanha esta dissertação). Elas precisam ser instaladas de acordo com o mecanismo de instalação do servidor utilizado. Os programas foram desenvolvidos em *Perl*, *C* e *Java*. Podem ser localizados, respectivamente, nos subdiretórios `/jad/apps4/cgipl/`, `/jad/apps4/cgic/` e `/jad/apps4/cgijava/`. Os arquivos com o código fonte são:

1. `Simple.pl`, `Simple.c` e `Simple.java` – aplicação simples que lê um arquivo do disco e o imprime na tela. Aplicação rápida – usa poucos recursos (memória e processador) do sistema e é limitada pela capacidade dos mecanismos de entrada e saída (*I/O bound*).
2. `Combina.pl`, `Combina.c` e `Combina.java` – aplicação que realiza a combinação de 100 números através de uma função fatorial recursiva. Aplicação lenta – usa muitos recursos do sistema e é limitada pela capacidade da CPU (*CPU bound*).
3. `CombinaMax.java` e `CombinaMax.c` – aplicação que faz a mesma coisa que `Combina`, mas utiliza 120 números e repete o processo 20 vezes. Aplicação muito lenta.

Com esses três tipos diferentes de aplicação, pretendemos demonstrar o comportamento dos servlets Java em relação a CGI em três cenários típicos. Comparações envolvendo linguagens de programação diferentes costumam ter pouca validade [KERN 98] devido à dificuldade de se encontrar estruturas realmente equivalentes entre linguagens. Nós mantivemos programas simples o suficiente para que se possa ter um controle razoável quanto à sua equivalência. As aplicações CGI são escritas em duas linguagens:

- Perl – uma das mais eficientes linguagens interpretadas. É a linguagem mais usada no desenvolvimento de aplicações CGI [WALL91].
- C – uma linguagem compilada, e veloz (mais eficiente que Java). É usada em muitas aplicações CGI onde o desempenho é crítico [KERN78].

Java [SUN98] é usada apenas nas aplicações que executarão usando a Java Servlet API. Com três cenários diferentes, poderemos demonstrar não só a influência da arquitetura CGI sobre as aplicações, mas também a influência do desempenho de cada linguagem nos resultados medidos pelas aplicações.

Podemos prever que as arquiteturas CGI e servlet introduzirão algum *overhead* sobre o tempo que as aplicações levariam para executar se fossem aplicações independentes. Criamos versões dos servlets e programas CGI que podem ser executados como aplicações do *MS-DOS* através de linha de comando. Essas aplicações imprimem o tempo que levaram para executar a mesma tarefa cujo tempo de execução medimos via Web. Com esses dados, poderemos calcular o tamanho do *overhead* introduzido pelo servidor. Essas aplicações estão no

subdiretório `/jad/apps4/` e contém o prefixo `App`, seguido do nome do programa Web que representam.

3.2.1 Funcionamento e descrição das aplicações

As aplicações têm todas a mesma interface inicial, e praticamente a mesma aparência depois de iniciadas. A interface do usuário consiste de uma página HTML (que deve ser carregada por um browser) que contém apenas uma caixa de texto e um botão. Apertando o botão, o browser fará uma requisição ao servidor para que ele execute uma aplicação. Quando terminar, o servidor retornará uma página ao browser que conterá uma nova caixa de textos com a diferença entre o *tempo que foi feita a requisição* e o *tempo da resposta*. O tempo é registrado em milissegundos.

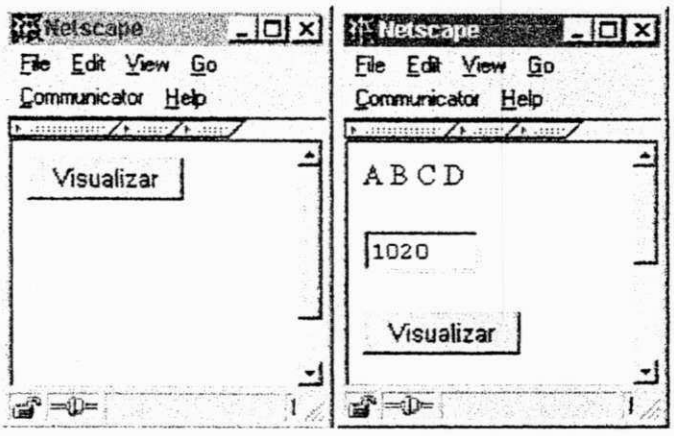


Figura 3-3 – Interface das aplicações.

As páginas HTML geradas pelas aplicações "CGI em Perl", "CGI em C" e pelo servlet são praticamente idênticas. A única diferença está na URL passada ao formulário de pesquisa dentro do código HTML (no bloco `<form> ... </form>`), que endereça programas diferentes. Esta diferença não é perceptível ao usuário que notará apenas as diferenças no tempo de resposta.

Para registrar o tempo que leva para que o cliente seja atendido depois de apertar o botão, fizemos com que o evento provocado pelo clique do botão gravasse um *cookie*[5] no cliente (usando JavaScript) contendo uma linha de texto com o instante em que o botão foi apertado. Logo em seguida, o servlet ou CGI é requisitado para realizar a tarefa solicitada. Assim que o

---

[5] Cookies [RFC2109] são pequenas quantidades de informação na forma de pares nome-valor que podem ser armazenadas na máquina do cliente ou lidas por solicitação de um programa fornecido pelo servidor ou instrução HTML, JavaScript ou HTTP. A forma de armazenamento depende do fabricante do browser. Em geral, ocupam um arquivo ou subdiretório e têm sua população limitada a 20 por domínio.

Os cookies possuem um espaço de nomes restrito a uma determinada instalação de browser, domínio do servidor, caminho e tempo de vida. Ou seja, não é possível ler um cookie com outro browser, após o término de sua vida ou usando um caminho ou domínio diferente do que foi utilizado para gravá-lo. Dentro do seu espaço de nomes, o nome de um cookie é único, podendo ser sobreposto, eliminado ou lido.

programa terminar a sua tarefa, ele retorna uma página HTML que será interpretada pelo browser. No início da página, uma outra instrução JavaScript lê o *cookie* gravado anteriormente e calcula a diferença entre o instante de tempo armazenado e o instante de tempo atual, que corresponde ao tempo transcorrido entre a requisição do browser e a resposta do servidor (*Round Trip Time – RTT*).

3.2.2 Realização do experimento

Esta seção descreve em detalhes a realização dos experimentos (para que possam ser repetidos) e a plataforma onde os testes foram realizados. Os resultados das medições obtidas são apresentados na seção seguinte.

Todos os experimentos foram realizados em uma máquina *Pentium* 200 MMX com 32MB de memória rodando *Windows95* (OSR2). Utilizamos o servidor HTTP *Netscape Fast-Track Server 2.0* com suporte a servlets através do módulo *Live Software JRun*. Os servlets executaram sobre a plataforma Java 2, instalada localmente. Os programas em Perl utilizaram o interpretador *GNU Perl 5 build 110* para *Windows95*.

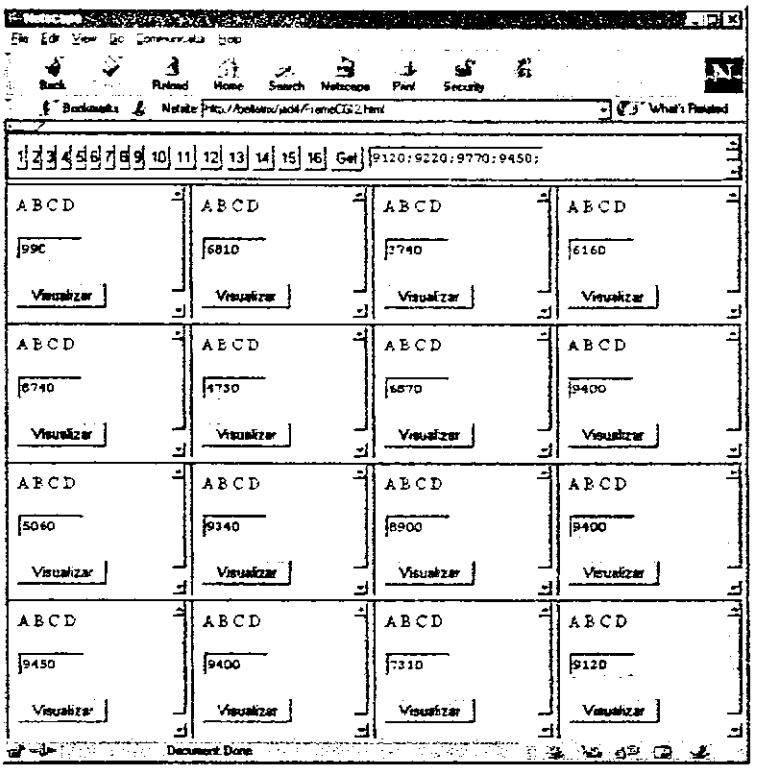


Figura 3-4 – *Frame* de controle (no alto da página) e *frames* de testes.

Os experimentos foram todos executados da mesma forma (a interface do usuário é uniforme). Foram realizados tendo 1 a 15 clientes fazendo simultaneamente requisições ao servidor. Para simular múltiplos clientes em uma mesma máquina, utilizamos uma página com *frames* HTML[6] (veja a figura 3-4) para dividir a janela do browser em janelas menores. Em cada *frame* há uma página (que possui um botão) que fará

[6] Frames são janelas que possuem todas as características de janelas do browser (podem ter documentos, nomes, etc.) menos a possibilidade de flutuar sobre as outras.

uma requisição ao CGI ou servlet. Criamos um *frame* adicional de controle que, através de instruções JavaScript, pode realizar as requisições de todas as janelas quase ao mesmo tempo.

O instante de tempo em que o último cliente fez sua requisição permanece armazenado no *cookie*. À medida em que cada cliente recebe a página de resposta, exibe o instante entre o *último cookie* definido (última requisição realizada) e o momento de recebimento da resposta (já que os outros "clientes", por estarem no mesmo browser, sobrepõem o *cookie* ao fazer suas requisições). O maior valor entre os clientes que fizeram a requisição contém o tempo que o último cliente levou para receber sua página. A página de controle possui um botão ("Get") que recupera este valor.

A instalação do CGI é dependente de servidor. No servidor *FastTrack*, há três tipos de configuração CGI: para aplicações *MS-DOS* (CGI), para aplicações *Windows* (Windows CGI) e para aplicações que necessitam de interpretador (Shell CGI) como os programas em Perl. A configuração consiste em definir URLs que serão usadas para identificar o tipo de CGI usado e mapear essas URLs a caminhos absolutos na máquina onde roda o servidor. Definimos os seguintes mapeamentos[7]:

- "CGI Shell" (programas em Perl): URL `/cgi-shl/` Caminho: `c:\jad\apps4\cgipl\`
- "CGI" (programas em C): URL: `/cgi-bin/` Caminho: `c:\jad\apps\cgic\`

Para instalar os servlets, copiamos os arquivos compilados (`*.class`), localizados no subdiretório `/jad/apps4/cgijava/` para o subdiretório `/servlets/` da aplicação *JRun*.

Criamos ainda um diretório de documentos adicional, para disponibilizar via servidor as páginas HTML usadas no teste. Elas estão no subdiretório `/jad/apps4/htdocs`, que vinculamos à URL `/jad4/`.

Estando os serviços instalados no servidor, podemos utilizá-los a partir do cliente. As seguintes URLs podem ser usadas para carregar a página de *frames* (que simula 16 clientes):

- Aplicação `Simples`: `http://nome_do_servidor/jad4/frameC.html`
  `http://nome_do_servidor/jad4/framePerl.html`
  `http://nome_do_servidor/jad4/frameJava.html`

---

[7] Esta é a configuração *default* dos programas e documentos HTML. Para repetir o experimento em outra máquina e outro servidor Web, é preciso alterar essas configurações de acordo com o novo ambiente.

- Aplicação `Combina`: `http://nome_do_servidor/jad4/frameC2.html`
  `http://nome_do_servidor/jad4/framePerl2.html`
  `http://nome_do_servidor/jad4/frameJava2.html`
- Aplicação `CombinaMax`: `http://nome_do_servidor/jad4/frameC3.html`
  `http://nome_do_servidor/jad4/frameJava3.html`

Para iniciar a simulação, depois de carregar uma das URLs acima, apertamos o botão (do *frame* de controle) correspondente ao número de clientes desejado (1 a 15). Em pouco tempo, o servidor começa a enviar as respostas. Como a página HTML não grava *cookie*, o primeiro teste não apresentou valor. Os testes seguintes retornaram valores úteis. Quando todas as respostas foram enviadas pelo servidor, obtivemos o maior tempo de acesso apertando o botão "Get" no *frame* de controle. Repetimos o experimento diversas vezes[8] para cada uma das 8 aplicações e anotamos os resultados.

Terminados os experimentos com o servidor, realizamos testes com aplicações rodando em *MS-DOS*. São as mesmas aplicações usadas via servidor Web modificadas somente para acrescentar instruções que permitissem medir o tempo de execução e imprimi-lo em linha de comando. As aplicações Java receberam também um método `main()` para torná-las executáveis. As aplicações são `AppCombina`, `AppSimples` e `AppCombinaMax`, disponíveis em Perl, como executáveis MS-DOS (C) e como classes executáveis Java (`.class`).

3.3 Análise dos resultados

Após executar todas as aplicações descritas na seção anterior, obtivemos diversos resultados que são expostos e analisados nesta seção. Esta análise está dividida em três partes, de acordo com o tipo de aplicação analisada.

3.3.1 Aplicação Simples

Esta aplicação, ao ser invocada pelo servidor HTTP, gera uma página contendo a informação lida de um arquivo de 50 bytes e imprime o código HTML na saída padrão (que no caso é a porta HTTP) onde será recuperado pelo browser. É uma aplicação que exige pouco do sistema (processador), de forma que o desempenho da *linguagem* deve ter menos influência no tempo de resposta do que o desempenho das operações de entrada/saída da tecnologia utilizada (*I/O bound*).

---

[8] Número de repetições varia por aplicação. Veja detalhes sobre cada aplicação nas seções 3.3.1, 3.3.2 e 3.3.3.

Servlet pode ser mais eficiente que CGI mesmo quando a linguagem usada no CGI é de desempenho superior a Java (como é o caso do C).

Os servlets foram executados usando dois interpretadores diferentes. O interpretador *default* do JRE possui um *Just-In-Time Compiler* que traduz o código de máquina Java (*bytecode*) para código de máquina do sistema durante a execução. O interpretador com JIT perde um pouco de tempo no início da aplicação para realizar a compilação, mas depois, durante a execução, se comporta como uma aplicação que é executada diretamente pelo processador do sistema. O interpretador sem o JIT pode ser ativado no JRE através de configuração. Não é *default* na plataforma *Windows* mas não está disponível em todas as plataformas Java.

Devido ao JIT, a inicialização do *servlet* pode levar um tempo maior, mas isto só acontece uma vez. Depois da primeira execução, o *servlet* permanece ativo como uma extensão do servidor enquanto o servidor não for reinicializado.

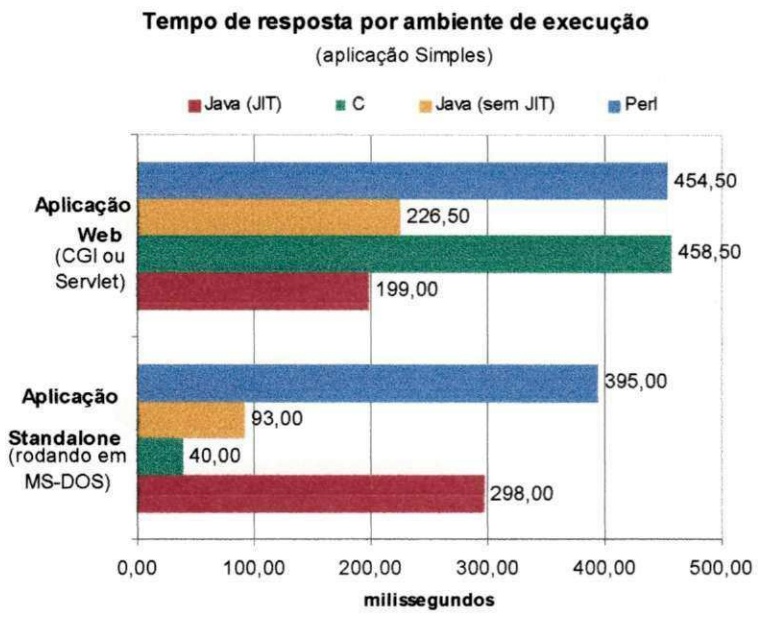


Figura 3-6 – Desempenho aplicação Web vs. aplicação *standalone*

Para melhor avaliar o impacto no desempenho causado pelo servidor Web, executamos a parte do servidor das aplicações Web como aplicações *standalone* no *MS-DOS* e comparamos com os resultados via Web para um único cliente. Os resultados observados estão na figura 3-6.

Como esperado, a aplicação em C obteve o melhor desempenho rodando como aplicação independente. O desempenho em Perl foi bem mais lento, tratando-se de uma linguagem interpretada. Já o desempenho do programa em Java *com* interpretador JIT foi *pior* que o desempenho da mesma aplicação *sem* o interpretador JIT, chegando a ser pior, inclusive, ao desempenho do próprio programa com interpretador JIT na sua versão servlet! A explicação, desta vez, está no *overhead* introduzido pelo interpretador JIT. Como a aplicação é muito simples, e não exige muito do sistema, o interpretador convencional

(sem JIT) obtém um resultado razoável (cerca de 2 vezes e meia pior que a aplicação em C). Já o interpretador com JIT perde tempo realizando a compilação do código para só depois executar. Quando roda como servlet, isto só acontece na primeira vez. Nas outras vezes, o servlet já está instalado e o JIT não precisa ser acionado.

Os valores medidos nas aplicações que usam CGI praticamente refletem o *overhead* introduzido na criação de um novo processo no sistema, o que não ocorre com servlets.

3.3.2 Aplicação Combina

Esta aplicação, ao ser invocada pelo servidor HTTP, realiza a seguinte operação

$C = n! / ((n - k)! k!)$

para n variando de 0 a 100 e, para cada n, k variando de 0 a n, imprimindo um ponto "." para cada n divisível por 10. Depois de completada a operação, a aplicação gera uma página contendo os pontos impressos e imprime o código HTML na saída padrão onde será recuperado pelo browser. É uma aplicação que exige muito do sistema (*CPU bound*), de forma que o desempenho da linguagem deve ter mais influência no tempo de resposta do que a tecnologia utilizada.

Utilizando as páginas Web `frameJava2.html` (com o JIT ligado e desligado), `frameC2.html` e `framePerl2.html`, da forma explicada na seção 3.2, medimos o tempo de resposta da aplicação no servidor com 1 a 15 clientes conectados simultaneamente. Os resultados desses testes (valores médios baseados em 10 a 20 repetições) estão mostrados na tabela 3-2. Os valores medidos foram plotados no gráfico da figura 3-7. Os intervalos de confiança referem-se a um nível de confiança de 95%.

Tabela 3-2 – Tempo de resposta (em milissegundos) para a aplicação Combina

Clientes	CGI/Perl		CGI/C		Servlet Java		Servlet/Java sem JIT compiler	
	Média	*Int. Conf.*	*Média*	*Int. Conf.*	*Média*	*Int. Conf.*	*Média*	*Int. Conf.*
1	**17995,00**	1,00%	**695,00**	1,75%	**373,00**	6,55%	**1161,00**	1,57%
5	**90137,00**	0,43%	**3045,00**	1,44%	**1891,00**	1,34%	**5747,00**	0,56%
10	**181670,00**	0,41%	**6032,00**	2,62%	**3806,00**	0,88%	**11593,00**	0,62%
15	**271898,00**	0,17%	**8964,50**	1,55%	**5759,00**	0,68%	**17435,50**	0,49%

Os resultados desta vez já destacam as diferenças entre as aplicações devido ao desempenho da linguagem de programação utilizada, mas ainda refletem a influência da arquitetura utilizada como extensão do servidor Web.

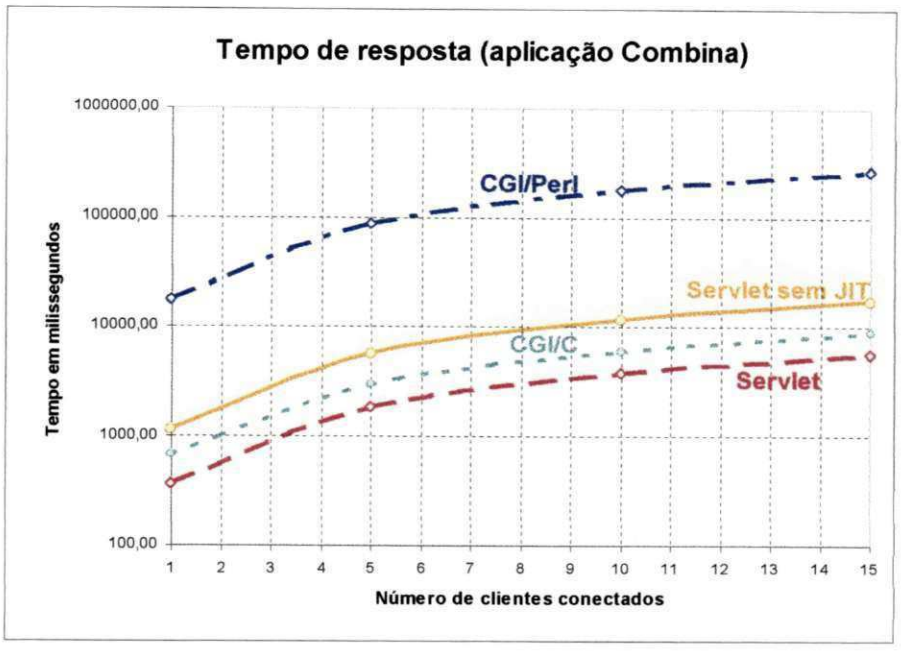


Figura 3-7 – Tempo de resposta das aplicações CGI (usando Perl e C), e Servlet (usando Java).

Por ser um programa que exige mais recursos do sistema, as aplicações interpretadas tiveram desempenho pior. A aplicação Perl, por exemplo, levou 48 vezes mais tempo para retornar que a aplicação usando servlets com interpretador JIT.

Os servlets ainda tiveram um desempenho superior em relação a CGI. O servlet (com JIT) respondeu quase duas vezes mais rápido que a aplicação em C e a resposta do servlet sem o JIT não chegou a levar o dobro do tempo em relação à aplicação em C (o que é muito bom, considerando-se que trata-se de código interpretado). Os resultados mostram que a arquitetura Servlet pode oferecer desempenho superior a CGI mesmo em aplicações que exigem muito do sistema.

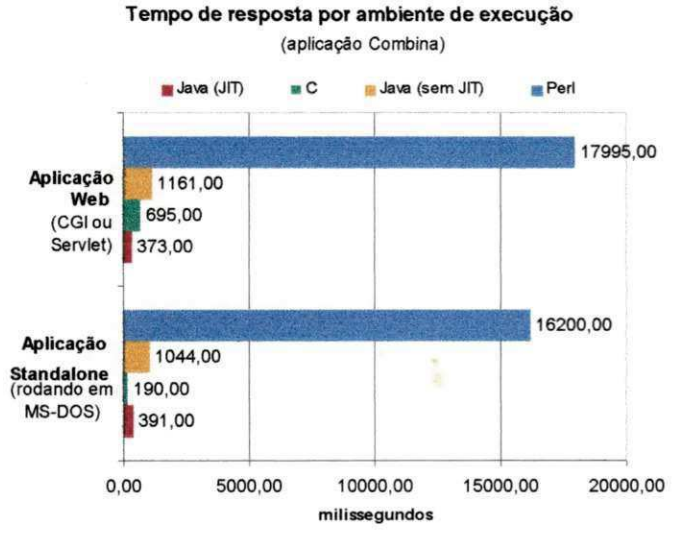


Figura 3-8– Desempenho aplicação Web vs. aplicação standalone

Novamente realizamos testes com aplicações independentes (*MS-DOS*) e obtivemos o gráfico da figura 3-8. O gráfico mostra que, como aplicação independente, o programa escrito em C mostrou-se duas vezes mais veloz que uma aplicação Java (e cinco vezes mais veloz que uma aplicação Java sem JIT).

Portanto, podemos concluir que o baixo desempenho na Web é devido não ao programa em si, mas à ineficiência da arquitetura CGI implementada no *FastTrack* server.

3.3.3 Aplicação CombinaMax

Esta aplicação é idêntica à aplicação `Combina`, mas calcula 120 combinações (em vez de 100) e repete o cálculo 20 vezes. Exige muito mais do sistema que a aplicação `Combina`, de forma que o desempenho da linguagem deve ser o efeito predominante no tempo de resposta.

Utilizando as páginas `frameJava3.html` (com o JIT ligado apenas) e `frameC3.html`, da forma explicada na seção 3.2, medimos o tempo de resposta da aplicação no servidor com 1 a 12 clientes conectados simultaneamente. Os resultados desses testes (valores médios baseados em 10 repetições) estão mostrados na tabela 3-3. Os valores medidos foram apresentados no gráfico da figura 3-9. Os intervalos de confiança referem-se a um nível de confiança de 95%.

Tabela 3-3 – Tempo de resposta (em milissegundos) para a aplicação CombinaMax

Clientes	CGI/C		Servlet Java	
	Média	*Int. Conf.*	*Média*	*Int. Conf.*
1	**6617,00**	0,60%	**6532,00**	0,30%
2	**13303,00**	1,88%	**13153,00**	0,31%
4	**26033,00**	0,35%	**26249,00**	0,17%
8	**52399,00**	0,59%	**52635,00**	0,13%
12	**79184,00**	0,86%	**79413,00**	0,15%

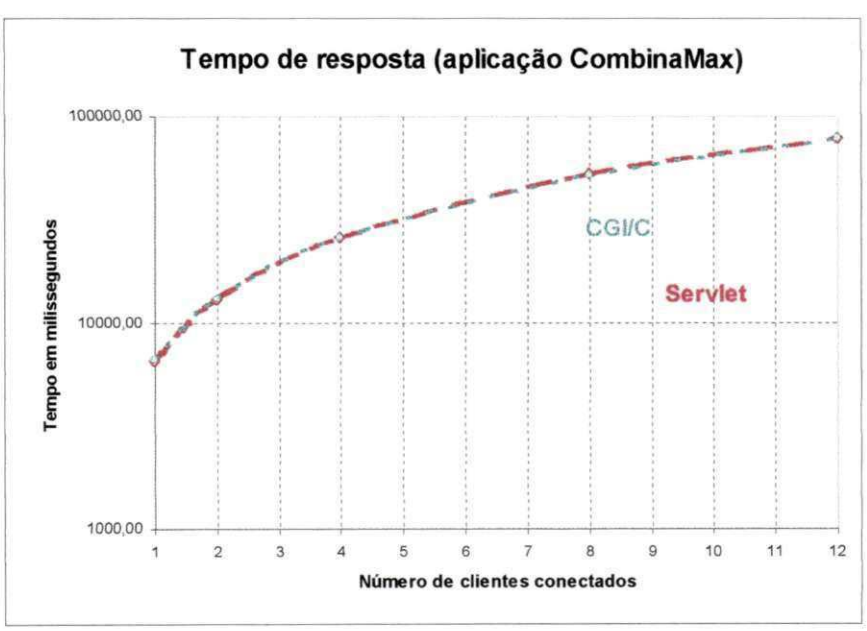


Figura 3-9 – Tempo de resposta das aplicações CGI (usando Perl e C), e Servlet (usando Java).

Com esta aplicação, conseguimos igualar os desempenhos de servlets em Java com CGI em C, em um servidor *Netscape FastTrack* rodando no *Windows95*. Não comparamos as duas tecnologias com servlets sem JIT (+ de 30s) e CGI com Perl (+ de 600s) por já sabermos, do teste anterior, que estas tecnologias seriam muito mais lentas.

Mas a aplicação em C é bem mais rápida que a aplicação em Java? Não. Como a aplicação é mais demorada, o efeito do *overhead* do compilador JIT é atenuado e a aplicação em C é apenas 12% mais veloz que a aplicação em Java, quando executam de forma independente. Os resultados do experimento estão mostrados na figura 3-10.

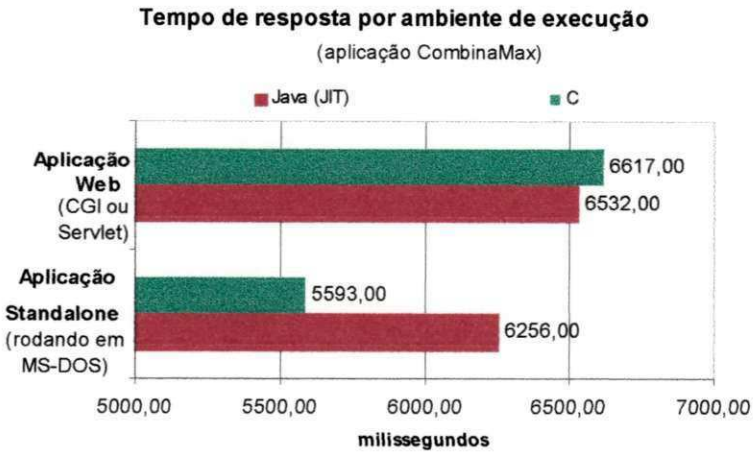


Figura 3-10 – Desempenho aplicação Web vs. aplicação *standalone*

3.4 Conclusão

Neste capítulo apresentamos resultados de experimentos envolvendo aplicações Web construídas com CGI e com servlets. Medimos o tempo de resposta em aplicações limitadas pelos recursos de entrada e saída (*I/O bound*), e em aplicações limitadas pelos recursos da CPU (*CPU bound*). Apesar dos experimentos terem se limitado a uma única plataforma, podemos, com base nos resultados e nas discussões sobre a arquitetura dos servlets e CGI, prever um desempenho superior para servlets em aplicações Web típicas, que se situam em um patamar intermediário (mais *I/O bound* que *CPU bound*).

O uso da plataforma *Windows* para hospedar um servidor Web não é raro em uma Intranet, onde freqüentemente a publicação de informação é distribuída pelos departamentos de uma empresa. Por ser uma plataforma mais limitada, a importância de se ter uma aplicação eficiente é maior. Assim, parece ser melhor desenvolver aplicações Web nessas plataformas usando servlets em vez de CGI pois verificamos um desempenho superior mesmo nos casos em que o programa CGI foi escrito em uma linguagem de alto desempenho, como C.

Este trabalho oferece alguns subsídios que podem orientar a escolha de uma tecnologia para aplicações Web em plataformas *Windows*, mas não fornece um panorama completo e genérico sobre o comportamento de servlets e aplicações CGI. Seus resultados são limitados uma vez que não foi analisado o comportamento de programas CGI e servlets em outras plataformas e servidores, mas apenas no *FastTrack Server 2.0* em *Windows 95*. Outras plataformas podem apresentar implementações melhores ou piores da máquina virtual Java. Outros servidores podem também oferecer mecanismos que melhoram o desempenho de aplicações CGI. Mas como os programas aqui apresentados podem ser instalados em outros servidores e plataformas, tais diferenças poderão ser verificadas pelo leitor que desejar ter dados mais precisos.

Capítulo 4
Cenários de uso de aplicações distribuídas

A finalidade deste capítulo é apresentar exemplos que ilustram o uso prático das tecnologias discutidas nos capítulos anteriores em cenários onde aplicações distribuídas são necessárias, comparando o desempenho (tempos de resposta) das aplicações em cada caso. O problema de acesso a um banco de dados localizado em uma rede TCP/IP será solucionado empregando tecnologias como CORBA, RMI e soquetes TCP em aplicações executando em três cenários diferentes. Os cenários se distinguem pela interface do usuário e plataforma utilizada. Na plataforma Web teremos um applet Java rodando no browser, e páginas HTML geradas por um servlet rodando no servidor HTTP. Na plataforma *Windows* teremos um programa executável Java. Realizaremos ainda testes para medir o tempo de resposta relativo em cada cenário, utilizando uma das tecnologias de rede, e discutindo a aplicabilidade de cada solução.

4.1 Introdução

A aplicação analisada permite o acesso a um banco de informações armazenadas em um arquivo de texto ou em banco de dados relacional, localizado em um servidor remoto. O acesso pode ser direto ou através de uma das quatro aplicações intermediárias que interceptam as requisições do cliente. Essas aplicações foram construídas para atuar como servidores e realizar a comunicação usando CORBA, RMI, RMI sobre IIOP ou soquetes TCP (`java.net`).

As aplicações mencionadas acima executam sob o sistema operacional local. Também analisaremos aplicações que funcionam sob a plataforma Web. São mais duas versões da aplicação de banco de dados, usando tecnologias Web *client-side* e *server-side*. A primeira versão, consiste de uma interface do usuário proporcionada por um applet Java – cuja lógica da aplicação reside no browser. A segunda aplicação Web utiliza HTML e JavaScript como interface do usuário e concentra a lógica da aplicação em um servlet Java instalado no servidor.

4.2 Apresentação e execução das aplicações

As figuras 4-1 a 4-5 ilustram as interfaces do usuário das aplicações analisadas. Todas possuem o mesmo núcleo. Foram construídas separando a lógica da aplicação (que não muda, de aplicação para aplicação) da interface do usuário e de armazenamento (que podem mudar).

4.2.1 Aplicações independentes (executam sob o sistema operacional)

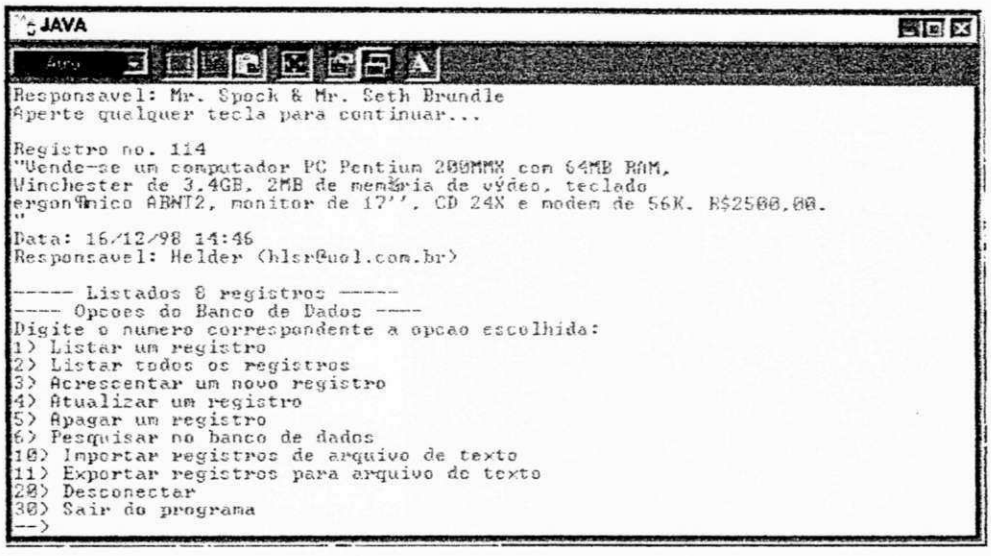


Figura 4-1 - Aplicação cliente com interface do usuário orientada a caracter

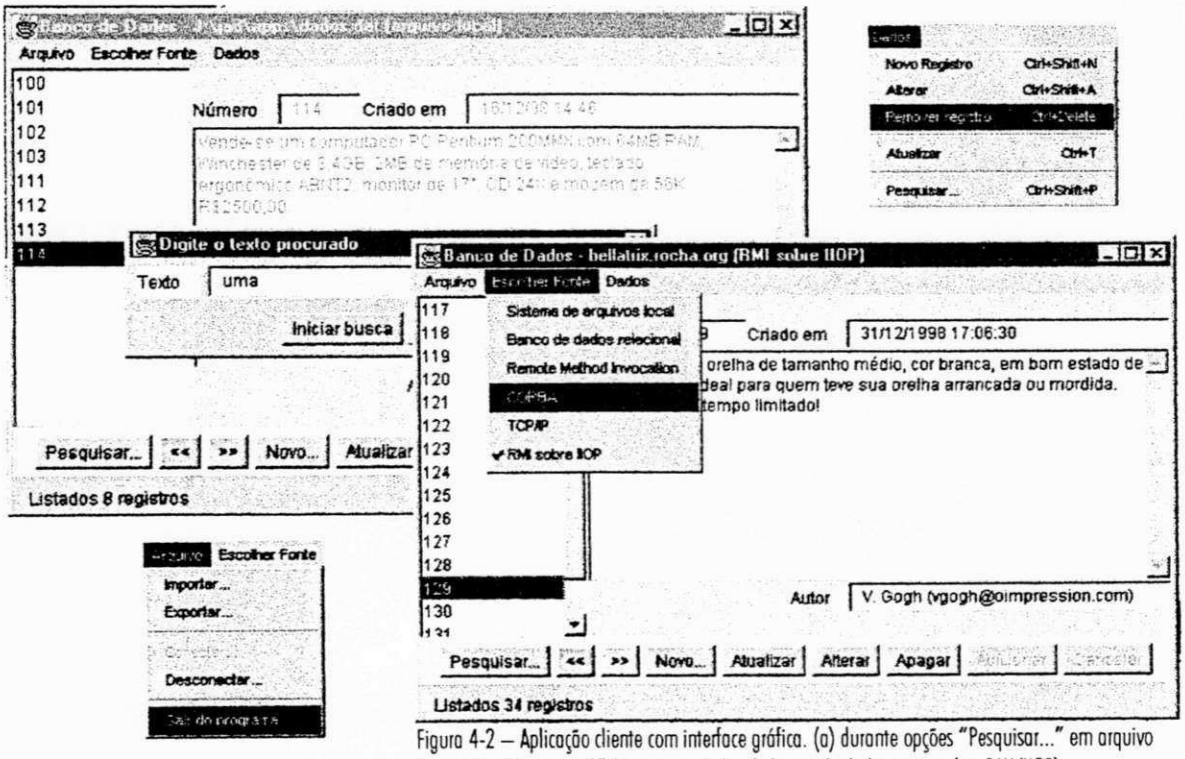


Figura 4-2 – Aplicação cliente com interface gráfica. (a) durante opções "Pesquisar..." em arquivo local; (b) e (c) menus; (d) listando os dados de banco de dados remoto (via RMI/IIOP)

4.2.2 Aplicações Web: Applets e Servlets

Figura 4-3 (a)– Interface cliente como applet em browser Netscape (com opção "pesquisar..." ativada)

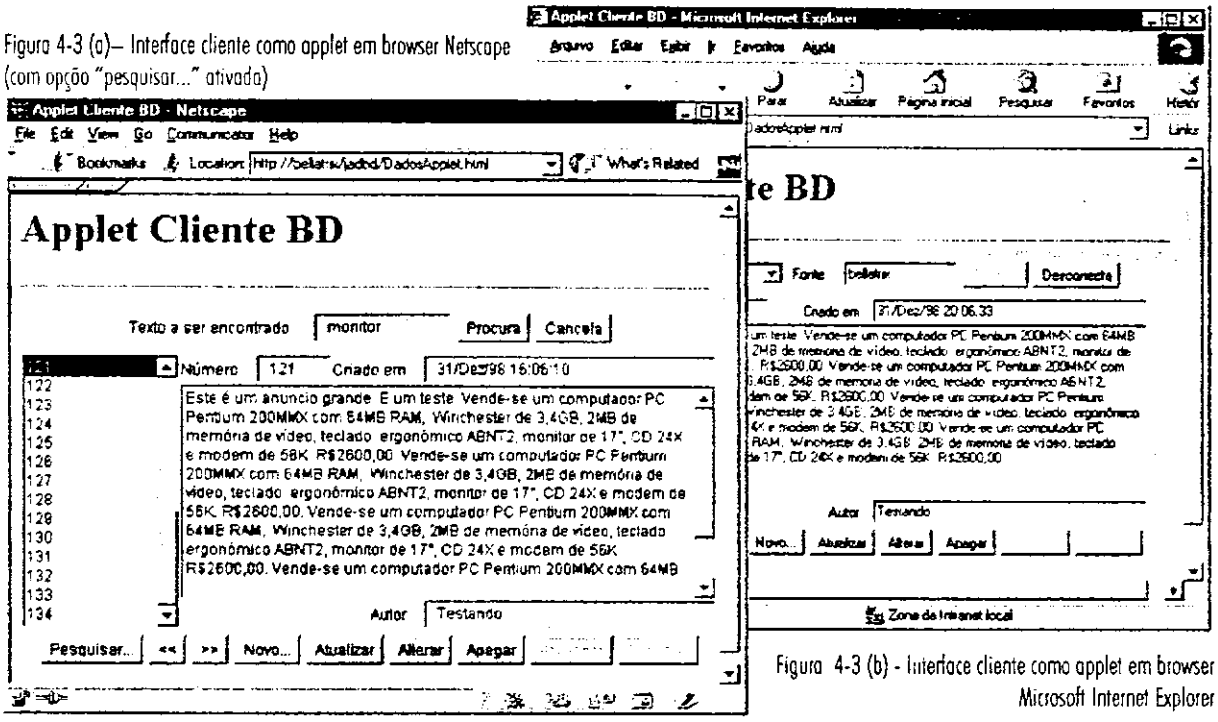


Figura 4-3 (b) - Interface cliente como applet em browser Microsoft Internet Explorer

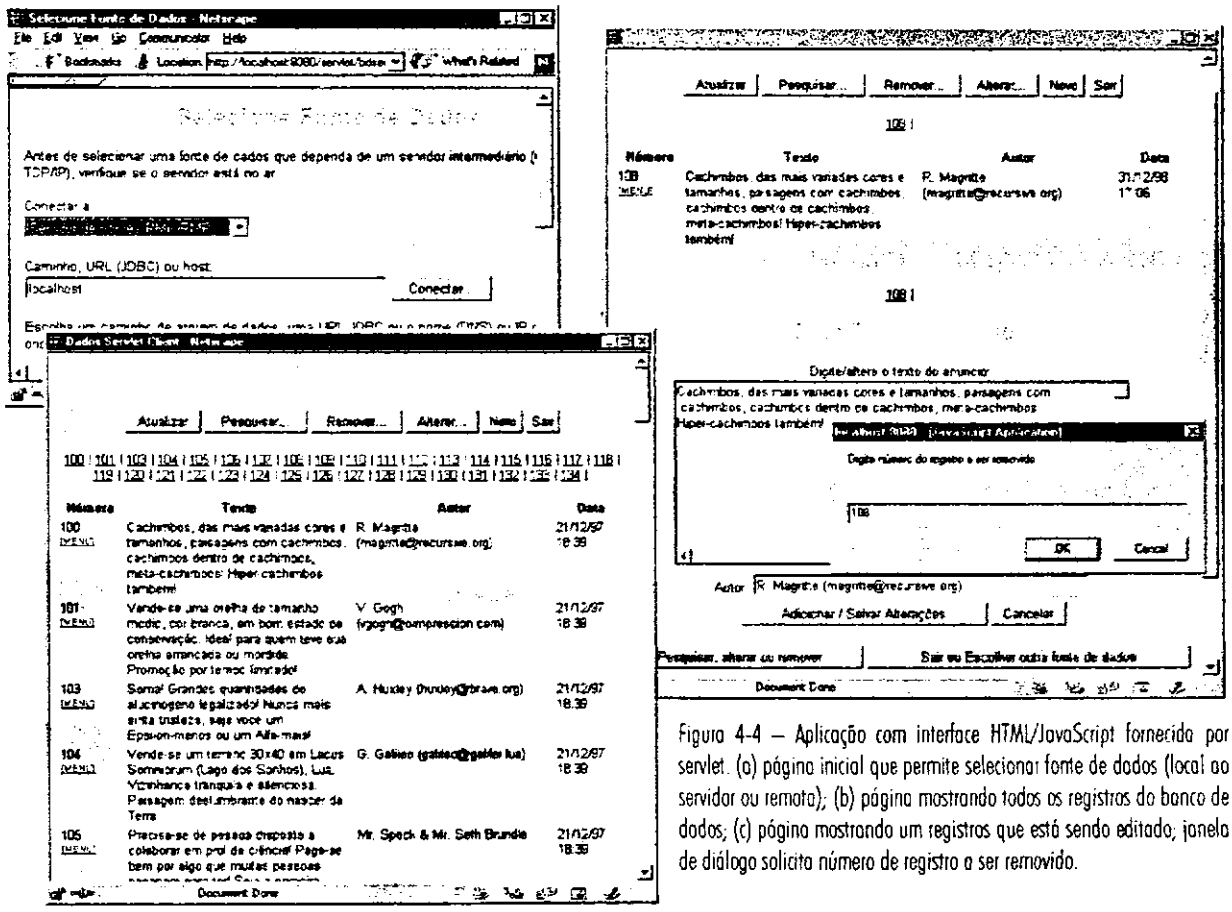


Figura 4-4 – Aplicação com interface HTML/JavaScript fornecida por servlet. (a) página inicial que permite selecionar fonte de dados (local ao servidor ou remota); (b) página mostrando todos os registros do banco de dados; (c) página mostrando um registros que está sendo editado; janela de diálogo solicita número de registro a ser removido.

4.2.3 Aplicações intermediárias (servidores)

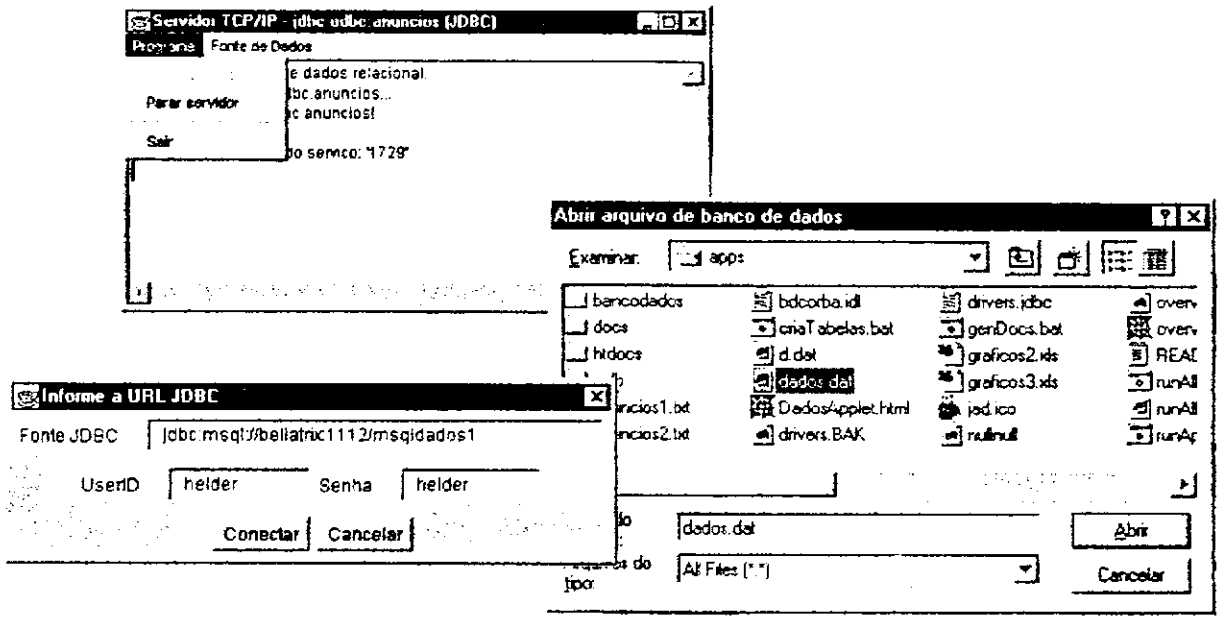


Figura 4-5 – (a) Aparência geral dos servidores intermediários (BDProtocol, CORBA, RMI, RMI/IIOP).
(b) Diálogo para conectar com banco de dados em arquivo. (c) Diálogo para conectar em banco de dados relacional através de URL JDBC

O banco de dados consiste de um conjunto de "anúncios" com número, nome, data e conteúdo. Todas as interfaces do usuário são adaptadas para entender esse formato. A tecnologia utilizada para armazenamento e a tecnologia utilizada para a comunicação remota podem ser diferentes. Todas são também suportadas pela aplicação. Em todas as interfaces console (figura 4-1), gráfica (4-2), applet (4-3) e servlet (4-4) é possível escolher se a transferência das informações será através de acesso local ou remoto usando CORBA, RMI, RMI sobre IIOP e TCP/IP. A aplicação também suporta qualquer banco de dados JDBC ou um formato proprietário baseado em arquivo para armazenar os dados. Os vários "blocos destacáveis" da aplicação estão ilustrados na figura 4-6.

Não é objetivo deste capítulo comparar o desempenho entre tecnologias de objetos remotos (isto foi feito em uma aplicação especialmente construída para esse fim no capítulo 2), portanto, as diferenças entre CORBA, RMI e TCP/IP não serão consideradas nas aplicações analisadas aqui (já foram no capítulo 2). Pretendemos, porém, discutir o funcionamento de uma aplicação funcionando com interface applet, HTML com servlet ou como executável do sistema operacional. Para isto, mediremos tempos de resposta e requisição entre cliente e servidor usando uma das tecnologias de rede (TCP/IP), e cada uma das três interfaces. As configurações utilizadas nos experimentos deste capitulo estão ilustradas na figura 4-7.

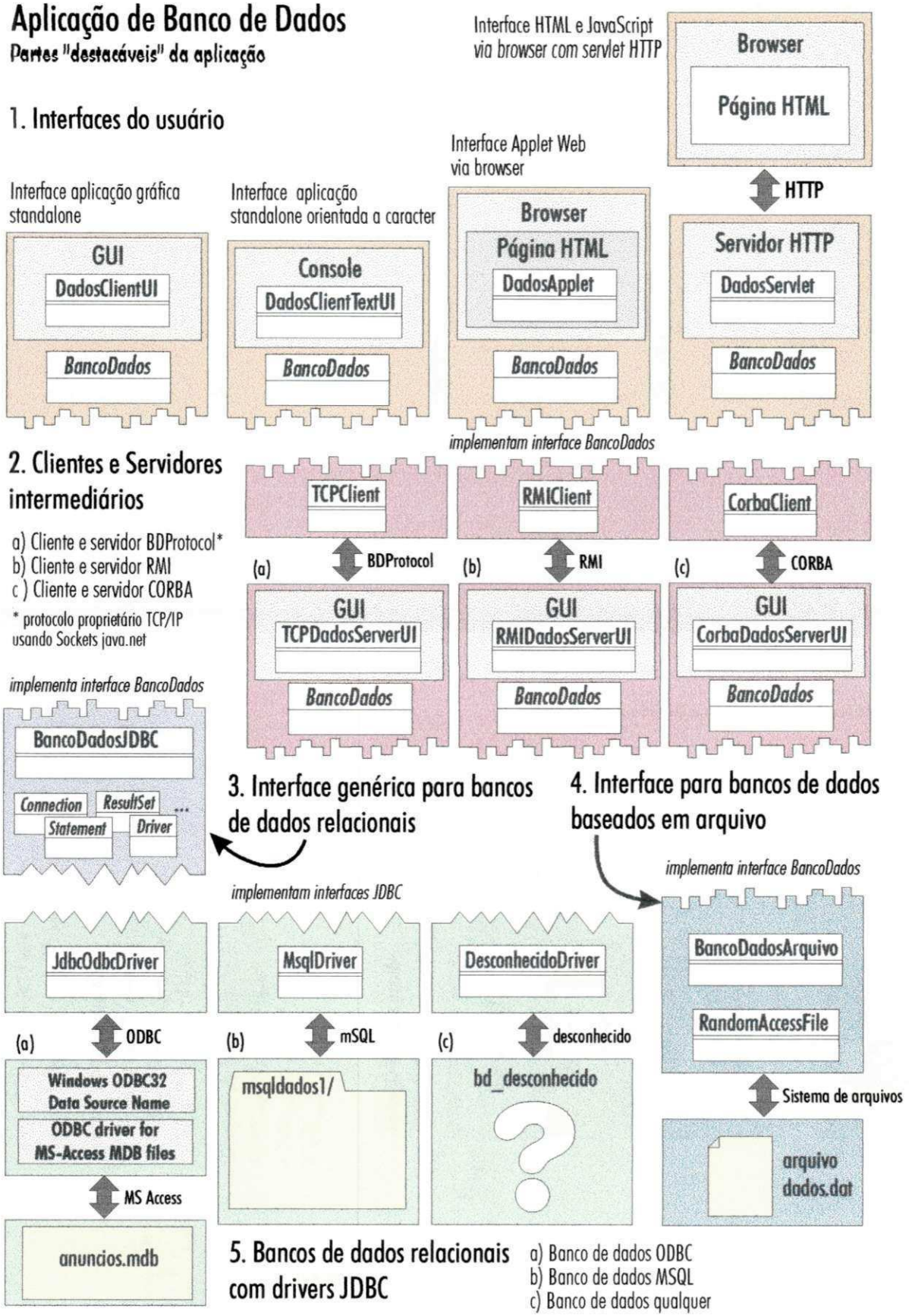


Figura 4-6 - Partes destacáveis da aplicação "bancodados". Veja mais detalhes sobre a estrutura da aplicação no apêndice C.

4.3 Execução das aplicações

Os pacotes (módulos Java) desta aplicação são compartilhados tanto pelas aplicações cliente como pelas aplicações servidoras, portanto, para rodar a aplicação em uma máquina é preciso instalar todo o pacote, mesmo que apenas parte da aplicação seja usada. As aplicações analisadas neste capítulo estão no diretório `/jad/apps/` criado após a descompactação do disquete distribuído juntamente com esta dissertação.

As aplicações podem ser executadas invocando diretamente o interpretador Java ou através de roteiros `.BAT` (Windows) ou Bourne-Shell (Unix). Para executar qualquer aplicação é preciso iniciar uma classe Java executável (cujo nome termina em `UI`). Também é preciso definir uma propriedade que informa o diretório de trabalho da aplicação. As propriedades podem ser passadas via linha de comando através do argumento `-D` do interpretador Java. O `classpath` só precisa ser definido se a aplicação for executada fora do seu diretório de trabalho. A sintaxe geral para executar *qualquer* programa da aplicação de banco de dados é:

```
java -Dapp.home=/jad/apps -classpath c:/jad/apps bancodados.user.xxxUI
```

É mais fácil iniciar uma aplicação usando um dos roteiros de execução disponíveis, em MS-DOS (ou Bourne-Shell). Para usar os roteiros é preciso configurá-los para que contenham o endereço correto do interpretador Java e definam corretamente variáveis de ambiente e propriedades do sistema. As instruções de como configurar as aplicações estão presentes como comentários no código de cada roteiro e na documentação HTML da aplicação.

Os roteiros executáveis *Unix* têm extensão `.sh` e os executáveis *Windows* têm extensão `.lnk` ou `.bat`. As aplicações Web têm extensão `.html`. Todos os programas/páginas têm a forma `runXXX`. Pode não ser necessário configurá-los caso o diretório `/jad/` tenha sido instalado nas localidades *default* (`c:\` para *Windows* e `~/` para *Unix*). Os programas, localizados em `/jad/apps/`, são:

- `runRMIServer.bat` e `runRMIServer.sh` - Executa servidor RMI e `rmiregistry`
- `runRIIOPServer.bat` e `runRIIOPServer.sh` - Executa servidor RMI/IIOP e `tnameserv`
- `runTCPServer.bat` e `runTCPServer.sh` - Executa servidor BDProtocol TCP/IP
- `runCorbaServer.bat` e `runCorbaServer.sh` - Executa servidor CORBA e `tnameserv`

- `runConsole.bat` e `runConsole.sh` - Executa cliente orientado a caracter
- `runGraphic.lnk` e `runGraphic.sh` - Executa cliente gráfico Windows/X-Window
- `runApplet.bat` e `runApplet.sh` - Roda *Applet* no `appletviewer`
- `runWeb.html` - Página que inicia a aplicação *Applet* (pode necessitar de servidores) e a aplicação *Servlet* (necessita de `servletrunner` executando). URL *default* para servlet é: `http://localhost:8080/servlet/bdservlet`.
- `runServlet.bat` e `runServlet.sh` - Inicia servidor Web `servletrunner`

Toda a documentação sobre a aplicação, incluindo suas classes, interfaces, métodos, pacotes está no diretório `jad/apps/docs/` a partir do arquivo HTML `index.html`.

Maiores detalhes sobre a construção das aplicações estão no apêndice C.

4.4 Experimento: Aplicações Web *vs.* Aplicações Windows

Nesta seção, executaremos a aplicação apresentada nas seções anteriores utilizando três arquiteturas diferentes, da forma ilustrada pela figura 4-7. Duas funcionam através de extensões *client-side* e *server-side* à plataforma Web. A última é uma aplicação gráfica nativa. Em cada caso, mediremos o tempo de resposta na transferência de 100 registros, com 1000 caracteres cada, utilizando um protocolo proprietário construído com o pacote `java.net`.

Nos experimentos com servlets não foi possível utilizar o *FastTrack* server (como no capítulo 3), por utilizarmos um módulo Java (*JRun*) limitado que não permitiu a definição de parâmetros necessários à execução do servlet[1]. Utilizamos, porém, o *Servletrunner* – pequeno servidor Web distribuído no *Java Servlet Development Kit*[2]. Os experimentos com applets, porém, fazem requisições HTTP ao servidor *FastTrack 2.0* utilizado no capítulo 3.

Para evitar a necessidade de sincronização dos relógios das máquinas, realizamos este experimento em uma única máquina. O computador utilizado foi um *Pentium* 200MMX com 32MB de memória, rodando *Windows 95*. A precisão do relógio é de 110ms, o que permite a medição de valores com intervalos de 50 e 60ms (o menor valor mensurável e maior que zero é

---

[1] Aparentemente o *JRun* suporta os parâmetros, mas a sua configuração não funcionou na versão utilizada.

[2] O *Servletrunner* é um servidor Web limitado. Não serve páginas. Apenas servlets. Sua finalidade é servir de ambiente de testes para servlets (e não ser utilizado em aplicações reais).

50ms. O segundo menor valor é 100ms, o terceiro 110ms, etc.). O browser utilizado para visualizar applets e páginas HTML foi o *Netscape Navigator 4.5*.

Medimos o tempo de acesso *local* utilizando as aplicações *Windows* e servlet para ler o arquivo que contém o banco de dados *diretamente*.[3] Esse tempo foi medido 60 vezes e obtivemos o valor: 2307,67 com intervalo de confiança 0,27% (para nível de confiança de 95%). Esse tempo, que representa uma medida independente da forma ou protocolo de acesso, foi subtraído dos outros resultados para expor apenas o tempo de resposta e o tempo de requisição de cada transferência. Utilizando o protocolo proprietário "BDProtocol", construído usando os recursos TCP/IP nativos do Java, medimos o tempo absoluto[4] em três instantes e calculamos os tempos de requisição, resposta e total da transação (RTT – *Round Trip Time*) para as aplicações *Windows*, servlet e applet. A tabela 4-1 ilustra os resultados de 30 repetições (apertando o botão "Atualizar" 30 vezes):

Tabela 4-1 – Tempos de transferência

Cenários	Aplicação Windows		Applet		Servlet	
	Tempo	*Int. Conf.*	*Tempo*	*Int. Conf.*	*Tempo*	*Int. Conf.*
RTT*	**574,67**	4,11%	**1191,00**	3,10%	**1292,33**	6,31%
Resposta	**461,67**	3,58%	**700,00**	2,56%	**815,33**	9,14%
Requisição	**113,00**	10,65%	**491,00**	4,44%	**477,00**	11,46%

* não leva em consideração o tempo ocupado no servidor processando os dados (valor constante – em torno de 2308ms para todas as aplicações)

A figura 4-8 mostra os resultados gráficos da tabela 4-1.

Os valores apresentados são *calculados*. Os instantes de tempo *medidos* correspondem ao instante em que o método que faz a requisição é iniciado no cliente (t_1), o instante em que o método no servidor devolve os dados (t_2) e o instante em que o método que fez a requisição no cliente termina (t_3).

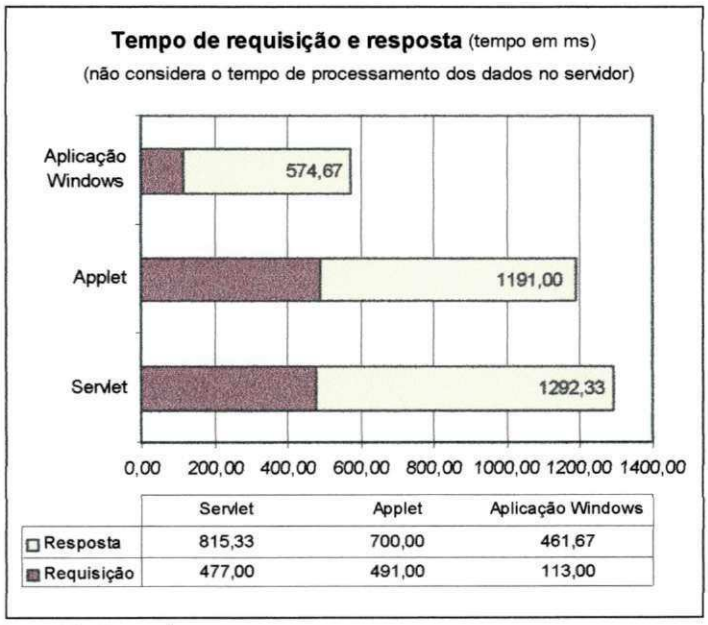


Figura 4-8 – Tempos de requisição e resposta

[3] Por causa de restrições de segurança, applets não permitem acesso direto a arquivos em disco local.

[4] UTC – Universal Coordinated Time. Número de milissegundos desde 1/1/1970 às 0:00:00 GMT.

O tempo $t_3 - t_1$ corresponde ao RTT (veja figura 4-7). A resposta foi calculada subtraindo t_3 de t_2. A requisição equivale a $t_2 - t_1 - t_{local}$ (tempo do acesso local: ~2308ms). Como a precisão do relógio da plataforma onde os testes foram realizados está em torno de 100ms, medidas próximas deste valor apresentaram intervalos de confiança maiores, principalmente o tempo de requisição, por ser calculado a partir de valores medidos próximos de 100ms.

Os tempos totais (RTT) das aplicações Web são equivalentes, considerando os intervalos de confiança, porém as causas do atraso em cada aplicação são diferentes, já que a comunicação utiliza protocolos diferentes. Apesar de executar em um browser, o applet se comunica com o servidor utilizando o protocolo "BDProtocol", da mesma maneira que a aplicação *Windows*, no entanto, apresenta um atraso equivalente à aplicação servlet, cujo atraso pode ser explicado por esta última possuir uma camada adicional (veja figura 4-7) onde a comunicação é realizada através de HTTP.

Para compreendermos melhor o significado dos tempos de transferência em cada cenário, e saber onde realmente ocorre cada atraso, medimos 5 a 7 instantes durante a execução em cada aplicação (2 a mais na aplicação servlet). Através dessas medidas, podemos identificar as partes da aplicação que provocam cada atraso. A figura 4-7 ilustra esses intervalos.

Com esses instantes, pudemos calcular três aspectos do tempo de transferência de dados (requisição e resposta) de cada aplicação. Na aplicação servlet, calculamos o tempo de requisição e resposta entre o browser e servidor Web, que é o tempo devido a requisições e respostas HTTP. Esse tempo ocorre fora do servlet, e foi medido via JavaScript, na página HTML. Chamamos de "Tempo *HTTP*". No applet, o tempo de requisição e resposta ocorre dentro da aplicação e não foi possível separá-lo do tempo ocupado pelo protocolo de transferência proprietário utilizado, que chamamos de "Tempo devido a *protocolos*". O restante do tempo refere-se ao tempo de transferência entre o cliente e servidor que chamamos de "Tempo da *aplicação*".

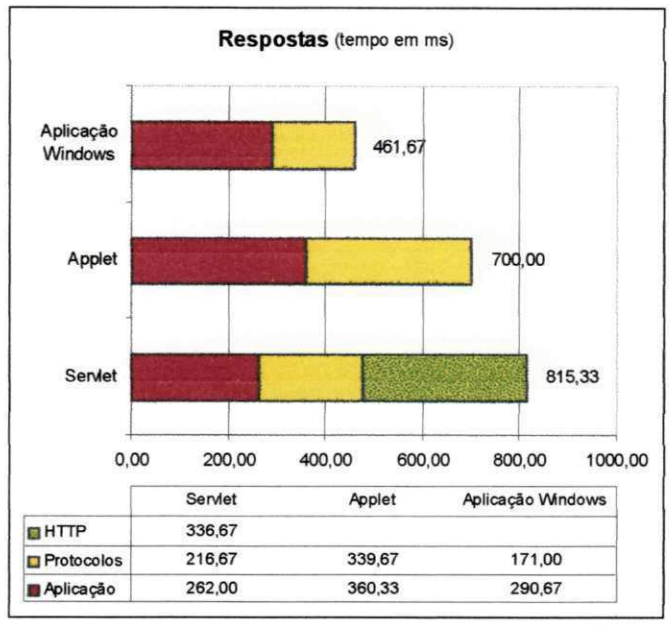
A tabela 4-2 organiza os tempos calculados com base nos instantes medidos. Todos os intervalos de confiança referem-se a um nível de confiança de 95%.

Tabela 4-2 – Tempos de transferência

	Windows				**Applet**				**Servlet**			
	Requisição		Resposta		Requisição		Resposta		Requisição		Resposta	
	Média	*I. Conf.*	*Média*	*I. Conf.*	*Média*	*I. Conf.*	*Média*	*I. Conf.*	*Média*	*I. Conf.*	*Média*	*I. Conf.*
Total*	**113,00**	10,65%	**461,67**	3,58%	**491,00**	4,44%	**700,00**	2,56%	**477,00**	11,46%	**815,33**	9,14%
Aplicação	**113,00**	10,65%	**290,67**	3,92%	**303,67**	5,97%	**360,33**	4,93%	**201,00**	22,64%	**262,00**	13,12%
Protocolos	**0,00**	-	**171,00**	9,35%	**187,33**	6,71%	**339,67**	3,77%	**0,00**	-	**216,67**	16,41%
HTTP	**-**	-	**-**	-	**-**	-	**-**	-	**276,00**	12,87%	**477,00**	11,65%

Devido aos baixos valores medidos (próximos da precisão da máquina), alguns valores da tabela acima apresentam intervalos de confiança elevados.

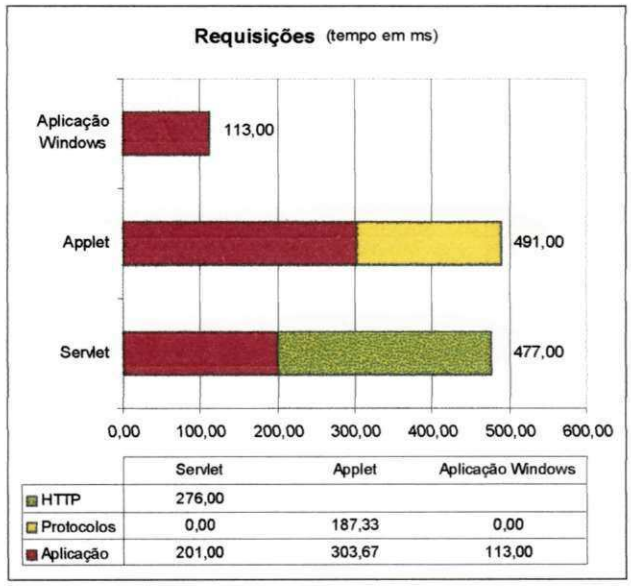
As figuras 4-9 e 4-10 mostram os resultados obtidos de forma gráfica.



	Servlet	Applet	Aplicação Windows
HTTP	336,67		
Protocolos	216,67	339,67	171,00
Aplicação	262,00	360,33	290,67

Figura 4-9 - Respostas

O tempo de resposta medido, infelizmente, não concentra apenas o tempo ocupado na transferência dos dados. Antes da transferência, o programa ainda realiza uma transformação nos dados que pesa sobre o tempo de resposta medido. A transformação, porém, é parte do protocolo *BDProtocol* e deve ocupar um tempo semelhante em todas as aplicações. Não é possível afirmar que o tempo devido a Protocolos é maior na aplicação que usa servlet, que na aplicação *Windows*, devido ao alto intervalo de confiança (16,41%) observado. Já o tempo "Protocolos" do applet certamente é maior. É aproximadamente 170ms maior que o tempo observado na aplicação *Windows*, que usa a mesma interface gráfica que o applet, mas este último a compõe dentro de um browser. Como a transferência dos dados no applet e na aplicação *Windows* é realizada da mesma forma, usando o protocolo *BDProtocol*, podemos deduzir que o atraso foi introduzido pelo browser.



	Servlet	Applet	Aplicação Windows
HTTP	276,00		
Protocolos	0,00	187,33	0,00
Aplicação	201,00	303,67	113,00

Figura 4-10 - Requisições

O atraso no servlet foi menor que no applet, mas o tempo de requisição foi maior devido ao atraso devido ao servidor HTTP, na comunicação entre a página Web e o servidor.

Nas requisições, pudemos medir o tempo de transferência de forma mais independente (sem incluir processamento de dados local, como ocorreu em relação à resposta). O resultado é que, na aplicação *Windows*, o tempo de transferência foi inferior a 50ms (e portanto, foi medido como 0,00). Nos applets, novamente tivemos um acréscimo devido ao browser, e, nos servlets, um acréscimo devido ao servidor HTTP.

Não é possível afirmar que uma aplicação Web é mais eficiente que outra uma vez que os intervalos de confiança são demasiadamente altos. A escolha entre uma ou outra tecnologia deve recair sobre outros aspectos não relacionados ao desempenho. A vantagem de se utilizar HTML em vez de um applet, por exemplo, está na possibilidade de se desenvolver para um público-alvo maior, que inclui browsers que não suportam applets. Os recursos JavaScript usados permitem que a aplicação servlet seja usada em browsers *Netscape 2.02* em diante e *Internet Explorer 3.02* em diante, enquanto que o applet só roda eficientemente em browsers *Netscape 4.5* e *Explorer 4* e *5* com Java plug-in 1.2. A vantagem do applet é que ele preserva a mesma interface usada pela aplicação *Windows* (mais ergonômica, dinâmica e fácil de usar) e oferece a possibilidade de se usar protocolos mais eficientes que HTTP na comunicação.

As duas seções seguintes discutem outros aspectos em que se pode comparar as aplicações analisadas.

4.4.1 Applets e Servlets

Usar um applet como camada de apresentação para uma aplicação localizada em um servidor remoto oferece as seguintes vantagens em relação a tecnologias baseadas no servidor:

- *Interface gráfica com mais funções e recursos.* Os recursos gráficos e interativos do HTML são limitados. Não é possível, por exemplo, redesenhar uma área da tela, trocar um formulário por outro ou fazer aparecer um texto na tela sem carregar uma nova página. Uma aplicação de pintura, que roda como um applet, é exemplo de uma aplicação que não poderia ser implementada apenas com HTML, JavaScript e servlets. Também conseguimos manter praticamente a mesma interface usada na aplicação de banco de dados, versão *Windows*, na versão applet, o que não seria praticável com servlets/HTML.
- *Resposta mais rápida a ações locais.* Usando applets, tarefas como navegação no banco de dados, mudança entre os modos de edição e navegação entre os registros do banco são mais rápidas e eficientes. Na versão HTML/servlets precisamos fazer uma

chamada ao servidor para mudar de modo e qualquer alteração implica na carga de uma nova página HTML. Mesmo que o tempo seja menor, o usuário sempre tem a impressão que o tempo é maior, já que a interface da aplicação some por uns instantes (enquanto a página é carregada).

- *Flexibilidade em relação ao protocolo de transferência de dados.* A solução HTML/servlet está presa ao protocolo HTTP que intermedia toda a sua comunicação. Usando um applet, o protocolo HTTP será usado apenas para transferir a aplicação para o browser. Depois que o applet estiver executando, poderá usar um outro protocolo aberto ou proprietário para se comunicar com o servidor [SUN95].
- *Flexibilidade em relação aos tipos de dados suportados.* Os browsers que utilizamos só suportam a exibição de imagens GIF, JPEG e PNG. Applets podem ser construídos para que suportem outros formatos proprietários [SUN 95].

Applets, porém, têm limitações. O uso de servlets para intermediar o acesso a uma aplicação remota e uma interface do usuário baseada em HTML e JavaScript permite realizar algumas tarefas difíceis ou impossíveis para os applets:

- *Interface do usuário disponível imediatamente.* A interface do usuário de uma aplicação baseada no servidor é uma página HTML, que geralmente é carregada rapidamente. Applets geralmente são maiores e levam tempo para iniciarem e aparecerem na tela.
- *Menos problemas de compatibilidade.* Aplicações baseadas no servidor podem ser utilizadas por um público-alvo mais amplo. Suporte total a Java é raro até nos browsers mais recentes. Uma aplicação baseada no servidor está praticamente imune a incompatibilidades entre versões e fabricantes de browser. O applet de banco de dados só roda sem problemas em versões mais recentes dos browsers comerciais.
- *Menos restrições de segurança.* Existem várias restrições de segurança associadas aos applets. Applets não podem[5], por exemplo, ter acesso máquinas da rede que não sejam a máquina de onde vieram. Servlets, como rodam no servidor e não no browser, estão livres desta restrição.

Applets e servlets não são tecnologias concorrentes. Podem ser usadas em conjunto com grandes benefícios, aproveitando as vantagens de ambos.

---

[5] É possível reduzir as restrições de segurança usando applets assinados.

4.4.2 Clientes Web e clientes nativos

O objetivo desta discussão é apontar as principais diferenças entre aplicações Web, executando em browsers, e aplicações *nativas*, executando em sistema operacional nativo (Windows, por exemplo). As duas formas são aplicadas em situações diferentes. Podemos utilizar os resultados quanto ao desempenho para descobrir quando vale a pena fazer o *download* da aplicação para instalação local (e acesso remoto) em vez de usar a interface proporcionada pelo applet dentro do browser.

A possibilidade de rodar uma aplicação dentro de um browser é um dos principais avanços proporcionados pela linguagem Java ao ambiente Web. As vantagens são muitas:

- *Facilidade de distribuição.* Na forma de um applet, a interface cliente da aplicação poderá ser distribuída facilmente através de uma Intranet ou da Internet, bastando que o usuário acesse a URL da página onde o applet está localizado.
- *Facilidade de atualização.* A qualquer momento a aplicação pode ser atualizada. Na próxima vez que um usuário solicitar o applet, ele já terá a última versão.
- *Facilidade de uso e instalação.* Não é preciso instalar o applet. Tendo-se um browser, é só carregá-lo.
- *Disponibilidade imediata.* O applet está imediatamente disponível. Não é preciso obter drivers externos ou instalar ambientes de execução. Os principais browsers do mercado oferecem um ambiente de execução nativo.
- *Segurança embutida.* Applets descarregados pela rede são sempre verificados pelo browser e não têm acesso ao sistema de arquivos local. Para eliminar essas restrições e ainda assim operar em um ambiente seguro, pode-se assinar applets digitalmente e fazer uso dos recursos de criptografia e autenticação disponíveis em Java.

Apesar de todas as vantagens, o ambiente Web ainda não segue um padrão bem definido. Os browsers apresentam incompatibilidades, são pouco eficientes e podem impor restrições em excesso. Rodar uma aplicação sobre o sistema operacional nativo, portanto, pode ser uma opção já que existem ambientes de execução Java para os principais sistemas operacionais, permitindo que o programa rode em *Windows*, *Unix*, *Macintosh*, etc. mesmo fora de um browser. Um usuário pode instalar a plataforma Java em sua máquina e rodar a aplicação cliente localmente, quando quiser. O browser seria usado apenas uma vez, para fazer o download da aplicação. As principais vantagens desse modelo sobre os applet são:

- *Controle sobre restrições de segurança.* As restrições impostas aos applets pelos gerentes de segurança dos browsers podem ser excessivas. Não é possível, por exemplo, fazer com que um applet[6] imprima, salve um arquivo temporário em disco local, ou realize conexões a outras máquinas da rede [GOSL96]. Com uma aplicação independente, podemos implementar um gerente de segurança mais flexível, com menos restrições.
- *Maior velocidade.* Verificamos que a aplicação de banco de dados rodando como uma aplicação *Windows* apresentou um tempo de resposta e requisição bem menor àquele obtido com a mesma aplicação, local, usando um applet. Isto não leva em conta o tráfego na rede, já que a medição foi obtida com o acesso local. O gerente de segurança e o próprio browser contribuem para o baixo desempenho do applet.
- *Independência de fabricante de browser.* Poucos browsers suportam CORBA, Swing, Java2D, RMI sobre IIOP e outros recursos que só recentemente passaram a fazer parte da plataforma Java. Se browser algum suporta um recurso essencial de uma aplicação, será necessário que ele descarregue toda a API que contém as classes usadas pelo recurso, todas as vezes em que for executado.
- *Possibilidade de oferecer mais recursos.* Applets são construídos sob um regime de austeridade. Precisam ter o menor tamanho possível e freqüentemente evitar usar novas e eficientes APIs, por falta de suporte dos browsers. Com uma aplicação nativa, é possível incluir recursos melhores, utilizar APIs proprietárias mesmo que isto resulte em uma aplicação maior. O tamanho é menos crítico pois o produto só será descarregado uma vez, e depois será instalado localmente.

4.5 Conclusão

Este capítulo apresentou exemplos de aplicações Java que utilizam as tecnologias exploradas nos capítulos anteriores. Também comparou diferentes implementações da camada de apresentação (interface do usuário) de uma mesma aplicação, que pode rodar como applet, em um browser Web; como aplicação independente, sob o sistema operacional *Windows*; ou como servlet, em um servidor Web gerando páginas dinamicamente para um browser. Esses três cenários refletem situações reais e a decisão de usar uma ou outra alternativa envolve

[6] É possível reduzir as restrições de segurança usando applets assinados.

diversos fatores discutidos neste capítulo, onde apontamos as vantagens e desvantagens de cada uma, mostrando qual o melhor cenário em que podem ser empregadas.

Os resultados obtidos fornecem subsídios que podem orientar decisões para usar uma ou outra interface do cliente. As discussões e comparações realizadas neste capítulo estão resumidas na tabela abaixo, para consulta rápida.

	Cliente Windows	**Cliente applet**	**Cliente HTML + servlet HTTP**
Interface gráfica	Completa e interativa *(usa todos os recursos do pacote java.awt)*	Completa e interativa *(usa todos os recursos do pacote java.awt)*	Limitada *(interatividade depende de programação adicional em JavaScript)*
Tempo de resposta da GUI (resposta a eventos locais)	Melhor tempo de resposta *(GUI nativa)*	Melhor tempo de resposta *(GUI nativa)*	Pior tempo de resposta *(precisa carregar nova página)*
Tempo de resposta de operações de rede	Melhor tempo de resposta	Tempo adicional devido ao browser	Tempo adicional devido ao protocolo HTTP
Protocolos suportados	Qualquer um	Qualquer um	HTTP
Formatos que podem ser exibidos (tipos de dados)	Qualquer tipo	Qualquer tipo	HTML, GIF, JPEG, PNG e tipos suportados pelo browser usado
Recursos e APIs Java suportadas pela aplicação	Todos os recursos disponíveis	Apenas recursos disponíveis no browser	Todos os recursos disponíveis
Segurança	Restrições definidas pelo programador	Restrições impostas pelo browser (ñ-assinados)	Restrições definidas pelo programador e servidor
Facilidades de distribuição, atualização e instalação	Usuário precisa instalar (uma vez) e atualizar.	Instalação automática após carga pelo browser, a cada acesso.	Instalação prévia no servidor Web. Usuário simplesmente usa o serviço.
Dependência de servidor ou browser	Não depende de outras aplicações	Browser deve suportar plataforma Java do applet (JDK 1.0, 1.1 ou Java 2)	Servidor deve suportar plataforma Java do servlet (JDK1.1 com JSDK 2.x)

Devido aos pequenos intervalos de tempo medidos, à complexidade da aplicação e à utilização de servidores e plataformas específicas, os resultados deste experimento são limitados. As conclusões referem-se ao ambiente específico onde a aplicação foi executada. Para obter suas próprias medidas de desempenho, o leitor, se quiser, pode executar as aplicações em sua própria plataforma e obter dados mais confiáveis.

Capítulo 5
Conclusões e sugestões de trabalhos futuros

5.1 Conclusões

Este trabalho apresentou estudos de casos utilizando diversas tecnologias distribuídas disponíveis ao programador Java. Nos cenários analisados, foram discutidos aspectos como desempenho (transferência de dados e tempo de resposta), limitações, facilidade de uso, compatibilidade e flexibilidade das soluções. Há casos em que o programador pode combinar uma ou mais tecnologias de rede em uma aplicação distribuída (servlets mais RMI, por exemplo). Em outros casos ele precisa escolher uma ou outra (RMI ou CORBA, por exemplo). Neste capítulo pretendemos fazer uma síntese dos resultados dos três capítulos anteriores, discutindo todas as tecnologias exploradas.

Para organizar todas as informações apresentadas, construímos três tabelas comparativas, comparando diversos aspectos das tecnologias analisadas e mostrando também onde (em que pacote nativo ou extensão) cada tecnologia é fornecida.

A tabela 5-1 compara características de aplicações Java distribuídas nos três cenários analisados no último capítulo (interfaces do usuário), fornecendo um resumo das discussões e resultados obtidos.

Tabela 5-1 – Quadro comparativo: tipos de interface do usuário de aplicações (cliente) Java.

Tipo de interface do usuário	**Cliente Windows[1]**	**Cliente applet**	**Cliente HTML + servlet HTTP**
DESEMPENHO			
Tempo de resposta e requisição	Mínimo	Tempo adicional devido ao browser	Tempo adicional devido à comunicação via HTTP
Tempo de inicialização da aplicação	Mínimo	Tempo adicional (pode ser excessivo) para *download* da aplicação.	Tempo adicional para geração e carga de página HTML

[1] Aplicação de referência.

Tipo de interface do usuário	Cliente Windows[1]	Cliente applet	Cliente HTML + servlet HTTP
LIMITAÇÕES			
Flexibilidade	Total (por ser aplicação de referência).	Interface dinâmica. Aplicação limitada por pelo browser.	Interface estática; suporta apenas tipos e protocolos suportados pelo browser.
Facilidade de uso e instalação da aplicação	Requer instalação pelo usuário (uma vez).	Instalada automaticamente (em cada acesso).	Instalada no servidor.
Facilidades ou restrições para acesso remoto	Sem restrições (programador pode defini-las).	Restrições de segurança impostas pelo browser.	Restrições definidas pelo servidor ou programador.
DESENVOLVIMENTO, MANUTENÇÃO E REUSO			
Flexibilidade de desenvolvimento[2]	Total (por ser aplicação de referência).	Limitada por herança (precisa estender a classe Applet).	Limitada por polimorfismo (precisa implementar interface Servlet)[3].
Facilidade de desenvolvimento[4]	Maior (referência).	Mesmo nível que aplicação de referência.	Requer conhecimentos de HTML e HTTP.
Quantidade de modificações e código adicional a escrever[5]	Nenhum (referência).	Poucas modificações (usa mesma interface gráfica).	Algumas modificações (precisa gerar HTML).
SUPORTE E DISPONIBILIDADE (PACOTES)			
JDK 1.2 (Java 2)	java.awt e javax.swing (ambos do núcleo JDK)	java.applet (núcleo)	javax.servlet (extensão)
JDK 1.1	java.awt (núcleo) e com.sun.java.swing (extensão)	java.applet (núcleo)	javax.servlet (extensão)
JDK 1.0	java.awt (núcleo)	java.applet (núcleo)	Não suportado.

Devido às limitações do experimento realizado, não foi possível chegar a conclusões genéricas quanto ao desempenho das aplicações. Foram mostradas formas diferentes de executar a mesma aplicação e cada uma se adapta melhor em um cenário específico. Na plataforma Web, tanto applets, servlets ou ambos podem ser usados. Servlets são indicados quando a aplicação requer comunicação com outras aplicações ou informações localizadas na máquina servidora, e quando uma aplicação necessita atingir um público-alvo amplo, que não pode ser excluído por não possuir um browser de última geração. Applets são recomendados nas aplicações Web em que os clientes possuem browsers de última geração e quando a aplicação requer

---

[2] Necessidade de seguir regras rígidas, como herança de certas classes, implementação de certos métodos, utilização dentro de certos parâmetros. Herança (extensão de classes) é bem menos flexível que polimorfismo sem herança (implementação de interfaces) uma vez que Java não suporta herança múltipla de implementações, mas permite que uma classe implemente várias interfaces.

[3] Tipicamente implementam-se servlets HTTP através de herança, estendendo a classe HttpServlet que por sua vez implementa a interface Servlet (polimorfismo sem herança).

[4] Leva em consideração número de linhas de código adicionais (em relação à aplicação *Windows*) que precisam ser escritas, linguagens e tecnologias que devem ser conhecidas (como HTML, HTTP). Não leva em conta a necessidade de se aprender a API específica para cada tecnologia.

[5] Em relação à aplicação *Windows*.

grande interação em tempo real no cliente (desenhos, por exemplo). O uso de aplicações independentes (sem usar browsers ou servidores HTTP) permite que se ofereçam mais serviços, interatividade e velocidade de acesso maior que applets e servlets, mas traz o inconveniente de requerer instalação por parte do cliente (o que é raro em aplicações Web, mesmo em intranets).

A tabela 5-2 compara tecnologias de objetos distribuídos. O cliente de uma aplicação distribuída pode ser qualquer uma das três interfaces do usuário comparadas na tabela 5.1. Essas tecnologias podem ser usadas na construção de camadas (*tiers*) adicionais entre o cliente e o servidor.

Tabela 5-2 – Quadro comparativo: tecnologias de objetos distribuídos e TCP/IP default.

Tecnologia de rede	Cliente (qualquer um) usando camada (*tier*) intermediária com tecnologia ...			
	Sockets TCP/IP	**RMI sobre JRMP**	**RMI sobre IIOP**	**CORBA sobre IIOP**
DESEMPENHO				
Taxa de transferência de dados[6]	Melhor possível.	Mais lento que TCP/IP.	Mais lento que RMI/JRMP.	Mesma ordem que RMI/IIOP.
DESENVOLVIMENTO, MANUTENÇÃO E REUSO				
Flexibilidade de desenvolvimento. Grau de flexibilidade: 1 a 5 (1 – nenhuma, 5 – total)	4	2	3	3
Facilidade de desenvolvimento[7] Grau de facilidade: 1 a 5 (1 – fácil a 5 – difícil)	5	3	3	4
Quantidade de código adicional a escrever (1 - pouco , 5 - bastante)	5	2	3	3
SUPORTE E DISPONIBILIDADE (PACOTES)				
Núcleo JDK 1.2 (Java 2)	java.net (núcleo)	java.rmi (núcleo)	javax.rmi (extensão)	org.omg (núcleo)
Núcleo JDK 1.1	java.net (núcleo)	java.rmi (núcleo)	-	org.omg (extensão)
Núcleo JDK 1.0	java.net (núcleo)	java.rmi (extensão)	-	-

Nos estudos de caso realizados no capítulo 2, comparamos tecnologias de objetos distribuídos com uma solução de conectividade usando TCP/IP (pacote `java.net`). Utilizar essa solução traz algumas vantagens: maior compatibilidade (por ser nativa ao núcleo da plataforma Java), flexibilidade (permite conectividade com agentes escritos em outras linguagens) e velocidade. A complexidade da programação em baixo nível e a baixa reutilização de código são mo-

---

[6] As limitações da rede utilizada no estudo de caso correspondente não permitiram que se apresentassem conclusões mais genéricas (e quantitativas) quanto ao desempenho. Veja mais detalhes no capítulo 2.

[7] Leva em consideração número de linhas de código adicionais (em relação a uma aplicação *standalone*) que precisam ser escritas, linguagens adicionais que precisam ser aprendidas (como IDL), necessidade de programar usando *threads*, necessidade de programar em baixo nível (conexões, *streams*, portas, etc.). Não leva em conta a necessidade de se aprender a API específica para cada tecnologia.

tivos que nos levam a procurar soluções de objetos distribuídos. Comparamos três tecnologias diferentes: RMI sobre JRMP (nativa desde o JDK1.1), CORBA sobre IIOP (nativa desde o JDK1.2[8]) e RMI sobre IIOP (extensão ao JDK1.2 ainda em beta).

No caso estudado, a tecnologia RMI sobre JRMP mostrou-se a mais eficiente das três. Também é a mais fácil de usar, pois não exige conhecimentos de rede (portas, soquetes, etc.) ou de outras linguagens (IDL), e é a que impõe menos alterações no código de uma aplicação *standalone* para torná-la uma aplicação distribuída. Por outro lado, é preciso que todas as partes da aplicação tenham sido escritas em Java, e que os objetos remotos permaneçam em locais definidos.

As vantagens em se usar CORBA ou RMI sobre IIOP está na possibilidade de integração da aplicação Java com outras aplicações ou objetos escritos em outras linguagens. Mas CORBA exige que o programador conheça IDL e saiba usar os métodos da API para realizar as transformações entre tipos de dados CORBA e Java. RMI esconde essa complexidade e, através do protocolo IIOP, permite que a aplicação se comporte de forma idêntica a uma aplicação CORBA. Nos casos estudados, as tecnologias CORBA e RMI sobre IIOP tiveram desempenho (quanto à taxa de transferência de dados) equivalente. Tal resultado é uma contribuição importante, apesar das limitações do experimento, pois reflete um aspecto de uma tecnologia lançada há pouco tempo. No caso analisado, o programador poderia desenvolver usando RMI sobre IIOP em vez de CORBA, obtendo o mesmo desempenho e a integração da arquitetura CORBA a um custo menor de desenvolvimento.

A última tabela (5.3) contém um sumário das diferenças entre as tecnologias CGI (usando C ou Perl) e Servlets.

Tabela 5-3 – Quadro comparativo: CGI x Servlets

Tipo de aplicação distribuída	**CGI**	**Servlets**
Tempo de resposta no caso estudado (servidor *Netscape/Windows95*).	A tecnologia CGI mostrou-se 2 vezes mais lenta que servlets em aplicação típica (*I/O bound*). Influência da linguagem utilizada foi baixa.	
Portabilidade	Depende da linguagem em que foi escrito o programa.	Roda em qualquer plataforma Java sem precisar recompilar.
Como lida com múltiplas requisições do browser	Abre múltiplos processos.	Inicia múltiplos *threads* dentro do processo do servidor Web.
Quando a aplicação é iniciada	Quando um cliente faz uma requisição.	Quando o servidor é inicializado ou quando um cliente faz a primeira requisição.
Servidores Web que suportam a tecnologia	Todos os servidores Web.	Suporte nativo em servidores da *Sun* e *W3C JigSaw*. Suporte através de plug-in (*JRun*) nos servidores mais populares.
Linguagens de programação suportadas	Perl, C, Visual Basic, Delphi, Shell (Unix), etc. Java e MS-DOS com restrições.	Java.

[8] Plataforma Java 2

A plataforma *Windows* não é a plataforma ideal para oferecer serviços Web a múltiplos clientes. Apesar disso, existem vários servidores para essa plataforma destinados ao uso em intranets. Todos suportam CGI e servlets através de *plug-in*. Devido aos recursos limitados do sistema, é importante que sejam usados da forma mais eficiente possível. Neste trabalho contribuímos com estudos de casos envolvendo aplicações Web construídas com a arquitetura CGI (a mais popular) e servlets, em um servidor popular do *Windows95*. Os resultados, onde verificamos um desempenho superior na arquitetura de servlets (mesmo quando a aplicação CGI foi construída usando uma linguagem mais eficiente como C), podem ser usados em estudos que procuram identificar causas de ineficiência em servidores Web no *Windows95*.

Lembramos, mais uma vez quanto às limitações desses experimentos. Eles não permitem tirar conclusões gerais, válidas para qualquer sistema ou plataforma. São estudos de casos: comparações que se aplicam a aplicações específicas rodando em uma plataforma específica (*Pentium/Windows 95*), portanto, as tabelas acima não devem ser usadas como referências definitivas sobre o desempenho de aplicações Java. Para uma maior segurança, sugerimos ao leitor que utilize os programas contidos no disquete anexo e os execute em sua própria plataforma, para que tenha dados mais precisos sobre o comportamento de cada tecnologia no seu ambiente de interesse.

Tempo de execução, tempo de resposta, taxa de transferência representam apenas *um* aspecto de uma tecnologia ou linguagem de programação. Outro aspecto é a adequação a determinadas tarefas. Em aplicações Web leves e de baixa utilização, usar CGI e Perl ainda é uma boa opção, principalmente quando já se tem uma grande base instalada nessa linguagem. Applets, embora tenham condições de apresentar uma interface do usuário muito mais próxima daquela utilizada na aplicação *Windows*, ainda são limitados por fatores como velocidade da rede (para *download* da aplicação), suporte dos browsers (ainda há falta de suporte a recursos como RMI e CORBA nos browsers mais novos) e principalmente restrições de segurança, o que os torna inadequados para várias tarefas. Servlets, nesses casos, podem oferecer a melhor solução em aplicações Web que applets.

5.1.1 Contribuições

As principais contribuições apresentadas por este trabalho são estudos de casos que permitem que se faça uma comparação entre tecnologias novas e pouco exploradas. A tecnologia RMI sobre IIOP, por exemplo, foi lançada como versão beta em Janeiro/1999. Os estu-

dos de caso, portanto, contribuem com informações novas e inéditas que podem servir de base para trabalhos futuros.

As aplicações apresentadas também constituem contribuições oferecidas pelo trabalho. A maior parte das aplicações são escritas em Java (rodam em todas as plataformas populares, sem necessitar recompilação[9]) o que permite que os resultados deste trabalho sejam verificados pelo leitor em outras situações, através da repetição dos experimentos.

Este trabalho também contribui com subsídios à escolha de tecnologias distribuídas em Java. Partindo dos estudos de casos, o desempenho e aplicabilidade de cada tecnologia em cenários diferentes são discutidos, oferecendo dados que permitem uma comparação entre tecnologias tendo em vista a adequação a tarefas específicas.

5.1.2 Dificuldades encontradas

Encontramos vários obstáculos durante os experimentos que realizamos até obter valores medidos com qualidade suficiente para permitir alguma análise.

Nos primeiros testes de transferência de dados entre aplicações usando tecnologias de objetos distribuídos (cap. 2), tentamos realizar os testes em uma única máquina, de forma a eliminar a necessidade de sincronizar relógios de duas máquinas, e evitar interferências devido a tráfego na rede. Encontramos, porém, um obstáculo maior e menos controlável: o maior consumo de recursos do sistema (já que tanto o cliente quanto o servidor consumiam os mesmos recursos limitados). Como resultado, ocorreram freqüentes alocações e esvaziamentos de páginas de memória pelo sistema nos últimos testes das aplicações CORBA e RMI, produzindo resultados erráticos com intervalos de confiança na faixa de 80 a 280%, e reduzindo a taxa de transferência nessas tecnologias. Os primeiros testes foram realizados em máquinas com 32 Mb de memória RAM e apresentaram resultados muito diferentes daqueles realizados em máquinas com 64Mb de RAM. Como algumas tecnologias consumiam mais recursos que outras, os valores relativos medidos eram afetados. A solução foi realizar os testes em rede, reduzindo esses efeitos (ainda presentes mas em menor escala). A diferença introduzida devido à imprecisão do sincronismo entre os relógios das máquinas não chegou a afetar os valores relativos, pois era a mesma para todas as aplicações[10]. Uma conclusão que pudemos obter de tudo isso

[9] Com exceção das aplicações CGI em C.

[10] Dentro de uma margem de erro de aproximadamente 50 ms (menor valor mensurável).

foi que as tecnologias RMI/IIOP e CORBA[11] podem apresentar um desempenho bem pior que aquele medido em nossos experimentos, caso sejam usados em aplicações (*Windows*) que façam maior uso dos recursos do sistema (por exemplo, aplicações gráficas).

Os testes utilizando CGI e servlets também precisaram ser refeitos diversas vezes por falta de um servidor na plataforma *Windows* que tivesse ao mesmo tempo um suporte eficiente a servlets e CGI. Começamos com o *O'Reilly WebSite Server*, um dos mais populares servidores *Windows*. As aplicações CGI funcionaram bem, mas o módulo *JRun* demorava até 15 segundos para responder (quando não deveria levar mais que 500ms). O outro servidor que tínhamos à disposição, *Netscape FastTrack Server*, travava quando recebia mais de 15 requisições CGI, porém integrava-se bem com o *JRun*. Decidimos, no final, realizar os testes neste último, por termos maior controle e poder limitar o número de acessos simultâneos a 15. Por causa dessas limitações, os resultados são insuficientes para que se possa apresentar conclusões genéricas, porém, representam uma situação que pode ocorrer em uma intranet já que *Windows95* e servidores *Netscape* são comuns nesse meio.

A precisão dos tempos medidos não foi melhor que 110ms devido a limitações da própria plataforma. Na comparação Java – C – Perl encontramos um obstáculo adicional nos recursos que essas linguagens e JavaScript ofereciam para medir tempo. Não havia uma sintaxe uniforme que sugerisse que os quatro programas faziam a mesma coisa. Em C, para obter um tempo equivalente ao medido em Java, tivemos que realizar cálculos adicionais. A nossa implementação do *Perl 5 para Windows* não possuía a função "`times`", que realiza medição de tempo com precisão de milissegundos. Tivemos, portanto, que usar a função "`time`", que só mede segundos inteiros e repetir cada teste 10 vezes, para obter uma média com uma precisão de centenas de milissegundos.

Como a maior parte dos trabalhos de mestrado, o trabalho apresentado aqui corresponde a bem menos do que foi realizado. Para desenvolver programas de comportamento equivalente usando todas as tecnologias exploradas foi necessário uma pesquisa exaustiva, e acúmulo de experiências através da criação de aplicações que utilizavam cada tecnologia. Foi necessário também conhecer bem as técnicas de orientação de objetos para isolar, da forma mais eficiente possível, as partes das aplicações que eram iguais em todas as aplicações, daquelas que eram

---

[11] Essas limitações referem-se à API analisada, nativa do JDK. APIs de outros fabricantes podem apresentar resultados diferentes.

diferentes, por causa da tecnologia que utilizavam. Um dos resultados desse estudo e pesquisa foi um tutorial de quase 200 páginas, detalhando cada tecnologia (CORBA, RMI, TCP/IP, Servlets, etc.) e ilustrando o processo de desenvolvimento das aplicações em cada uma. Inicialmente, o tutorial *era* a dissertação, mas, à medida em que avançamos e focalizamos nos objetivos centrais do trabalho, o tutorial foi caminhando para o apêndice até que finalmente resolvemos deixá-lo de fora, pois não mais se referia ao objetivo central deste trabalho. Pretendemos, porém, reorganizar o tutorial para dar-lhe a forma de um livro.

5.2 Sugestões para trabalhos futuros

Este trabalho explorou as principais tecnologias de rede utilizadas pela linguagem Java, e as comparou entre si e com outras soluções do mercado. Procuramos ao máximo restringir-nos a tecnologias nativas, embutidas no JDK/JRE, ou distribuídas como extensões padrão à plataforma Java. Várias outras soluções de terceiros existem, são bastante populares e não foram abordados aqui. Sendo soluções Java, os programas utilizados neste trabalho podem ser adaptados de forma a utilizar uma dessas tecnologias, e o mesmo procedimento de testes pode ser aplicado obtendo uma análise do desempenho da tecnologia em questão. Uma comparação entre as implementações comerciais de ORBs (Orbix, Visibroker) entre si, seria uma possível aplicação.

Uma outra proposta seria analisar o desempenho de aplicações Java usando objetos distribuídos em linguagens diferentes. Mencionamos o fato de que CORBA permite que um programa escrito em Java possa utilizar objetos escritos em C, C++, Delphi ou COBOL, mas não demonstramos tal aplicação. Para realizar tal teste com a estrutura usada no capítulo 2, precisaríamos de um ORB na linguagem nativa (C, por exemplo). Depois, é preciso compilar a IDL `bench.idl` (localizada nos arquivos do disquete, em `/jad/apps3/`) e gerar um servidor nessa outra linguagem. Tal trabalho poderia demonstrar a capacidade de integração de sistemas usando CORBA, e se o desempenho de uma aplicação Java poderia melhorar, caso utilizasse um objeto remoto escrito em linguagem mais eficiente.

Várias publicações mencionam que aplicações Web usando CGI são menos eficientes que Servlets, mas encontramos poucas dispostas a fazer o teste. Fizemos o teste e comprovamos que isto é verdade no *Netscape FastTrack Server 2.0* for *Windows 95*. Esse servidor, porém, não é o mais popular, nem *Windows95* é a plataforma ideal para hospedar um servidor Web que irá receber muitas requisições simultâneas. De acordo com [NETC 99], em uma pesquisa reali-

zada em abril de 1999 em mais de 5 milhões de sites no mundo, o servidor Web mais popular é o *Apache*, usado em 56% dos sites, seguido por 23% dos sites usando *Microsoft IIS/PWS*. A grande parte dos servidores *Apache* roda em *Linux*, e os servidores *Microsoft* em *NT*. Essas seriam, portanto, plataformas alvo onde a metodologia e aplicações aqui apresentadas para comparar CGI e servlets poderiam ser utilizadas para obter uma análise de grande interesse, uma vez que CGI é uma tecnologia nativa dos servidores *Apache*, e, servlets, que no nosso experimento se mostraram mais eficientes, são suportados nesses servidores através de módulos *JRun*.

Apêndice A

Este apêndice fornece detalhes relacionados à estrutura e código Java das aplicações utilizadas para realizar os experimentos do capítulo 2. Todo o código fonte pode ser encontrado no subdiretório `/jad/apps3/` do arquivo compactado no disquete que acompanha esta dissertação.

A.1 Diagramas de objetos (UML e Java)

A figura A-1 (página inteira) ilustra os pacotes e subpacotes Java de toda a suite de aplicações. Cada pacote contém classes e interfaces Java que realizam tarefas relacionadas entre si. Todas as aplicações compartilham as classes do pacote `bench`, que é utilizado pelas camadas de apresentação (interface gráfica) tanto do cliente como do servidor de cada aplicação.

Os subpacotes de `bench.client` contêm as classes usadas pelas aplicações cliente e os subpacotes de `bench.server` contêm as classes usadas pelas aplicações servidoras. Todos os clientes também importam interfaces e/ou *stubs* que foram mantidas em seus respectivos subpacotes de `bench.server`.

A arquitetura em camadas de todas as aplicações obedece ao diagrama da figura A-2. As pastas ilustradas representam pacotes Java e dividem a aplicação em camadas. Qualquer par de aplicações sempre utiliza classes do pacote `bench` (camada de apresentação para o cliente e o servidor), de um subpacote de `bench.client` (lógica da aplicação cliente) e de um subpacote de `bench.server` (lógica da aplicação servidora).

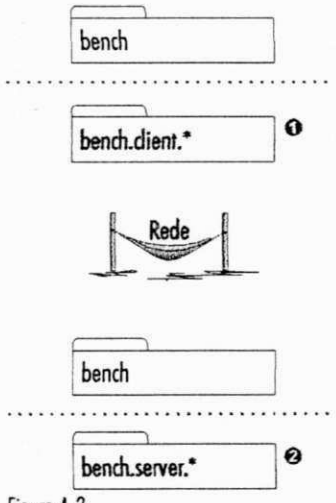


Figura A-2

As classes `bench.server.*.Server` estendem a classe abstrata `bench.ServerFrame`, que implementa a infraestrutura necessária para que suas subclasses se apresentem como aplicações gráficas de um ambiente de janelas. O principal método a ser implementado pelas classes `bench.server.*.Server` é o método `init()` que realiza a inicialização do servidor de acordo com a sua arquitetura específica (CORBA, RMI, TCP/IP).

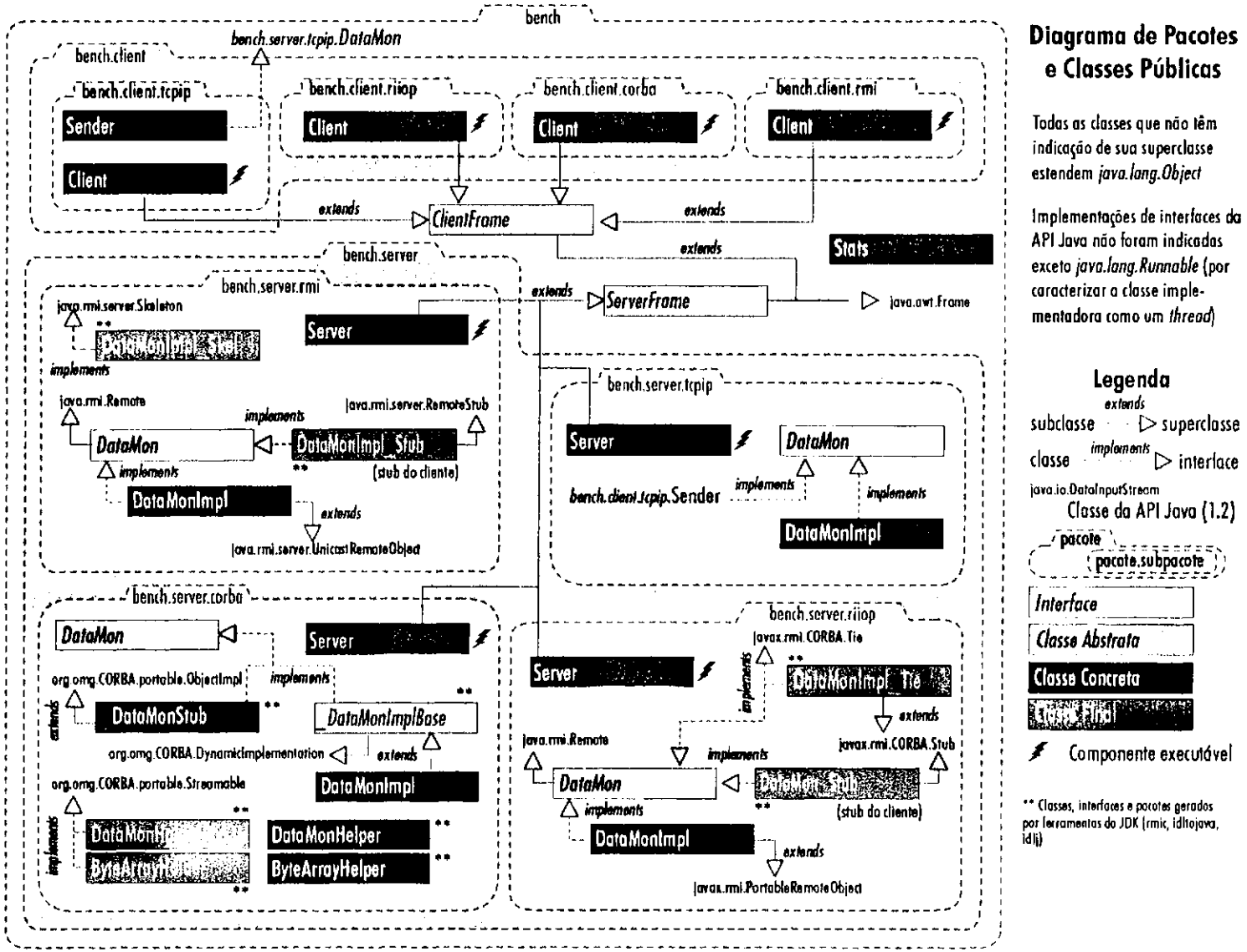


Diagrama de Pacotes e Classes Públicas

Todas as classes que não têm indicação de sua superclasse estendem *java.lang.Object*

Implementações de interfaces da API Java não foram indicadas exceto *java.lang.Runnable* (por caracterizar a classe implementadora como um *thread*)

Legenda

** Classes, interfaces e pacotes gerados por ferramentas do JDK (rmic, idltojava, idlj)

Figura A-1 - Diagrama de classes públicas da aplicação 1 "bench". Classes de uso local e da API Java foram omitidas. Somente as relações de herança e polimorfismo estão mostradas.

De forma semelhante, as classes `bench.client.*.Client` estendem a classe abstrata `bench.ClientFrame`, que fornece a interface gráfica padrão para todos os clientes. Também constrói os vetores a serem enviados e chama os métodos abstratos implementados em cada classe `bench.client.*.Client`. O método `connect()` (veja código-fonte), que deve ser implementado em cada subclasse, realiza a conexão ao servidor remoto.

Essa organização em camadas permite isolar as particularidades de cada arquitetura em poucas classes bem definidas, para que se possa ter um maior controle sobre a qualidade do código – essencial para garantir programas equivalentes.

Os nomes das classes contidos em cada pacote de servidor ou cliente são os mesmos nomes usados nas classes de outros pacotes que realizam tarefas semelhantes. Além das classes `bench.server.*.Server` e `bench.client.*.Client`, todas as aplicações têm:

- **`bench.server.*.DataMon`** – interface Java que contém a assinatura dos métodos remotos. Na aplicação CORBA esta interface foi gerada automaticamente a partir de uma interface IDL. Nas outras aplicações, ela teve que ser construída em Java. Esta interface possui um papel diferente na aplicação TCP/IP onde não há realmente chamadas de métodos em objetos remotos.
- **`bench.server.*.DataMonImpl`** – implementação da interface `DataMon` e do objeto remoto nas aplicações CORBA e RMI. Na aplicação CORBA esta classe estende o esqueleto `_DataMonImplBase`. Nas aplicações RMI estende `UnicastRemoteObject` (JRMP) ou `PortableRemoteObject` (IIOP). Na aplicação TCP/IP implementa os métodos da interface `DataMon` e é usada para responder a requisições do cliente.

A figura A-3 ilustra a arquitetura em camadas da aplicação TCP/IP, mostrando os detalhes referentes à interação entre as classes. As outras aplicações possuem a mesma estrutura de

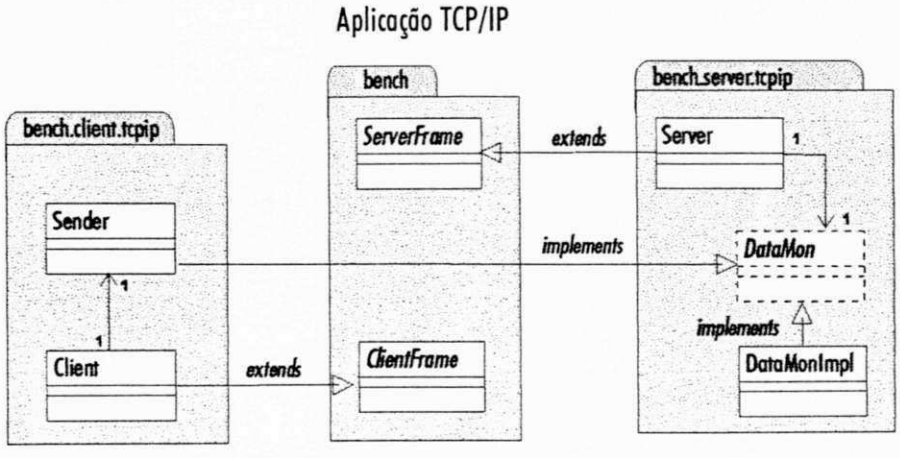


Figura A-3 – Diagrama de aplicação TCP/IP (cliente e servidor)

pacotes (veja a figura A-1). Todas se comportam da maneira representada pelo diagrama em camadas da figura A-2.

A.2 Construção das aplicações cliente e servidor

Esta seção apresenta detalhes do código Java usado na construção das aplicações. O código apresentado corresponde a trechos de listagens completas que podem ser encontradas no subdiretório `/jad/apps3/` (resultante da expansão dos arquivos do disquete).

A.2.1 Estrutura de todas as aplicações cliente

Os objetos a serem enviados (vetores de *bytes*) são criados logo após a conexão ao servidor, no método `actionPerformed()`, que é chamado quando o usuário aperta o botão para "Enviar dados". Primeiro os vetores são criados. Depois são enviados chamando o método `enviarObjetos()`:

```
connect();

byte[][] objetos = new byte[14][];
int i = 0;
for (; i < 10; i++) {
   int bytes = 100000 * (i+1);
   ta.append("Alocando byte[" + bytes + "] \n");
   objetos[i] = new byte[bytes];
}
for (; i < objetos.length; i++) {
   int bytes = (1000000 * (i-8));
   ta.append("Alocando byte[" + bytes + "] \n");
   objetos[i] = new byte[bytes];
}
int numVezes = Integer.parseInt(ch.getSelectedItem());
enviarObjetos(objetos, numVezes);
```

Os métodos abstratos `enviarObjetos()` e `print()` estão implementados em cada `bench.client.*.Client`, e têm praticamente a mesma forma. O método `connect()` também é implementado em cada classe `Client`, mas contém código específico para cada tecnologia utilizada.

A.2.2 Estrutura de todas as aplicações servidoras

A interface remota da aplicação — definida em Java ou IDL, dependendo da tecnologia utilizada — consiste de dois métodos e é implementada em todos os servidores,. Um método (`print()`) é utilizado apenas para imprimir os relatórios no servidor a partir de um cliente. O

outro método (`envia()`) é o mais importante. É utilizado para enviar os dados. Este método requer três argumentos, que são:

- vetor de bytes (`byte[]`) contendo os dados
- inteiro (`int`) contendo o tamanho do vetor
- inteiro longo (`long`) contendo o tempo de envio

A interface da aplicação é implementada e utilizada de maneira diferente pelas aplicações CORBA, TCP/IP e RMI. Na aplicação CORBA, ela é totalmente especificada em linguagem IDL. Na aplicação TCP/IP ela é definida por um protocolo de comunicações baseado em códigos inteiros (para identificar qual dos comandos o cliente enviou: imprimir ou enviar dados) e um fluxo de dados (`DataOutputStream`) para enviar parâmetros. Nas aplicações RMI, a interface remota é definida a partir de uma interface Java.

O método `envia()` (classe `DataMonImpl`) foi implementado para, além de receber os dados, montar uma coleção com os mesmos para posterior impressão de um relatório. A classe `DataMonImpl` é implementada de forma quase idêntica nas quatro aplicações. A *única* diferença está nas superclasses e interfaces que são diferentes dependendo da tecnologia empregada. A listagem abaixo[1] mostra a implementação de `bench.server.corba.DataMonImpl`:

```
package bench.server.corba;

import java.util.*;
import java.io.*;
import bench.Stats;

public class DataMonImpl extends _DataMonImplBase {

    private Hashtable table;

    DataMonImpl() {
      table = new Hashtable(10); // cria coleção p/ armazenar dados
    }
    public synchronized void envia (long inicio, int tam, byte[] bytes) {
        long agora = System.currentTimeMillis();  // instante atual
        long intervalo = agora - inicio;
        // armazena os dados em Hashtable e imprime a ocorrencia
        System.out.println(Stats.armazena(table, intervalo, bytes.length));
    }

    public void print() {       // imprime relatorio
      Stats.print(table);
    }
}
```

[1] A versão disponível no código fonte em disquete poderá apresentar linhas de código não mostradas abaixo.

A implementação acima representa um objeto remoto. No caso de CORBA, uma outra classe (a classe `bench.server.corba.Server`) precisa criar uma instância desse objeto e registrá-lo no ORB e no sistema de nomes.

A classe `bench.Stats` é uma classe utilitária que possui dois métodos estáticos: `print(Hashtable)` e `armazena(Hashtable, long, int)`. O primeiro método ordena e imprime o conteúdo de uma coleção de dados passada como parâmetro. O segundo método é chamado pelas implementações de `DataMon.envia()` e armazena os valores medidos em cada transferência.

O cliente CORBA precisa saber o nome do objeto e o endereço de um serviço de nomes capaz de localizá-lo. Não precisa saber onde está o objeto nem em que linguagem foi escrito. Localizando-se o objeto, pode-se obter uma referência remota para ele e chamar o método `envia()` (ou qualquer outro definido no IDL) como se fosse local:

```
dataRef.envia(System.currentTimeMillis(), byteObject.length, byteObject);
```

No trecho acima, `dataRef` é uma referência do tipo `bench.server.corba.DataMon`.

A.3 Detalhes de implementação característicos de cada tecnologia

A.3.1 Aplicação CORBA

O primeiro passo para construir uma aplicação CORBA é especificar a sua interface IDL. No nosso exemplo (arquivo `bench.idl`), ela consiste de duas operações: `envia()` e `print()`.

```
module bench {
  module server {
    module corba {

        typedef sequence <octet> ByteArray;

        interface DataMon {

              void envia (in long long quando,
                     in long tamanho,
                     in ByteArray bytes);

              void print();
        };

  };};};
```

As palavras em negrito são reservadas em OMG IDL. Esta interface pode ser implementada em qualquer linguagem que possua um mapeamento IDL como especificado em

[OMG98], sendo possível realizar experimentos para medir o desempenho de aplicações Java contendo objetos escritos em C, COBOL ou SmallTalk, por exemplo. Para implementar o IDL em Java, utilizamos a ferramenta `idlj` distribuída com o Java 2. A execução de

```
idlj -fserver -fclient bench.idl
```

cria uma coleção de arquivos `*.java` em um subdiretório `bench/server/corba`, pertencentes a um pacote `bench.server.corba`. Entre os arquivos gerados estão:

- **`DataMon.java`** – interface Java análoga à interface IDL que a gerou. Contém o código Java a seguir:

```
package bench.server.corba;
public interface DataMon
{
      void envia (long quando, int tamanho, byte[] bytes);
      void print ();
}
```

- **`_DataMonImplBase.java`** – classe abstrata e base para implementação do servidor que age como esqueleto no modelo CORBA desta aplicação
- **`_DataMonStub.java`** – *stub* do cliente que intercepta as requisições enviadas.

É preciso criar uma classe servente, estendendo `_DataMonImplBase` e implementando todos os seus métodos abstratos, que são os métodos da interface `DataMon`. Essa classe será usada pelo servidor para construir objetos e exportá-los através do ORB, para que clientes possam chamar seus métodos remotamente. Uma listagem desta classe foi mostrada na seção anterior. Veja o código-fonte com a implementação do servidor no disquete.

A.3.2 Aplicações RMI (usando JRMP e IIOP)

Em RMI, é preciso especificar uma interface `java.rmi.Remote` para declarar os métodos que irão compor a interface remota. Todos os métodos precisam declarar que provocam a exceção `java.rmi.RemoteException`. A interface usada está listada abaixo.

```
package bench.server.rmi;
import java.rmi.*;      //Remote e RemoteException

public interface DataMon extends Remote {

   public void envia (long inicio, int tam, byte[] obj) throws RemoteException;
   public void print() throws RemoteException;

}
```

Esta interface é utilizada tanto pela implementação usando JRMP como pela implementação que usa IIOP como protocolo de comunicação. Não há grandes diferenças nos objetos remotos. A única diferença entre a implementação CORBA (mostrada acima) e as implementações do objeto em RMI/JRMP e RMI/IIOP é a superclasse que estendem. Existe ainda a restrição imposta pela arquitetura RMI que todos os métodos e construtores devem declarar que provocam `java.rmi.RemoteException`. O conteúdo dos métodos `print()`, `envia()` e do construtor da classe são idênticos.

Na versão RMI/JRMP, a implementação de `DataMonImpl` começa da forma:

```
package bench.server.rmi;
public class DataMonImpl
        extends java.rmi.server.UnicastRemoteObject
        implements bench.server.rmi.DataMon {
```

Compare com a versão RMI/IIOP (as diferenças estão marcadas em negrito):

```
package bench.server.riiop;
public class DataMonImpl
        extends javax.rmi.PortableRemoteObject
        implements bench.server.riiop.DataMon {
```

A implementação do servidor que criará instâncias das classes acima e as exportará como objetos remotos difere bastante nas versões RMI/JRMP e RMI/IIOP pois o processo de registrar um objeto no *RMI Registry* é bem mais simples que o processo de registrá-lo em um ORB através do serviço de nomes (`tnameserv`). Veja as diferenças no código-fonte.

O cliente RMI (JRMP), precisa saber o nome do objeto e o endereço da máquina onde reside. O nome é resolvido pelo *RMI Registry* e uma referência a um objeto é retornada. *O RMI Registry* tem que estar rodando nas máquinas que possuem objetos remotos mas não precisa rodar nas máquinas cliente. Se o cliente receber como parâmetro algum tipo desconhecido, ele tentará fazer o *download* da classe correspondente no servidor. Localizando-se o objeto, pode-se obter uma referência remota para ele e chamar o método `envia()` (ou qualquer outro definido na interface `bench.server.rmi.DataMon`) como se fosse local.

O método `connect()` da classe `bench.server.rmi.Client` inicializa a conexão do cliente obtendo uma referência através do serviço de nomes do *RMI Registry*. Na classe `bench.server.riiop.Client` obtém uma referência através do mesmo serviço de nomes do JRE, usado na aplicação CORBA.

A.3.3 Aplicação TCP/IP

A implementação usando soquetes não constrói verdadeiros objetos remotos. A chamada dos métodos remotos no cliente foi realizada indiretamente através de uma classe adaptadora (`bench.client.tcpip.Sender`) que implementa a interface remota e traduz as chamadas de métodos em instruções de um protocolo proprietário. As instruções são enviadas através de um soquete aberto na conexão do cliente.

Usando esta adaptadora, o cliente age como se estivesse fazendo a invocação remota de forma semelhante a RMI e CORBA, mas o objeto `DataMon` criado através de `Sender` é local. Abaixo está a implementação do método `envia` em `Sender`:

```
public void envia (long inicio, int tam, byte[] bytes)
                                                    throws IOException {
        out.writeInt(SEND);       // int SEND = 100
        out.writeLong(inicio);
        out.writeInt(tam);
        out.write(bytes);
}
```

O trecho a seguir mostra como o método acima é chamado na classe `Client` (`dataRef` é um objeto do tipo `DataMon` implementado em `Sender`):

```
for (int j = 0; j < objetos.length; j++) {
   byte[] byteObject = objetos[j];
   dataRef.envia(System.currentTimeMillis(),
                 byteObject.length,
                 byteObject);
}
```

Como nesta aplicação não há registro ou busca de objetos remotos, não é necessário haver um serviço de nomes rodando a não ser o serviço DNS para localizar endereços IP através de nomes. Os clientes precisam saber em que máquina está o servidor e em que porta roda o serviço. A comunicação entre os clientes e o servidor é iniciada a partir da porta 1999.

O método `connect()` da classe `bench.server.tcpip.Client` tenta abrir uma conexão com a máquina servidora através de sua porta 1999. Conseguindo, instancia um objeto para tratar os dados (tipo `Sender`).

No servidor, as instruções são decodificadas e os métodos correspondentes são chamados. Dependendo da instrução recebida, o servidor continua a escutar o soquete para receber os dados que irá converter nos parâmetros do método.

As instruções que compõem o protocolo são representadas por números inteiros definidos como constantes (`SEND`, `EXIT` e `PRINT`) na interface `DataMon`. Os formatos aceitos são:

`100 <long> <int> <byte[]>`	(chama `envia()` com seus três parâmetros)
`200`	(chama `close()` para que servidor feche a conexão)
`300`	(chama `print()` para imprimir o relatório)

O cliente envia os dados utilizando os métodos da classe `DataOutputStream`, que permitem o envio de tipos primitivos. O servidor empacota o fluxo de dados recebido em um `DataInputStream` e usa os métodos `readLong()`, `readInt()` e `readFully(byte[])` para recuperar os dados enviados.

Ao receber o comando `100`, o servidor lê o próximo `long` (instante do envio), o próximo `int` (tamanho dos bytes) e um vetor de bytes de tamanho fornecido pelo parâmetro anterior. Com esta informação, é capaz de construir uma chamada para o método remoto.

O trecho de código a seguir mostra a rotina do *thread* (método `run()`) no servidor que é responsável por redirecionar as instruções para os seus métodos:

```
int comando = -1;
   while ((comando = in.readInt()) != -1) {
      switch (comando) {
         case CLOSE:
            close();
         break;

         case SEND:
            long inicio = in.readLong();
            int tam = in.readInt();
            byte[] received = new byte[tam];
            in.readFully(received);
            envia(inicio, tam, received);
         break;

         case PRINT:
            print();
         break;
      }
   }
```

Os métodos `close()`, `print()` e `envia()` além do método `run()` que contém o código acima são implementados na classe `bench.server.tcpip.DataMonImpl`. O construtor e os métodos `print()` e `envia()` têm a mesma sintaxe que as implementações CORBA e RMI.

Para maiores detalhes, consulte o código fonte e documentação da aplicação em `/jad/apps3/`.

Apêndice B

Este apêndice apresentará alguns detalhes referente à construção das aplicações utilizadas nos experimentos do capítulo 3. Os detalhes referem-se principalmente ao JavaScript e páginas HTML. São também listados fragmentos que contêm a essência do código dos programas. Não é objetivo deste apêndice, porém, abordar programação em Perl, JavaScript, desenvolvimento de aplicações Web com CGI ou desenvolvimento de *servlets*.

B.1 Página HTML e JavaScript

As duas aplicações analisadas neste capítulo são iniciadas através de uma página HTML cujo código está mostrado abaixo. Essa página HTML contém código JavaScript embutido (mostrado em itálico) entre os descritores HTML `<script>` e `</script>` e nos atributos `onclick` dos descritores de formulário `<input type=button>` que constróem botões em HTML. O atributo HTML `onclick` contém código que será executado quando ocorrer o evento de clicar no botão correspondente.

O primeiro bloco `<script>...</script>` situa-se no cabeçalho do documento HTML (`<head>...</head>`) e contém a definição das funções `getCookie()` e `enviar()` que serão chamadas dentro da página. Quando a página é carregada, essas funções são lidas e ficam na memória, podendo ser chamadas enquanto a página estiver ocupando a janela do browser.

```
<html><head>
<script>
function getCookie(nome) {
    cookies = document.cookie;
    loc = cookies.indexOf("tempo=");
    igual = cookies.indexOf("=", loc);
    fim = cookies.indexOf(";", loc);
    if (fim == -1) fim = cookies.length;
    return cookies.substring(igual+1, fim);
}
```

```
function enviar(f) {
    document.cookie = "tempo" + "=" + (new Date()).getTime();
    f.submit();
}
</script>
</head>
```

No corpo do documento há outro bloco `<script>...</script>`. Neste caso não há funções e o código é interpretado linha-por-linha quando a página é carregada. O valor do *cookie* chamado `tempo` é lido (pode ser `NaN`, se o *cookie* ainda não existir). Ele contém o instante em que o botão foi apertado. O código seguinte subtrai o valor obtido do instante atual e imprime o resultado (`intervalo`) dentro de um campo de texto no início da página.

```
<body>
<script>
agora = (new Date()).getTime();      // obtem instante atual
antes = parseInt(getCookie("tempo")); // obtem cookie chamado 'tempo'
intervalo = agora - antes;            // calcula intervalo
// imprime HTML na página com caixa de texto contendo intervalo
document.write("<form><input type=text size=7 value=" + intervalo + "></form>");
<script>
```

O segundo formulário da página contém um botão que, ao ser apertado, faz com que o browser interprete o código contido no seu atributo `onclick`, chamando a função `enviar()` (acima) que grava um *cookie* com o instante atual e envia a requisição.

```
<FORM action="/servlet/Simple" method="POST">
<INPUT type=button value="Visualizar"
       onclick="enviar(this.form)">
</FORM>
</BODY></HTML>
```

Depois que o CGI ou *servlet* recebem a requisição, geram uma nova página com o mesmo código HTML e JavaScript acima que, quando for interpretada pelo browser, irá ler o *cookie* gravado e mostrar o tempo de resposta na tela.

B.2 Aplicações

As listagens a seguir mostram, na íntegra, o código de três aplicações `Simples` usando CGI em C, CGI em Perl e Servlet em Java. Também mostra as aplicações `Combina`, omitindo as partes que são idênticas à aplicação `Simples`. A maior parte do código é geração de HTML que é igual em todas as aplicações. A parte em negrito destaca as diferenças.

B.2.1 Java Servlet

```
/** Aplicação Simples.java */
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Simples extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
                                                throws ServletException {
        PrintWriter out = null;
        String arquivo = "i:\\jad\\apps4\\cgic\\bigfile.txt";
        try {
          out = res.getWriter();
          // cabeçalho HTTP
          res.setContentType("text/html");
          // página HTML
          out.println("<html>\n<head>\n");
          out.println("<script>" );
          out.println("function getCookie(nome) {"  );
          out.println("    cookies = document.cookie;"   );
          out.println("    loc = cookies.indexOf(\"tempo=\");"   );
          out.println("    igual = cookies.indexOf(\"=\", loc);" );
          out.println("    fim = cookies.indexOf(\";\", loc);"  );
          out.println("    if (fim == -1) fim = cookies.length;" );
          out.println("    return cookies.substring(igual+1, fim);"  );
          out.println("}" );
          out.println("");
          out.println("function enviar(f) {" );
          out.println("    document.cookie = \"tempo\"
                      + \"=\" + (new Date()).getTime();");
```

```
            out.println("    f.submit(); ");
            out.println("}");
            out.println("</script>");

            out.println("</head>\n<body>");
            BufferedReader fr = new BufferedReader(new FileReader(arquivo));
            String st = null;
            while((st = fr.readLine()) != null) {
                out.print(st);
            }
            out.flush();
            out.println("<script>");
            out.println("agora = (new Date()).getTime();");
            out.println("antes = parseInt(getCookie(\"tempo\"));");
            out.println("intervalo = agora - antes;");
            out.println("document.write(\"<form><input type=text size=7 value=\"
                        + intervalo + \"></form>\");" );
            out.println("</script>");
            out.println("<form action=\"/servlet/Simple\" method=\"GET\">");
            out.println("<input type=button value=\"Visualizar\" "
                      + "onclick=\"enviar(this.form)\">");
            out.println("</form>");
            out.println("</body>\n</html>");
        } catch (IOException e) {
            out.println("<html><body><h1>500 IOException</h1>");
            out.println("<p>Aconteceu um erro de E/S. </body></html>");
        }
        out.close();
    }
} // fim da classe Simples ----------------------------------------------
```

```
/** Aplicação Combina.java */
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Combina extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
                                                throws ServletException {
    PrintWriter out = null;
      try {
      // (... esta parte é idêntica à aplicação Simples ...)
         out.println("</head>\n<body><p>");
         for (int x = 0; x < 100; x++) {
             if ((x % 10) == 0) out.print(". ");
             for (int k = 0; k < x; k++) {
                double comb = fatorial(x) / (fatorial(x - k) * fatorial(k));
             }
         }
         out.println();
      //  (... esta parte é praticamente idêntica à aplicação Simples ...)
 }
 // método fatorial
 private double fatorial(double n) {
       if (n == 1 || n == 0) {
            return 1;
       } else {
           return fatorial(n - 1) * n;
       }
 }
} // fim da classe Combina ----------------------------------------
```

B.2.2 CGI em C

```
/* Programa Simples.c*/
#include <stdio.h>

int main(void) {
  FILE *pFile = NULL;
  char str[80];

  /* cabeçalho HTTP */
  printf("Content-type: text/html\n\n");
  /* página HTML  */
  printf("<html>\n<head>\n");
  printf("<script>");
  printf("function getCookie(nome) {");
  printf("    cookies = document.cookie;");
  printf("    loc = cookies.indexOf(\"tempo=\");");
  printf("    igual = cookies.indexOf(\"=\", loc);");
  printf("    fim = cookies.indexOf(\");\", loc);");
  printf("    if (fim == -1) fim = cookies.length;");
  printf("    return cookies.substring(igual+1, fim);");
  printf("}");
  printf("");
  printf("function enviar(f) {");
  printf("    document.cookie = \"tempo\" + \"=\" + (new Date()).getTime();");
  printf("    f.submit(); ");
  printf("}");
  printf("</script>");
  printf("</head>\n<body>");
  pFile = fopen("bigfile.txt", "r");
  while (fgets(str, 8, pFile) != NULL) {
        printf("%s", str);
  }
  fclose(pFile);
  printf("<script>");
  printf("agora = (new Date()).getTime();");
  printf("antes = parseInt(getCookie(\"tempo\"));");
  printf("intervalo = agora - antes;");
```

```
  printf("document.write(\"<form><input type=text size=7 value=\"
                                          intervalo + \"></form>\");");
  printf("</script>");
  printf("<form action=\"/cgi-bin/Simple.exe\" method=\"GET\">");
  printf("<input type=button value=\"Visualizar\""
                                      "onclick=\"enviar(this.form)\">");
  printf("</form>");
  printf("</body>\n</html>\n");
} /* ------------------------------------------------------------- */


/* Programa Combina.c */
#include <math.h>

// função fatorial
double fatorial(double n) {
   if (n == 1 || n == 0) {
       return 1;
   } else {
    return fatorial(n - 1) * n;
   }
}

int main(void) {
  int x, k, y;
  double comb, q;
//  (... esta parte é idêntica à aplicação Simples ...)
    printf("</head>\n<body><p>");
    for (x = 0; x < 100; x++) {
        if ((x % 10) == 0) printf(". ");
        for (k = 0; k < x; k++) {
              comb = fatorial(x) / (fatorial(x - k) * fatorial(k));
              q = sqrt(comb);
        }
    }
    printf("\n");
//  (... esta parte é praticamente idêntica à aplicação Simples ...)
}
```

B.2.3 CGI em Perl

```
# Aplicação Simples.pl
&main;
sub main {
    # cabeçalho HTTP
    print "Content-type: text/html\n\n";
    # página HTML
    print "<html>\n<head>\n";
    print "<script>";
    print "function getCookie(nome) {";
    print "    cookies = document.cookie;";
    print "    loc = cookies.indexOf(\"tempo=\");";
    print "    igual = cookies.indexOf(\"=\", loc);";
    print "    fim = cookies.indexOf(\";\", loc);";
    print "    if (fim == -1) fim = cookies.length;";
    print "    return cookies.substring(igual+1, fim);";
    print "}";
    print "";
    print "function enviar(f) {";
    print "    document.cookie = \"tempo\" + \"=\" + (new Date()).getTime();";
    print "    f.submit(); ";
    print "}";
    print "</script>";
    print "</head>\n<body>";
    if (open (FILE, "<" . "i:\\jad\\apps4\\cgipl\\bigfile.txt")) {
       while($linha = <FILE>) {
            print $linha;
       }
    } else {
       print "Erro ao ler arquivo!";
    }
    print "<script>";
    print "agora = (new Date()).getTime();";
    print "antes = parseInt(getCookie(\"tempo\"));";
    print "intervalo = agora - antes;";
    print "document.write(\"<form><input type=text size=7 value=\"
                                    + intervalo + \"></form>\");";
    print "</script>";
```

```
    print "<form action=\"/cgi-shl/Simple.pl\" method=\"GET\">";
    print "<input type=button value=\"Visualizar\"
                                        onclick=\"enviar(this.form)\">";
    print "</form>";
    print "</body>\n</html>\n";
    close(STDOUT);
}
# ---------------------------------------------------------------------


# Aplicação Combina.pl
&main;
sub main {
#  (... esta parte é idêntica à aplicação Simples ...)
   for ($x = 0; $x < 100; $x++) {
     if (($x % 10) == 0) {print(". ");}
       for ($k = 0; $k < $x; $k++) {
             $comb = fatorial($x) / (fatorial($x - $k) * fatorial($k));
       #print("C\($x\, $k\) = $comb\<br\>");
       }
    }
    print("\n");
#  (... esta parte é praticamente idêntica à aplicação Simples ...)
}
sub fatorial {
   local($n) = @_;
   if ($n == 1 || $n == 0) {
       1;
   } else {
       &fatorial($n - 1) * $n;
   }
}
```

Apêndice C

Este apêndice fornece alguns detalhes sobre a estrutura da aplicação descrita no capítulo 4. Maiores detalhes sobre o código em Java podem ser encontrados no diretório `/jad/apps/` (resultante da expansão dos arquivos do disquete) no código-fonte, detalhadamente comentado (`/jad/apps/bancodados/`) e na documentação em hipertexto, gerada a partir do código-fonte, que descreve todos os pacotes, classes e métodos (`/jad/apps/docs/`).

Este apêndice supõe que o leitor esteja familiarizado com a linguagem Java [ARNO96] [CAMP99] [CORN97], JDBC [REES97], HTML e JavaScript [ROCH99].

C.1 Estrutura da aplicação

A aplicação estudada no capítulo 4 permite o acesso a um banco de dados contendo registros de *anúncios*. Fornece diversas interfaces do usuário e opções de servidores intermediários. O núcleo da aplicação, porém, consiste apenas de dois *tipos* Java:

- `bancodados.BancoDados`: uma interface que define os métodos utilizados por qualquer objeto que implemente serviços de acesso ao banco de dados. Representa o banco de dados ou quadro de avisos (do sistema de anúncios).
- `bancodados.Registro`: classe que representa um registro do banco de dados (ou um anúncio).

Estes dois tipos estão armazenados em um pacote chamado `bancodados`. A figura C-1 ilustra os diagramas da classe `Registro` e interface `BancoDados` e seus métodos.

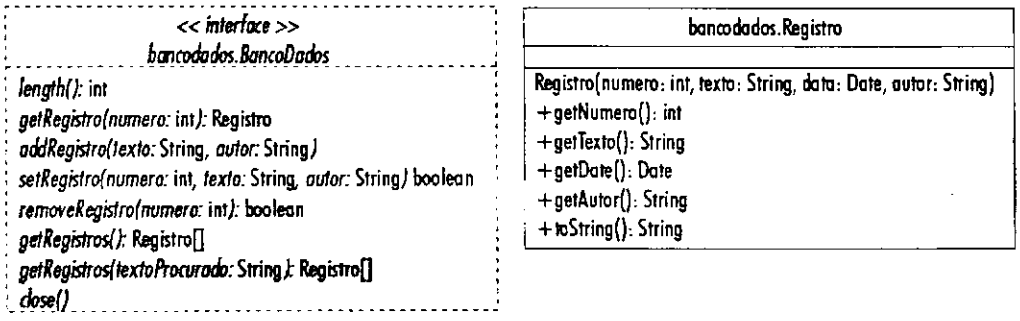


Figura C-1 Diagramas de BancoDados e Registro

A interface `BancoDados` é utilizada por classes que implementam a interface do usuário. Através dela, todas as interfaces do usuário podem chamar métodos para realizar operações em um banco de dados sem precisar saber coisa alguma sobre sua estrutura interna ou sobre sua localização, pois a interface contém apenas a *assinatura* dos métodos.

Nesta aplicação, desenvolvemos oito diferentes interfaces do usuário (4 clientes e 4 aplicações intermediárias, operadas remotamente pelos clientes) que criam, removem, atualizam, pesquisam e recuperam registros de um banco de dados reutilizando a interface `BancoDados`. As classes que compõem as interfaces do usuário estão no pacote `bancodados.user`. Todas usam referências do tipo `BancoDados` através das quais realizam as operações solicitadas.

As classes que terminam em `UI` são executáveis (possuem método `main()` e podem ser executadas pelo sistema de tempo de execução Java). Classes que possuem o nome **`Client`** são usadas como componentes do cliente. Classes que possuem o nome **`Server`** são usadas como componentes das aplicações que implementam os servidores intermediários.

As aplicações que interagem com o usuário utilizam as classes:

- **`DadosClientTextUI`**: Interface do usuário orientada a caracter.
- **`DadosClientPanel`**: Interface gráfica do usuário. Esta não é uma classe executável. É um `java.awt.Panel` que é usado como parte de um objeto do tipo `DadosClientUI` ou `DadosApplet` oferecendo uma interface uniforme nos dois tipos de aplicação.
- **`DadosClientUI`**: `java.awt.Frame` que fornece a estrutura para que a interface gráfica do usuário (`DadosClientPanel`) possa ser usada como uma aplicação do *Windows*.
- **`DadosApplet`**: `Applet` que fornece uma estrutura para que a interface gráfica do usuário (`DadosClientPanel`) possa ser executada dentro de um browser.
- **`DadosServlet`**: `Servlet` que permite que o servidor Web atue como cliente para a aplicação e seja controlado por uma página HTML em um browser.

São quatro as aplicações usadas para implementar servidores intermediários. Elas agem como servidores e clientes ao mesmo tempo. Como servidores, recebem as requisições dos clientes. Como clientes, repassam as requisições à camada inferior, que pode ser outra aplicação intermediária ou um driver para o meio de armazenamento. A aparência gráfica de todas as aplicações é a mesma pois todas estendem uma classe que fornece essa estrutura:

- **`DadosServerFrame`**: Classe abstrata derivada de `Frame` que fornece uma interface gráfica de apresentação e métodos padrão para todos os servidores intermediários.

- `RMIDadosServerUI`, `RMIIIOPDadosServerUI`, `CorbaDadosServerUI`, `TCPDados-ServerUI` Servidores intermediários que manipulam o banco de dados a partir de instruções remotas enviadas por clientes RMI/JRMP, RMI/IIOP (ou CORBA), CORBA e *BDProtocol* (protocolo proprietário), respectivamente.

Dependendo do tipo de servidor escolhido (nas aplicações cliente) as classes utilizadas poderão ser diferentes, como mostram as figura 4-7 (capítulo 4) e C-4, mas sempre preservam a estrutura de camadas mostrada na figura C-2.

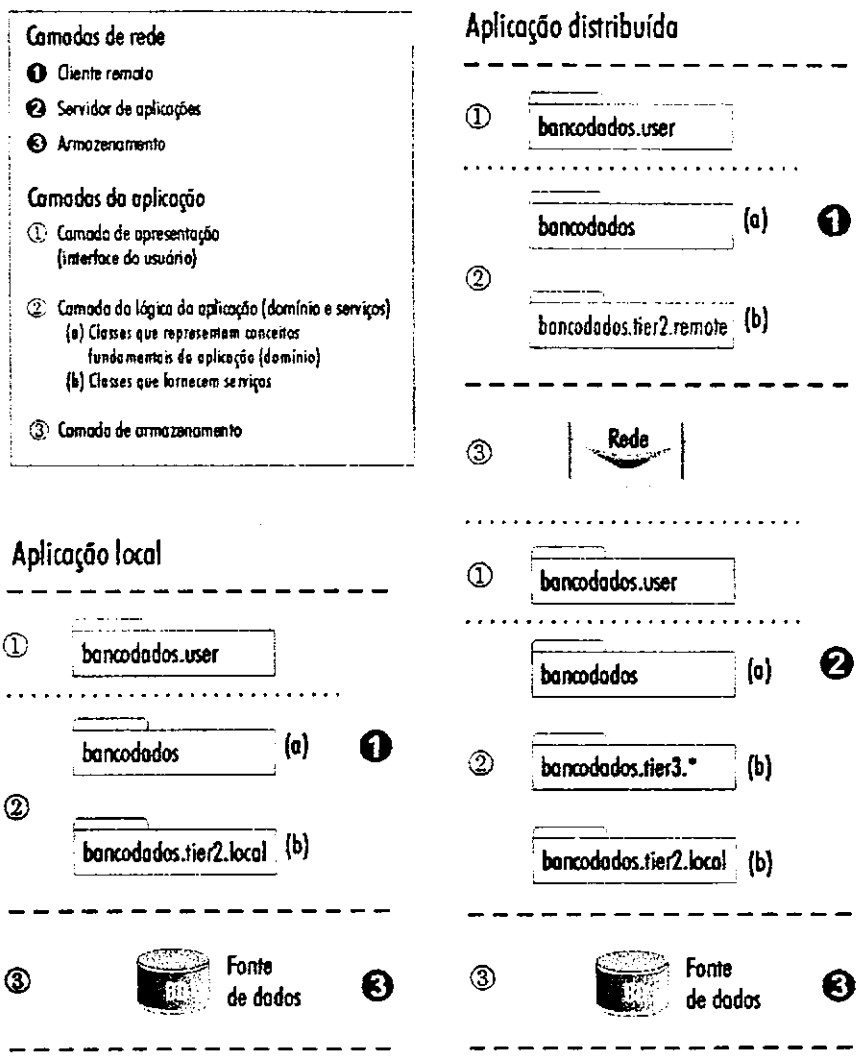


Figura C-2 – Arquitetura em camadas e pacotes Java/UML das aplicações de banco de dados

As classes restantes do pacote `bancodados.user` são janelas de dialogo e adaptadoras de eventos usadas pelas aplicações gráficas. A figura C-3 mostra todas as classes e todos os pacotes da aplicação.

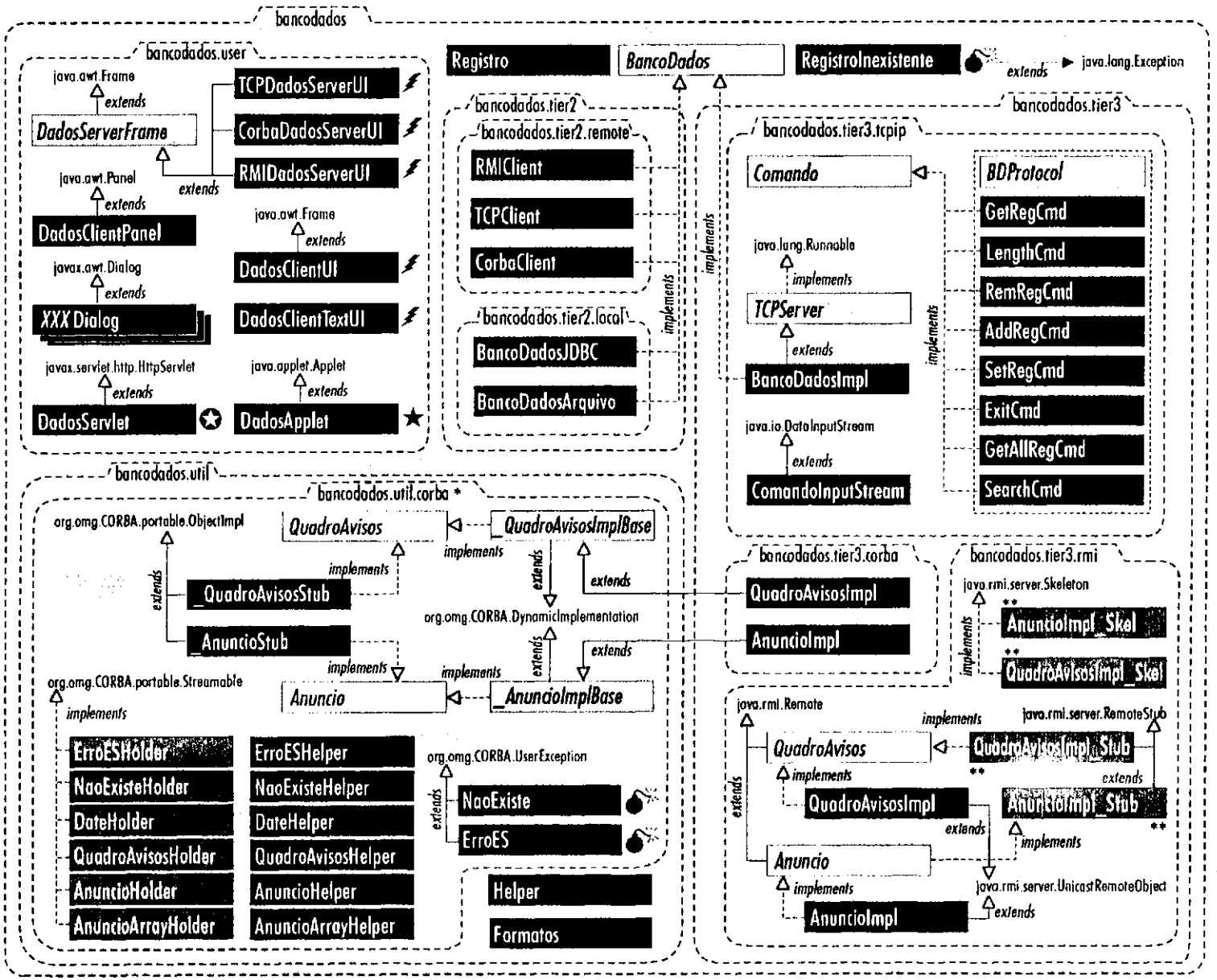


Diagrama de Pacotes e Classes Públicas

Todas as classes que não têm indicação de sua superclasse estendem *java.lang.Object*

Implementações de interfaces da API Java não foram indicadas exceto *java.lang.Runnable* (por caracterizar a classe implementadora como um *thread*)

Legenda

subclasse —extends▷ superclasse

classe ---implements▷ interface

java.io.DataInputStream
Classe da API Java (1.2)

pacote
pacote.subpacote

Interface

Classe Abstrata

Classe Concreta

Classe Final

Componente executável

Applet

Servlet

Exceção

* Este pacote e todas as suas 20 classes foram geradas pelo aplicativo *idltojava* (JDK1.2) a partir do IDL *bdcorba.idl*

** Estas 4 classes foram geradas pelo aplicativo *rmic* (JDK1.2)

Figura C-3 - Diagrama de classes públicas da aplicação 4 "bancodados". Classes de uso local, classes internas e da API Java foram omitidas. Somente as relações de herança e polimorfismo estão mostradas

A figura C-4 mostra um cenário, de uma aplicação rodando como applet e usando CORBA para intermediar o acesso ao banco de dados.

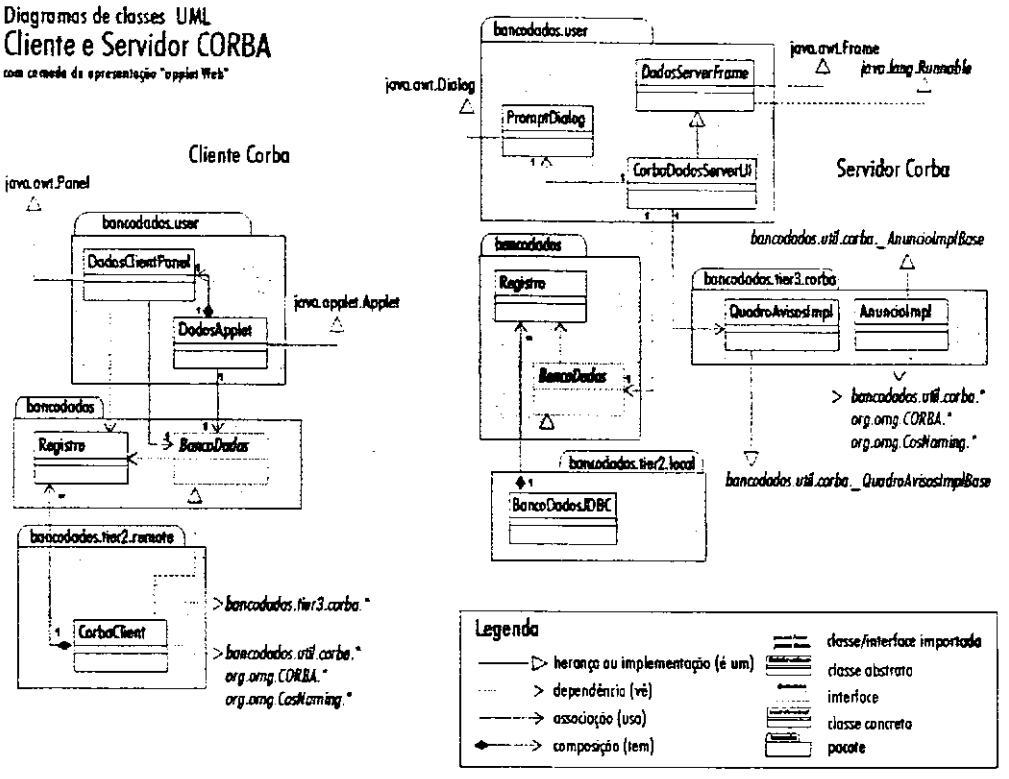


Figura C-4 - Diagramas de classes. Cliente e Servidor CORBA com camada de apresentação "applet Web".

C.2 Estrutura do código: *camada de armazenamento*

Esta seção apresenta alguns detalhes sobre as classes que servem de "drivers" aos bancos de dados implementados em arquivo e em sistemas de bancos de dados relacionais.

C.2.1 Aplicação de banco de dados em arquivo

Para implementar o banco de dados precisamos implementar a interface `bancodados.BancoDados` (figura C-1). Cada método deve realizar suas operações sobre um objeto do tipo `java.io.RandomAccessFile` que armazenará os objetos do tipo `Registro` em disco. O registro tem o seguinte formato:

- `int`: número do anuncio
- `String`: texto do anuncio
- `long`: data (tempo em milissegundos desde 1/1/1970)
- `String`: autor do anuncio

Podemos usar os métodos da classe `RandomAccessFile`: `writeInt()`, `writeLong()` e `writeUTF()` para gravar os tipos `int`, `long` e `String`, respectivamente e `readInt()`, `readLong()` e `readUTF()` para recuperá-los posteriormente.

O banco de dados tem a seguinte organização no arquivo:

- Os registros serão acrescentados ao arquivo em seqüência.
- Cada novo registro será acrescentado no final do arquivo com um número igual ao maior número pertencente a um registro existente mais um, ou `100`, se não houver registros;
- Registros removidos terão o seu número alterado para `-1` (continuarão ocupando espaço no arquivo).
- Registros alterados serão primeiro removidos e depois acrescentados no final do arquivo com o mesmo número que tinham antes (também continuarão ocupando espaço no arquivo).

A classe que desenvolvemos está em `/jad/apps/bancodados/tier2/local/` e chama-se `BancoDadosArquivo.java`. Implementa `BancoDados` podendo ser utilizada por qualquer outra classe que manipule com essa interface. A classe possui um objeto `RandomAccessFile` que representa o arquivo onde os dados serão armazenados. Suas variáveis membro e a implementação de seu construtor estão mostrados abaixo:

```
public class BancoDadosArquivo implements BancoDados {

   private RandomAccessFile arquivo;    // descritor de arquivo
   private boolean arquivoAberto;       // inicialmente false
   private Hashtable bancoDados;        // relaciona posicao do ponteiro
                                        // do RandomAccessFile com registro
   private int maiorNumReg = 0;         // Maior número de registro

   public BancoDadosArquivo(String arquivoDados) throws IOException {
      try {
         arquivo = new RandomAccessFile(arquivoDados, "rw");
         arquivoAberto = true;
      } catch (IOException e) {
         close();
         throw e;    // propaga excecao para metodo invocador
      }
   }
(...)
```

A referência **`arquivo`** é utilizada em todos os métodos que manipulam com os dados no arquivo. Abaixo listamos o método `addRegistro()`, que adiciona um novo registro.

```
public synchronized void addRegistro(String anuncio, String contato) {
      try {
         arquivo.seek(arquivo.length());  // posiciona ponteiro no fim
         arquivo.writeInt(getProximoNumeroLivre());
         arquivo.writeUTF(anuncio);
         arquivo.writeLong(new Date().getTime());
         arquivo.writeUTF(contato);
      } catch (IOException ioe) {
         ioe.printStackTrace();
      }
   }
```

Para remover um registro, é preciso saber em que posição ele está. O `Hashtable bancoDados` (definido em `getRegistros()`) contém um mapa que relaciona o número do registro com a posição no arquivo. O método `removeRegistro()` utiliza então esta informação para localizar o registro que ele deve marcar como removido.

```
public synchronized boolean removeRegistro(int numero)
                                        throws RegistroInexistente {
      try {
        getRegistros();
        String pointer = (String)bancoDados.get(new Integer(numero));
        if (pointer == null)
           throw new RegistroInexistente("Registro não encontrado");
        long posicao = Long.parseLong(pointer);
        arquivo.seek(posicao);
        int numReg = arquivo.readInt();
        if (numReg != numero)
           throw new RegistroInexistente("Registro não localizado");
        arquivo.seek(posicao);
        arquivo.writeInt(-1);   // marca o registro como removido
        arquivo.seek(0);
      } catch (IOException ioe) { // (...)
      }
      return true;
   }
```

Nesta interface que desenvolvemos para o arquivo usando `RandomAccessFile`, os registros removidos nunca são realmente removidos. Para limpar o arquivo, livrando-o de espaço ocupado inutilmente, é preciso exportar todos os registros válidos e importá-los de volta em um novo arquivo.

Consulte o código fonte para detalhes sobre os outros métodos.

C.2.2 Construção de uma aplicação JDBC

Nesta seção, construímos uma aplicação JDBC usando os mesmos dados do capítulo anterior, desta vez organizado em um BD relacional. Para reutilizar toda a interface do usuário e as classes que representam os conceitos fundamentais do programa, criamos uma classe que

implementa a interface `bancodados.BancoDados`. Como a interface do usuário usa a interface `BancoDados`, podemos usar a classe `bancodados.tier2.local.BancoDadosJDBC`, preservando a mesma interface do usuário usada para a versão baseada em arquivo.

Os dados utilizados por esta aplicação são do mesmo tipo que aqueles manipulados pela aplicação da seção anterior. Teremos, portanto, apenas uma tabela no banco de dados com a seguinte estrutura:

Tabela C-1 – Estrutura do banco de dados

Coluna	Tipo de dados das linhas	Informações armazenadas	Observações
`codigo`	`int`	número do anúncio	`integer` chave primária
`data`	`String`	data de postagem do anúncio	`char(24)`
`texto`	`String`	texto do anúncio	`char(8192)`
`autor`	`String`	autor do anúncio	`char(50)`

Utilizamos os tipos de dados mais fundamentais para garantir a compatibilidade com uma quantidade maior de bancos de dados.

Na classe `BancoDadosJDBC` carregamos um driver de acordo com a URL passada pelo usuário, que permitirá, no final, obter um objeto `java.sql.Statement` (JDBC) através do qual poderemos enviar requisições SQL ao servidor. O método `executeUpdate`, da interface `Statement`, pode ser usado para inserir registros na implementação de `addRegistro()`:

```
public synchronized void addRegistro(String texto, String autor) {
      (...)
      String insert = "INSERT INTO anuncios VALUES (" + numero + ", '"
                                                     + quando + "', '"
                                                     + texto +  "', '"
                                                     + autor +  "')";
      try {
         stmt.executeUpdate(insert); // objeto tipo Statement
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
```

O método `getRegistros()` obtém uma tabela de dados como resposta a uma requisição `SELECT` (SQL) enviada pelo método `executeQuery()`. Esse método retorna um `ResultSet` (que contém uma tabela virtual navegável *linha-a-linha* via seu método `next()`). Para cada posição, usamos os métodos `get`*Tipo*`()` apropriados para ler inteiros e strings.

```
public Registro[] getRegistros() {
     ResultSet rs;
     Vector regsVec = new Vector();
     String query = "SELECT numero, data, texto, autor " +
                    "FROM anuncios ";
     try {
        rs = stmt.executeQuery(query);
        while (rs.next()) {
           int numero = rs.getInt("numero");
           String texto = rs.getString("texto");
           String dataStr = rs.getString("data");
           java.util.Date data = df.parse(dataStr);
           // java.util.Date data = rs.getDate("data");
           String autor = rs.getString("autor");
           regsVec.addElement(new Registro(numero, texto, data, autor));
        }
     } catch (SQLException e) { // (...)
     } catch (java.text.ParseException e) { // (...)
     }
     Registro[] regs = new Registro[regsVec.size()];
     regsVec.copyInto(regs);
     return regs;
  }
```

A remoção do registro é implementada de forma mais simples ainda, bastando encapsular uma instrução SQL DELETE:

```
public synchronized boolean removeRegistro(int numero)
      throws RegistroInexistente {

      ResultSet rs;
      String delete = "DELETE FROM anuncios WHERE numero = " + numero;

      try {
         stmt.executeUpdate(delete);
         return true;
      } catch (SQLException e) {
         e.printStackTrace();
         return false;
      }
  }
```

C.3 Estrutura do código: *interfaces do usuário*

Nesta seção apresentamos detalhes referentes ao funcionamento da interface do usuário orientada a caractere (que contém as mesmas operações básicas presentes nas interfaces do usuário baseadas em applet, servlet e aplicação *Windows*) e das interfaces HTML, geradas pelo servlet.

C.3.1 Interface orientada a caractere

Os arquivos utilizados nesta aplicação estão nos subdiretórios a seguir. Em negrito está o único arquivo que trata desta interface:

Tabela C-2 – Componentes da aplicação de banco de dados, com acesso local apenas.

Subdiretório	Arquivo-fonte Java	Conteúdo
`bancodados/user`	**`DadosClientTextUI.java`**	interface do usuário orientada a caracter
`bancodados/`	`BancoDados.java`	interface genérica para o banco de dados (interface)
`bancodados/`	`Registro.java`	representação de um registro (classe concreta)
`bancodados/tier2/local`	`BancoDadosArquivo.java`	implementação de `BancoDados`
`bancodados/tier2/local`	`BancoDadosJDBC.java`	implementação de `BancoDados`

A interface do usuário deve manipular com um objeto `BancoDados`. Na prática, estará manipulando com o `RandomAccessFile` através da classe `BancoDadosArquivo` ou com os métodos JDBC através da classe `BancoDadosJDBC`, mas ela não precisa saber disso. Se estiver usando uma aplicação intermediária, poderá estar usando um cliente CORBA ou RMI que implementa `BancoDados`. Resumindo, a interface do usuário é uma camada que independe da forma de organização ou da localização dos dados que manipula.

A classe `DadosClientTextUI` declara uma variável membro do tipo `BancoDados`:

```
private BancoDados client;
```

e em todos os seus métodos chama métodos de `BancoDados` através de `client`. Apenas o menu principal refere-se ao `BancoDadosArquivo` ou `BancoDadosJDBC`. Quando o usuário escolhe um dos dois, ele é instanciado e sua referência é passada para `client`. A partir daí, todos os métodos operam sobre a interface `BancoDados`.

Se o usuário decidir criar um novo registro, por exemplo, a aplicação chamará o método local `criar()`, que contém:

```
public void criar() throws IOException {
      (...)
      client.addRegistro(texto, autor); // método de BancoDados
   }
```

Para listar todos os registros, o método `mostrarTodos()` é chamado:

```
public void mostrarTodos() throws IOException, RegistroInexistente {

        (...)
        Registro[] regs = client.getRegistros();
        (...)
}
```

Em *nenhum* dos métodos há indicações que acontece alguma coisa em um `RandomAccessFile` (banco de dados baseado em arquivo) ou na interface `Statement` (banco de dados relacional), portanto, a camada de apresentação está isolada da segunda camada.

C.3.2 Interface HTML com servlets HTTP

Com servlets, a aplicação de banco de dados pode ser acessível através de uma interface HTML. Construímos uma interface baseada em duas páginas. A primeira contém a interface onde o usuário pode escolher endereço e serviço que irá fornecer os dados. Passando da primeira página (um serviço foi selecionado e este aceitou a conexão), uma segunda página será mostrada com todos os registros disponíveis no banco de dados[1]. A primeira página não é alterada pelo servlet. É simplesmente lida do disco e repassada ao browser.

Um esqueleto da segunda página deve ser lido pelo servlet que irá preencher uma tabela com todos os registros encontrados antes de enviá-la para o browser. Este preenchimento também inclui uma lista de vínculos (links) de acesso rápido, antes e depois da tabela, e vínculos rápidos para o menu em qualquer registro. No início da segunda página há um painel de controle que permite gerenciar o banco de dados. Esses botões respondem a eventos JavaScript e alteram as informações que serão enviadas para o servlet. No final da página há uma área para edição e criação de novos registros, construídos usando formulários HTML. As páginas estão localizadas no diretório `/jad/apps/htdocs/`. A página foi criada para ser carregada via servlet (a carga direta do arquivo via browser provocará erros de JavaScript).

O código da página dados.html antes de ser lida pelo servlet contém marcadores de posição indicados por comentários HTML (`<!-- -->`):

```
(...)
<!-- links -->
</p><table border=0 width=590 bgcolor="#dddddd">
<tr><th>Número</th><th>Texto</th><th>Autor</th><th>Data</th></tr>
<!-- dados -->
(...)
```

[1] Não foram tomadas providências para quebrar a página em páginas menores a medida em que o número de registro crescer, portanto, esta versão pode ser impraticável para acessar grandes quantidades de informação.

```
<tr><td width=70> </td><td> </td><td> </td><td> 
</td></tr></table><p>
<!-- links -->
</p></div>
(...)
```

Após a leitura da página pelo servlet, o browser recebe uma página gerada dinamicamente, com os marcadores `<!--links -->` e `<!--dados -->` (em negrito no código acima e abaixo) substituídos por informações geradas a partir dos dados armazenados (veja figura 4-4 no capítulo 4):

```
<a href="#100">100</a> | <a href="#101">101</a>
  (... várias linhas removidas ...)
| <a href="#134">134</a> |
</p><table border=0 width=590 bgcolor="#dddddd">
<tr><th>Número</th><th>Texto</th><th>Autor</th><th>Data</th></tr>
<tr valign=top><td><a name="100">100</br><a
href="#top"><small>[MENU]</small></a></td><td>Cachimbos, ... tam-
bém!</td><td>R. Magritte (magritte@recursive.org)</td><td>21/12/97
18:39</td></tr>
 (... várias linhas removidas ...)
<tr valign=top><td><a name="131">131</br><a
href="#top"><small>[MENU]</small></a></td><td>Vende-se um ... da
Terra.</td><td>G. Galileo (galileo@galilei.lua)</td><td>31/12/98
17:06</td></tr>
<input type=hidden name=tarea value="">
<input type=hidden name=tfld value="">
<input type=hidden name=numero value="0">
(...)
<tr><td width=70> </td><td> </td><td> </td><td> 
</td></tr></table><p>
<a href="#100">100</a> | <a href="#101">101</a>
  (... várias linhas removidas ...)
| <a href="#134">134</a> |
</p></div>
```

Para implementar a interface do usuário, podemos usar somente HTML. A vantagem é que nossa página será acessível até pelo mais primitivo dos browsers. O problema é que HTML é muito limitado quanto aos recursos de interação com o usuário. HTML oferece três tipos de eventos:

- "clicar sobre um link" que inicia uma requisição `GET` ao servidor.
- "apertar um botão submit" que envia os dados de um formulário ao servidor através de uma requisição `POST` ou `GET`
- "apertar um botão reset", que reinicializa os campos de um formulário aos seus valores *default*.

Os controles da aplicação de banco de dados são mais complexos. É preciso que os botões façam mais que simplesmente enviar dados ao servidor. Pode ser que o usuário da aplicação Web selecione um registro e queira apagá-lo ao apertar o botão. Pode ser que ele selecione um registro e aperte o botão para alterá-lo. Pode ser que ele queira realizar uma busca.

Para superar essa e outras limitações do HTML, usamos JavaScript. JavaScript possui cerca de 13 eventos e botões podem ser reprogramados para realizarem algo diferente de limpar campos ou enviar dados. Para programá-los, usamos botões neutros (que não provocam eventos). Estes botões são representados em HTML por:

```
<input type="button">
```

O evento de clique do botão é representado pelo atributo `onclick` que pode ser usado em qualquer botão e contém instruções JavaScript que devem ser executadas assim que o botão for apertado.

```
<input type=button onclick="alert('O botão foi apertado!')">
```

Na nossa interface, fizemos com que alguns botões chamassem funções, definidas no bloco `<script> ... </script>` no início do arquivo, e outros mudassem *parâmetros ocultos*, representados em HTML como

```
<input type=hidden name="variavel" value="valor da variavel">
```

Os dados dos campos ocultos são passados na requisição do browser da mesma maneira em que são passados os dados de campos de texto e outros dispositivos de entrada. O poder do JavaScript está na possibilidade de mudar o valor dos campos ocultos enquanto uma página está sendo exibida, em resposta a um evento (um aperto de botão, por exemplo).

Usamos campos ocultos, por exemplo, para que o servlet saiba qual o 'comando' que foi solicitado pelo usuário da aplicação Web, alterando o campo

```
<input type="hidden" name="comando" value="getReg">
```

Quando o usuário aperta o botão "Remover" para remover um registro, o conteúdo do atributo `onclick` desse botão é executado:

```
<INPUT TYPE="button" NAME="tira" value="Remover..."
                     onclick="remover(this.form)">
```

`remover()` é o nome de uma função JavaScript definida no início da página. `this.form` é uma referência que passa o próprio formulário como argumento da função. A função `remover()` está definida como:

```
<script>
function remover(entra) {
  candidato = prompt("Digite número do registro a ser removido","");
  if (candidato) {
    entra.numero.value = candidato;  // muda valor de campo oculto 'numero'
    if (confirm("Tem certeza que quer remover o registro " + candidato + "?")) {
      entra.comando.value = "remReg"; // muda valor do campo 'comando'
      entra.submit();
    }
  }
}
</script>
```

Dentro da função, o formulário é representado pela variável **entra**. `entra.comando` é uma referência ao campo oculto de nome `comando`. `entra.comando.value` é o valor deste campo que na linha marcada em negrito acima é alterado de '`getReg`' para '`remReg`'. A linha seguinte

```
entra.submit();
```

envia o formulário. O servlet, após decodificar a linha de dados recebida, buscará pelo nome '`comando`' e receberá o valor '`remReg`', indicando que o usuário deseja remover um registro. O número do registro a ser removido foi armazenado em outro campo oculto que o servlet pode ler. Desta forma, é possível implementar todas as outras funções da interface do usuário.

Para maiores detalhes sobre esta aplicação, consulte o código fonte localizado no arquivo `DadosServlet.java` (diretório `jad/apps/bancodados/user`).

Os exemplos deste capítulo foram testados usando o *Servletrunner*. Para executar os servlets, estes precisam ser instalados no *Servletrunner*. A instalação é realizada através de um arquivo de configuração chamado `servlet.properties` que vincula o nome do servlet a um arquivo `.class` e passa quaisquer parâmetros adicionais de inicialização necessários. Para esta aplicação, o arquivo `servlet.properties` deverá conter os seguintes dados:

```
servlet.bdservlet.code=bancodados.user.DadosServlet
servlet.bdservlet.initArgs=\
        htmlDados=j:/jad/apps/htdocs/dados.html,\
        htmlSelecao=j:/jad/apps/htdocs/selecao.html,\
        appHome=j:/jad/apps
```

O *Servletrunner* pode ser iniciado para esta aplicação rodando o arquivo `runServlet.sh` (ou `runServlet.bat`) no subdiretório `/jad/apps/`. Depois de iniciado, o *Servletrunner* estará no ar na porta 8080 (*default*) e irá servir servlets armazenados em (ou localizáveis a partir de) `jad/apps/`. A URL para chamar o servlet `bdservlet` é `http://servidor:8080/servlet/bdservlet`. O nome do servidor e a sua porta devem ser os nomes e porta de onde o *Servletrunner* está rodando.

C.4 Estrutura do código: *aplicações intermediárias*

Não há grandes diferenças entre a parte cliente dessas aplicações e as camadas de armazenamento, do ponto de vista da camada de apresentação. Todas implementam a interface `bancodados.BancoDados`. Seu código fonte se está em `/jad/apps/bancodados/tier2/remote/`.

Cada servidor, porém, tem uma estrutura própria de acordo com a tecnologia que utiliza. Precisam rodar como processos ativos para que possam ficar aguardando clientes. Todos os servidores possuem uma interface gráfica em `bancodados.user` que estende a classe abstrata `DadosServerFrame`. Ela possui a infraestrutura básica para qualquer servidor e permite que o cliente escolha um arquivo ou uma fonte de dados JDBC que o servidor irá servir.

O pacote `bancodados.tier3` contém um sub-pacote para cada implementação de servidor. Para maiores informações sobre essas aplicações, consulte o código-fonte em `/jad/apps/bancodados/tier3/`.

Referências bibliográficas e obras consultadas

[ARNO96] Ken ARNOLD e James GOSLING. *The Java Programming Language.* "The Java Series". JavaSoft / Addison-Wesley, 1996.

[BARR99] José BARROS. *Java Tip 65: Measure data transfer speeds via Sun's ORB in JDK.* JavaWorld Magazine. January 1998. URL: `http://www.javaworld.com/javaworld/javatips/jw-javatip65.html`.

[BINK98] C. William BINKO, *CORBA-IDL Mapping.* Java Report Vol. 3. No. 3 (March 1998).

[BOOC94] Grady BOOCH. *Object Oriented Analysis and Design.* 2nd. Ed. Benjamin/Cummings, 1994.

[CAFF97] CaffeineMark. *System Comparisons.* Pendragon Software Corp. URL: `http://www.webfayre.com/pendragon/jpr/jpr049702.html`.

[CAMP99] Mary CAMPIONE e Kathy WALRATH. *The Java Tutorial - on-line edition.* URL `http://java.sun.com/tutorial/`.

[CHAN97] Patrick CHAN. *The Java Developer's Almanac 1998.* The Java Series – Addison Wesley, 1998.

[CHAN98] Dan T. CHANG e Dan HARKEY. *Client/Server Data Access With Java and XML.* Wiley, 1998.

[COAD97] Peter COAD & Mark MAYFIELD. *Java Design – Building Better Apps & Applets.* Yourdon Press Computing Series. Prentice-Hall, 1997.

[CORN97] Gary CORNELL e Cay HORTSMANN. *Core Java Vols. 1 & 2.* SunSoft / Prentice-Hall 1997.

[COOP97] James W. COOPER. *Principles of Object Oriented Programming in Java 1.1.* Ventana Press, 1997.

[CURT97] David CURTIS. *Java, RMI and CORBA. White Paper.* Object Management Group – OMG. 1997. URL: `http://www.omg.org/news/wpjava.html`.

[CURT98] David CURTIS. IIOP: *OMG's Internet Inter-ORB Protocol - A Brief Description.* Object Management Group – OMG, 1998. URL: `http://www.omg.org/library/iiop4.html`.

[EDWA] Nigel EDWARDS, Owen REES. *Performance of HTTP and CGI.* ANSA. URL: `http://www.ansa.co.uk/ANSA/ISF/1506/APM1506.html`.

[FARL98] Jim FARLEY. *Java Distributed Computing.* O'Reilly and Associates. 1998.

[FLAN97] David FLANAGAN. *Java In a NutShell* (2nd. Ed.). O'Reilly and Associates, Inc., 1997.

[FOOD98] Mike FOODY. *Distributed Java Applications – Comparing RMI, CORBA and Network Ease for Applets.* Java Report Vol. 3. No. 3 (March 1998).

[GAMM94] Erich GAMMA et. al. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series, 1994.

[GROT98] R. GROTHMANN. *Java Benchmark.* 1998. URL: `http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann/java/bench/Bench.html`.

[GUND96] Shishir GUNDAVARAM. *CGI Programming on the World Wide Web.* O'Reilly and Associates. 1996.

[HAHM 96] Qusay H HAHMOUD., *Sockets Programming in Java: A tutorial.* JavaWorld Magazine, December 1996. URL: `http://www.javaworld.com/javaworld/jw-12-1996/jw-012-sockets.html`.

[HARO97] Elliote R. HAROLD. *Java Network Programming.* O'Reilly and Associates, Inc., 1997.

[HUGH97] Merlin HUGHES. *Drawing the world: Networked whiteboards.* JavaWorld Magazine, November 1997. URL: `http://www.javaworld.com/jw-11-1997/jw-11-step.html`.

[HUGH972] Merlin HUGHES. *Networking our whiteboard with 1.1.* JavaWorld Magazine, December 1997. URL: `http://www.javaworld.com/jw-12-1997/jw-12-step.html`.

[KERN78] Brian KERNIGHAN, Dennis RITCHIE. *The C Programming Language.* Prentice-Hall. 1978.

[KERN98] Brian KERNIGHAN, Christopher WYK. *Timing Trials, or, the Trials of Timing: Experiments with Scripting and User-Interface Languages.* Bell Labs 1998. URL: `http://cm.bell-labs.com/cm/cs/who/bwk/interps/pap.html`.

[LARM98] Craig LARMAN. *Applying UML and Patterns – An Introduction to Object Oriented Analysis and Design.* Prentice-Hall, 1998.

[LIU96] Cricket LIU et al. *Managing Internet Information Services.* O'Reilly and Associates. 1996.

[MAGE98] MAGELANG Institute Online Training. *Fundamentals of Java Servlets.* Magelang Institute. 1998.

[MICR97] MICROSOFT Corporation. *Internet Server API Overview.* Microsoft Corporation, 1997. URL: `http://www.eu.microsoft.com/win32dev/apiext/isapimrg.htm`

[MORG97] Bryan MORGAN. *Distributed objects meet the Web - Build scalable, n-tier applications on the Web using Java and CORBA.* JavaWorld Magazine, October 1997. URL: `http://www.javaworld.com/jw-10-1997/jw-10-corbajava.html`.

[MORG981] Bryan MORGAN. *Applied CORBA: Integrating legacy code with the Web.* JavaWorld Magazine, January 1998. URL: `http://www.javaworld.com/jw-01-1998/jw-01-corbalegacy.html`.

[MORG982] Bryan MORGAN. *Java 1.2 extends Java's distributed object capabilities.* JavaWorld Magazine, April 1998. URL: `http://www.javaworld.com/jw-04-1998//jw-04-distributed.html`.

[MOUR86] Antão MOURA, Jacques SAUVÉ, William GIOZZA, Fábio MARINHO. *Redes Locais de Computadores – Protocolos de Alto Nível e Análise de Desempenho.* McGraw-Hill/Embratel, 1986.

[MURP99] Gary L. MURPHY. *Are You Being Served?* Teledynamics. January 1999. URL: `http://www.geocities.com/SiliconValley/7704/servlets.html`.

[NETC99] NETCRAFT. *The Netcraft Web Server Survey.* URL: `http://www.netcraft.com/Survey/`.

[OAKS97] Scott OAKS & Henry WONG. *Java Threads.* O'Reilly and Associates, Inc., 1997.

[OMG98] OMG - Object Management Group. *CORBA 2.2 Specification.* OMG 1998. URL: `http://www.omg.org`.

[REES97] George REESE. *Database Programming with JDBC and Java.* O'Reilly and Associates, Inc., 1997.

[RFC2068] R. FIELDING et al. *Hypertext Transfer Protocol - HTTP/1.1.* Request for Comments: 2068. IETF, January 1997.

[RFC2109] D. KRISTOL and L. MONTULLI. *HTTP State Management Mechanism.* Request for Comments: 2109. IETF, February 1997.

[RFC2045] N. FREED and N. BORENSTEIN. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* Request for Comments: 2045. IETF, November 1996.

[ROCH99] Helder da ROCHA. *Desenvolvendo Web Sites Interativos com JavaScript.* Apostila ASIT. IBPINET/COPPE-UFRJ, 4a. Rev. Janeiro 1999.

[SHIE98] Ted SHIEH. *Programming Language Comparison.* 1998. URL: `http://odin.bio.sunysb.edu/tedshieh/software/`.

[SHOF97] Michael SHOFFNER. *Scale an application from two to three tiers with JDBC.* JavaWorld Magazine, June 1997. URL: `http://www.javaworld.com/jw-06-1997/jw-06-step.html`.

[SHOF98] Michael SHOFFNER. *Networking our whiteboard with servlets.* JavaWorld Magazine, January 1998. URL: `http://www.javaworld.com/jw-01-1998/jw-01-step.htm`.

[SHOF982] Michael SHOFFNER. *Add the power of CORBA to our distributed whiteboard* JavaWorld Magazine, March 1998. URL: `http://www.javaworld.com/jw-03-1998/jw-03-step.htm`.

[SLOT97] Louis P. SLOTHOUBER. *A Model of Web Server Performance.* 1997. URL: `http://louvx.biap.com/webperformance/modelpaper.html`.

[SPAI96] Stephen SPAINHOUR & Valerie QUERCIA. *Webmaster in a Nutshell – A Desktop quick reference.* O'Reilly and Associates, 1996.

[SPER95] Simon E. SPERO. *Analysis of HTTP Performance Problems.* 1995. URL: `http://metalab.unc.edu/mdma-release/http-prob.html`.

[STEV94] W. Richard STEVENS. *TCP/IP Illustrated Volume 1 – The Protocols.* Addison-Wesley Professional Computing Series, 1994.

[SUN96] James GOSLING, Henry McGILTON. *The Java Language Environment: A White Paper.* URL: `http://java.sun.com/docs/white/langenv/`.

[SUN97] SUN Microsystems. *Remote Method Invocation - RMI FAQ.* Sun/JavaSoft, 1997. URL: `http://java.sun.com/pr/1997/june/statement970626-01.faq.html`.

[SUN98] SUN Microsystems. *Java 2 Platform Documentation.* Sun/JavaSoft. 1998. URL: `http://java.sun.com/`.

[SUN982] SUN Microsystems. *Java Servlet Development Kit Documentation 2.0.* Sun/JavaSoft 1998. URL: `http://jserv.java.sun.com`.

[SUN99] SUN Microsystems. *RMI over IIOP beta 0.2 draft documentation.* Sun/JavaSoft 1998.

[TANE89] Andrew TANENBAUM. *Computer Networks.* Prentice-Hall, 1989.

[VALE97] Tom VALESKY. *Build Three-Tier Applications Using RMI and JDBC.* Internet Java & ActiveX Advisor, June 1997.

[VOGE98] Andreas VOGEL and Keith DUDDY. *Java Programming with CORBA – Advanced Tehniques for Building Distributed Applications.* Second Edition. OMG/Wiley 1998.

[WALL91] Larry WALL, Randall SCHWARTZ. *Programming Perl.* O'Reilly and Associates. 1991.