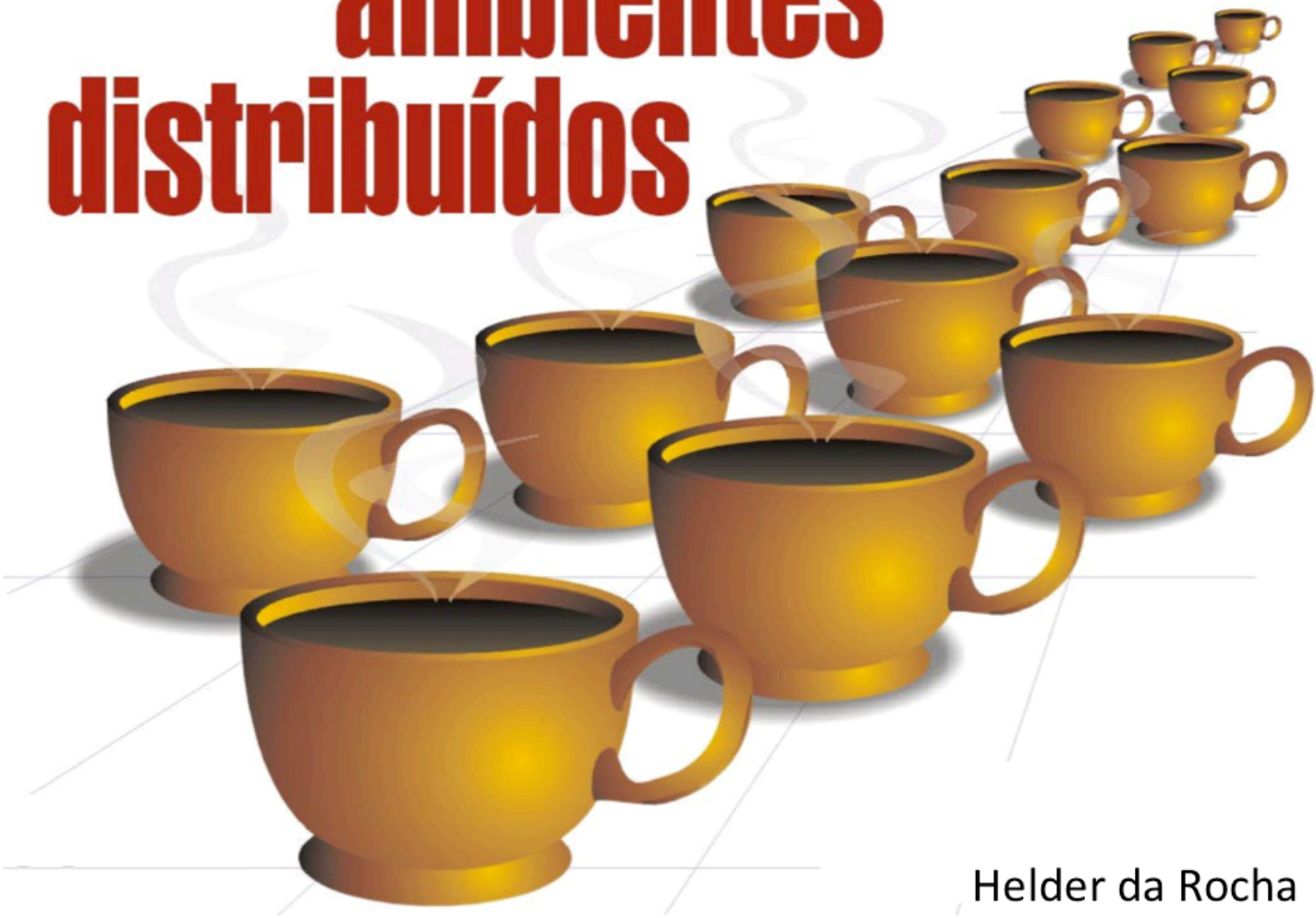
Usando Java™ em ambientes distribuídos



Helder da Rocha

JAVA 10

Uma aplicação distribuída

A FINALIDADE DESTE MÓDULO É APRESENTAR UM EXEMPLO DIDÁTICO que ilustre o uso prático das tecnologias discutidas nos módulos anteriores (e no módulo de servlets) em cenários onde aplicações distribuídas são necessárias. 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. Este módulo serve de referência e é essencial para a total compreensão dos exemplos dos módulos Java 6 e Java 7.

Tópicos abordados neste módulo

- Execução e instalação de uma aplicação Java multiplataforma cujo código foi analisado em módulos anteriores
- Discussão a respeito de desempenho e aplicabilidade de cada solução em diversos cenários.

Índice

Objetivos

No final deste módulo você deverá ser capaz de:

- comparar as várias soluções de conectividade usando Java e saber escolher a mais adequada a determinado ambiente.
- perceber as vantagens do uso de interfaces e outros mecanismos orientados a objetos que promovem o reuso de componentes.

10.1.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.

10.2. Apresentação e execução das aplicações

As figuras 10-1 a 10-5 ilustram as interfaces do usuário das aplicações analisadas. Todas possuem o mesmo núcleo. Foram construídas separando a lógica da aplicação da interface do usuário e de armazenamento.

Aplicações independentes (executam sob o sistema operacional)

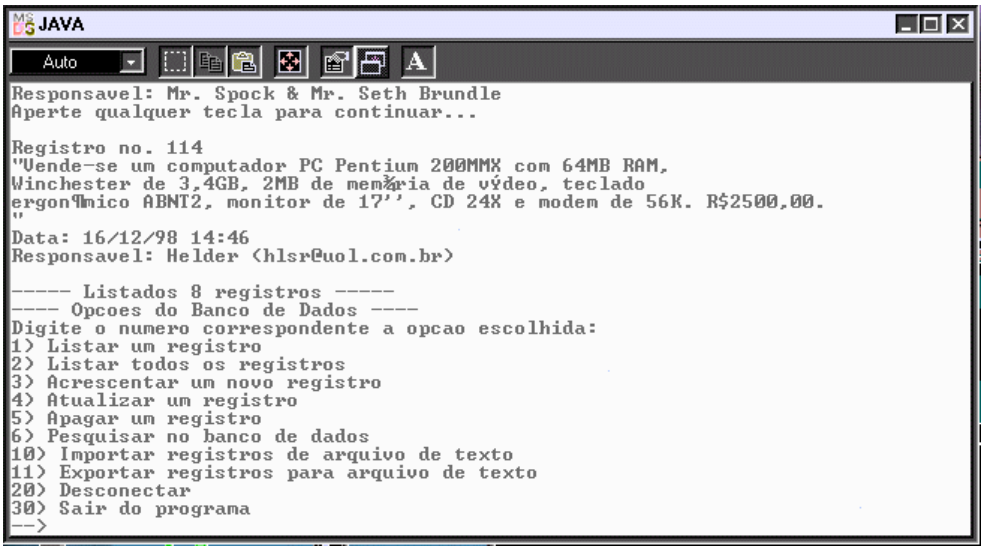


Figura 10-1 - Aplicação cliente com interface do usuário orientada a caracter

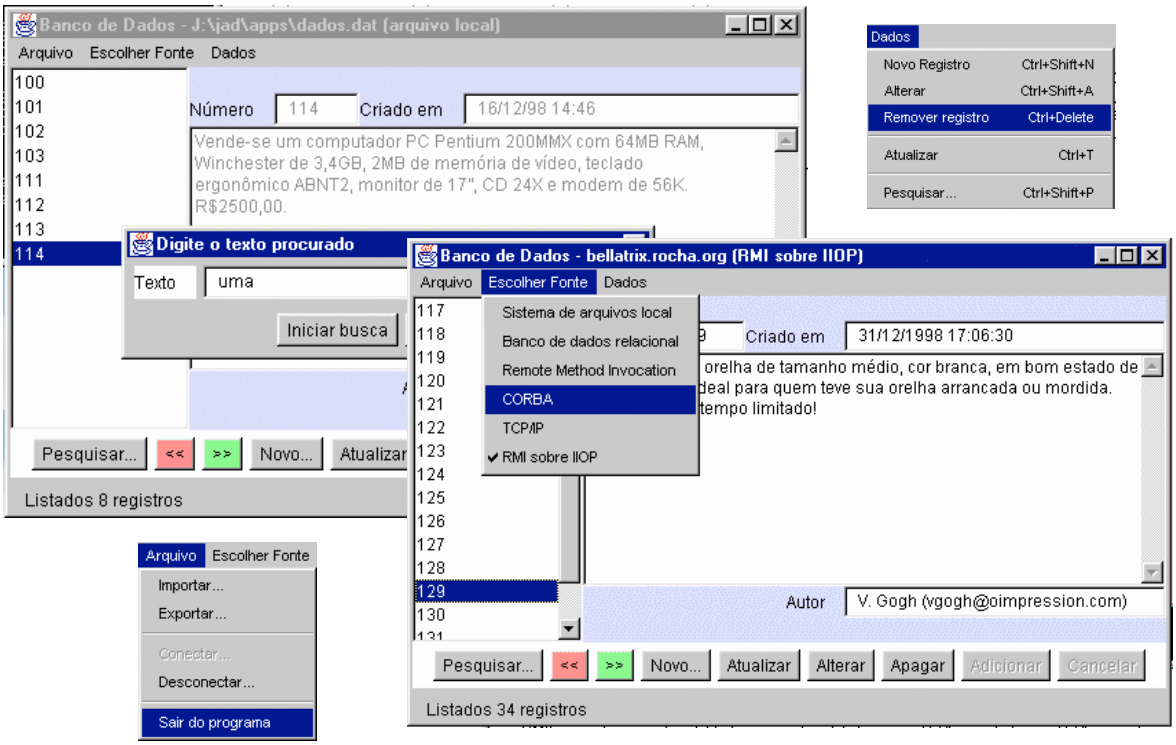


Figura 10-2 – Aplicação cliente com interface gráfica. (a) durante opções "Pesquisar..." em arquivo local; (b) e (c) menus; (d) listando os dados de banco de dados remoto (via RMI/IIOP)

Aplicações Web: Applets e Servlets

Figura 10-3 (a)– Interface cliente como applet em browser Netscape(com opção "pesquisar..." ativada)

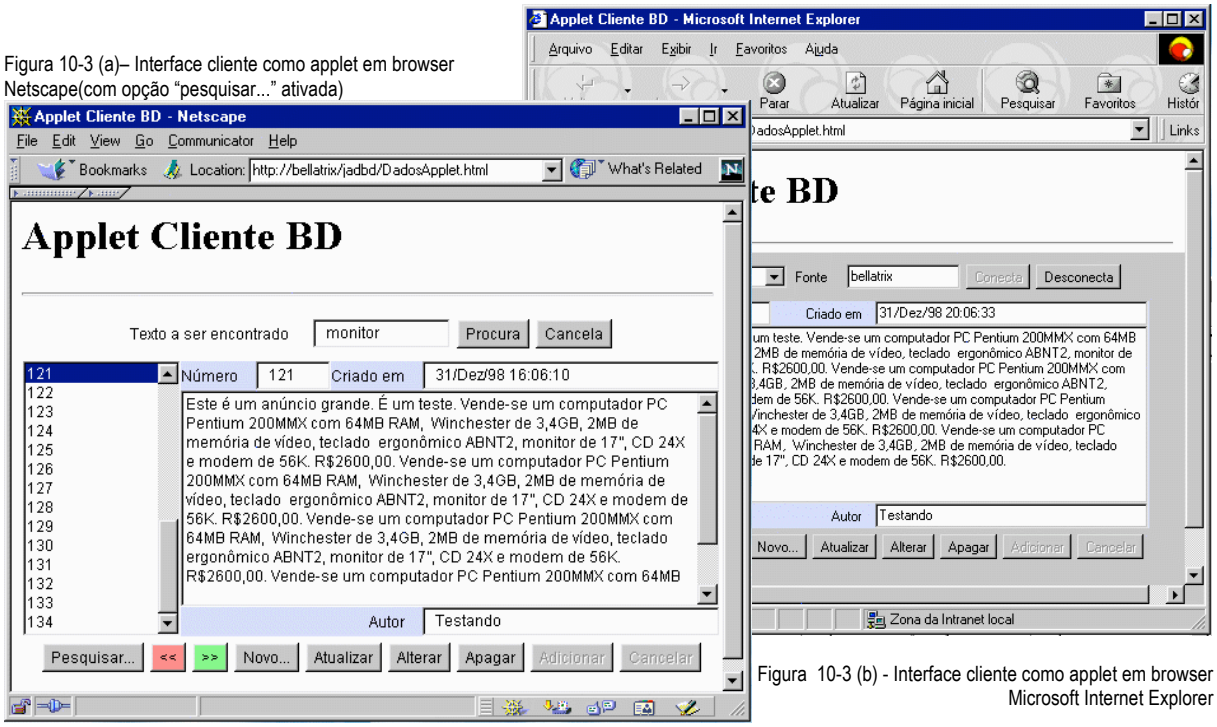


Figura 10-3 (b) - Interface cliente como applet em browser Microsoft Internet Explorer

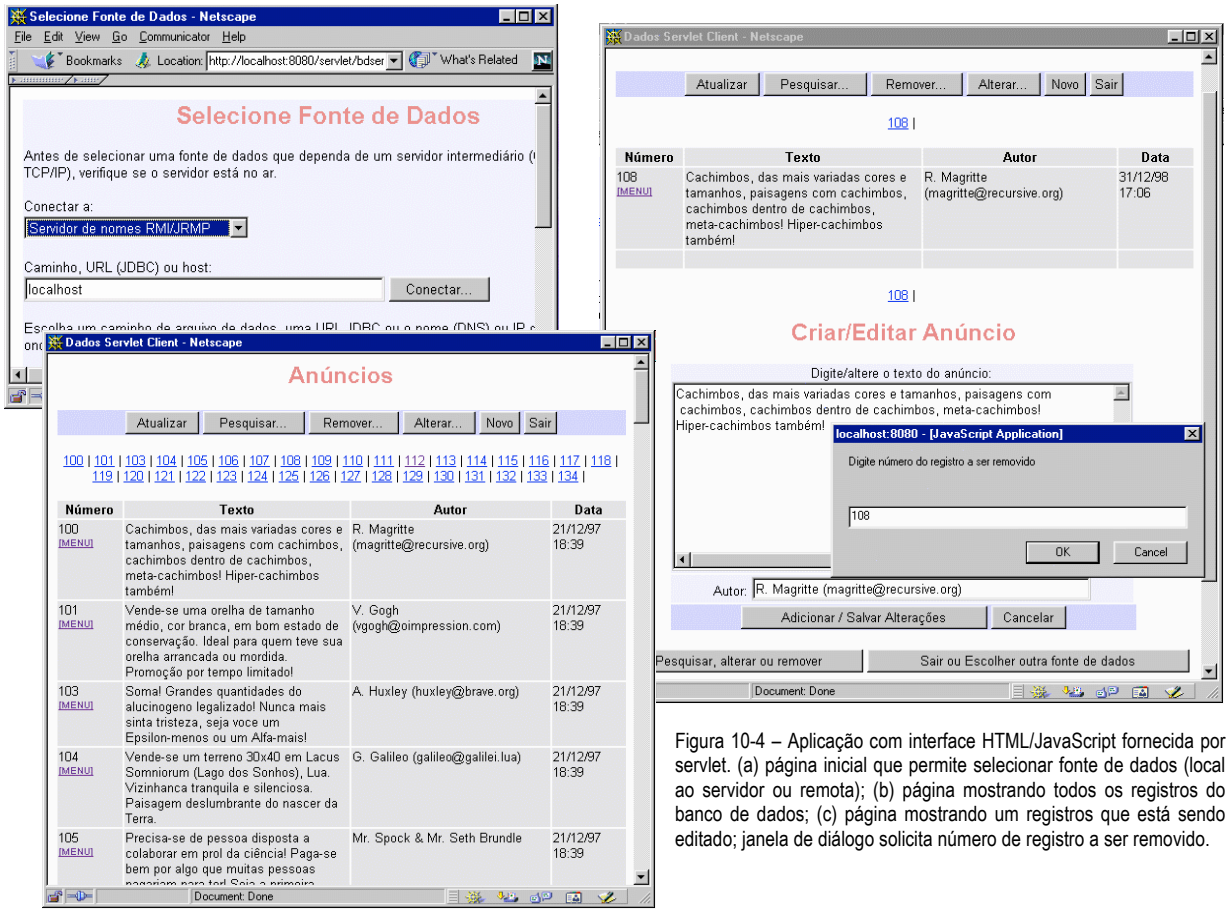


Figura 10-4 – Aplicação com interface HTML/JavaScript fornecida por servlet. (a) página inicial que permite selecionar fonte de dados (local ao servidor ou remota); (b) página mostrando todos os registros do banco de dados; (c) página mostrando um registros que está sendo editado; janela de diálogo solicita número de registro a ser removido.

Aplicações intermediárias (servidores)

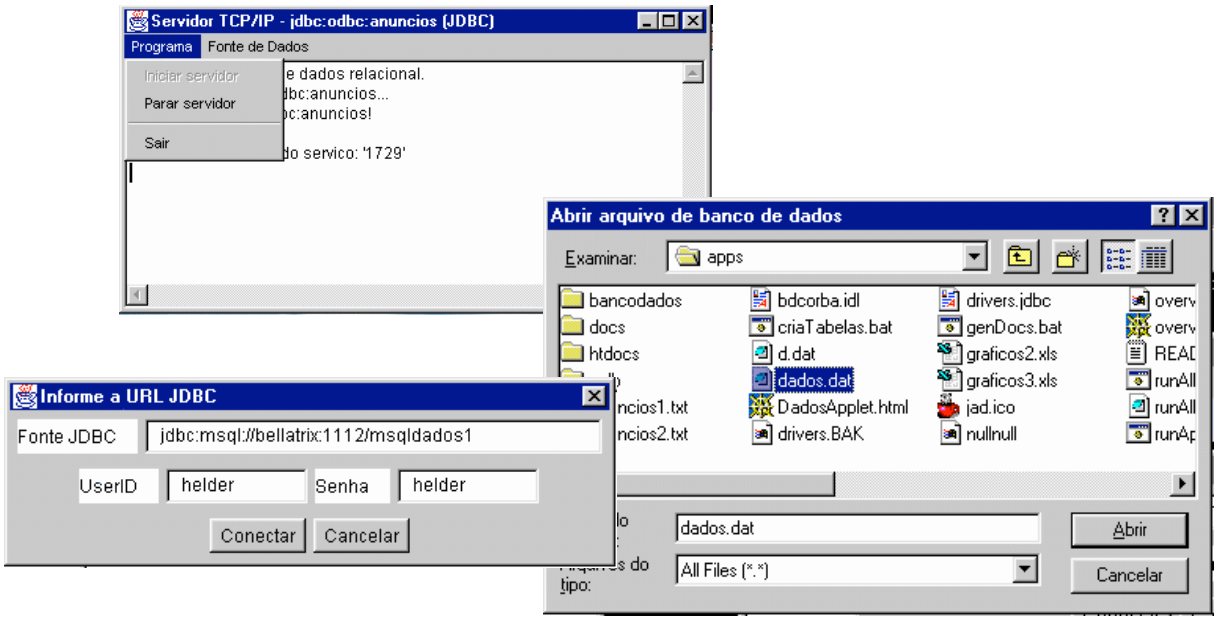


Figura 10-5 – (a) Aparência geral dos servidores intermediários (BDProtocol, CORBA, RMI, RMI/IIOP).
(b) Diálogo para conectar com banco de dados em arquivo. (c) Diálogo para conectar em banco de dados relacional através de URL JDBC

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 10-1), gráfica (10-2), applet (10-3) e servlet (10-4) 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 10-6.

Não é objetivo deste capítulo comparar o desempenho entre tecnologias de objetos remotos, portanto, as diferenças entre CORBA, RMI e TCP/IP não serão consideradas nas aplicações analisadas aqui. 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 10-7.

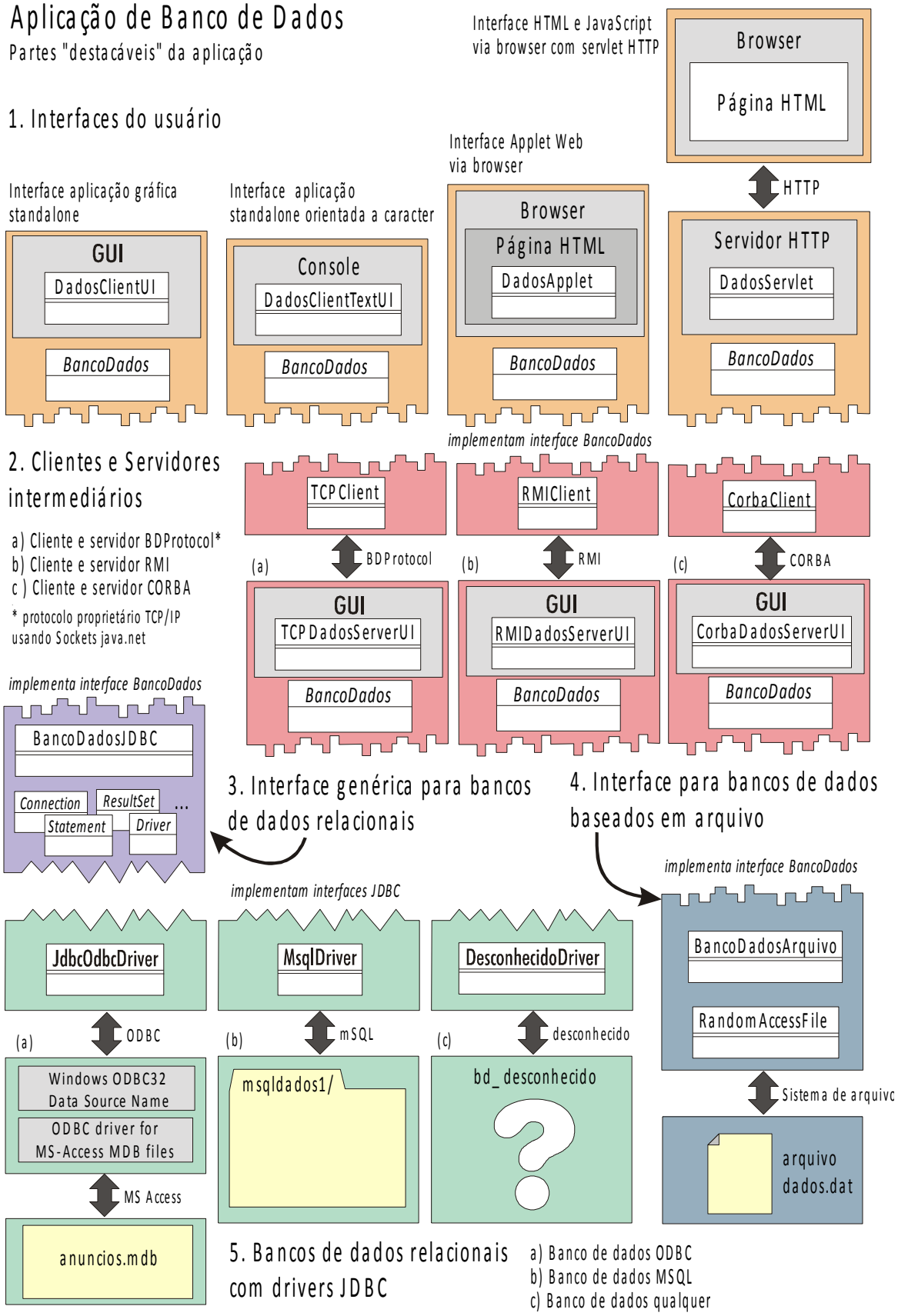


Figura 10-6 - Partes destacáveis da aplicação "bancodados". Veja mais detalhes sobre a estrutura da aplicação no apêndice C.

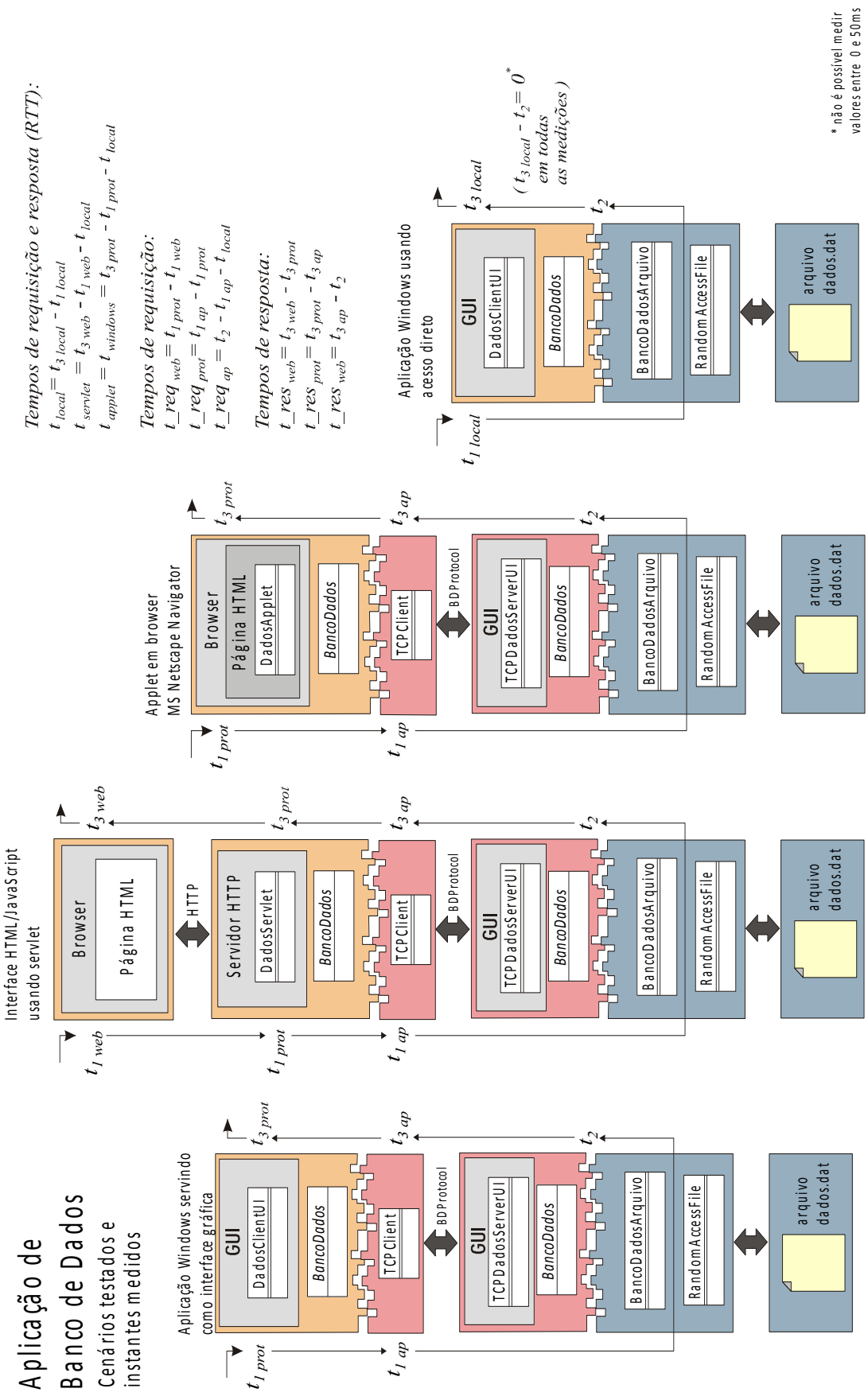


Figura 10-7 - Combinações dos blocos da aplicação "bancodados" utilizadas nos testes

10.3. 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 módulo estão no diretório `/jad/apps/` criado após a descompactação do disquete distribuído juntamente com este texto.

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`. O arquivo MS-DOS `genDocs.bat` gera a documentação caso esta não esteja disponível. Neste caso, é preciso criar um diretório `docs` abaixo de `jad/apps/`.

Maiores detalhes sobre a construção das aplicações estão nos módulos *Java 11*, *Java 6* e *Java 7*.

10.4.Construção da aplicação

Este apêndice fornece alguns detalhes sobre a estrutura da aplicação descrita na seção anterior. 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, HTML e JavaScript.

Estrutura da aplicação

A aplicação estudada nos módulos Java 6 e Java 7 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 10-8 ilustra os diagramas da classe `Registro` e interface `BancoDados` e seus métodos.

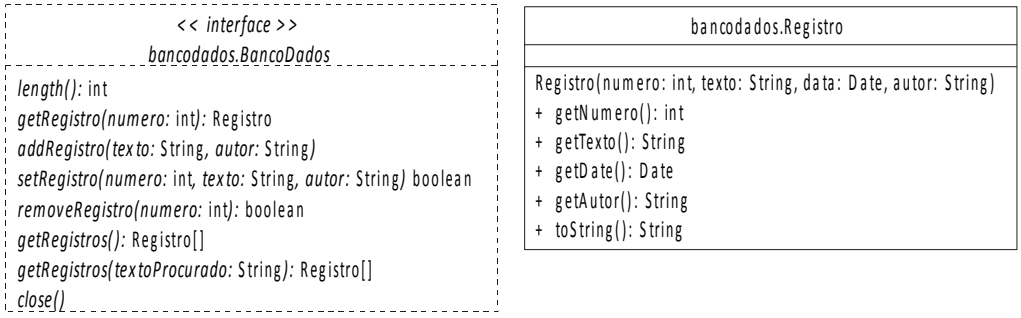


Figura 10-8 Diagramas de BancoDados e Registro

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`**, **`TCPDadosServerUI`** 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 figuras 10-7 e 10-11, mas sempre preservam a estrutura de camadas mostrada na figura 10-9.

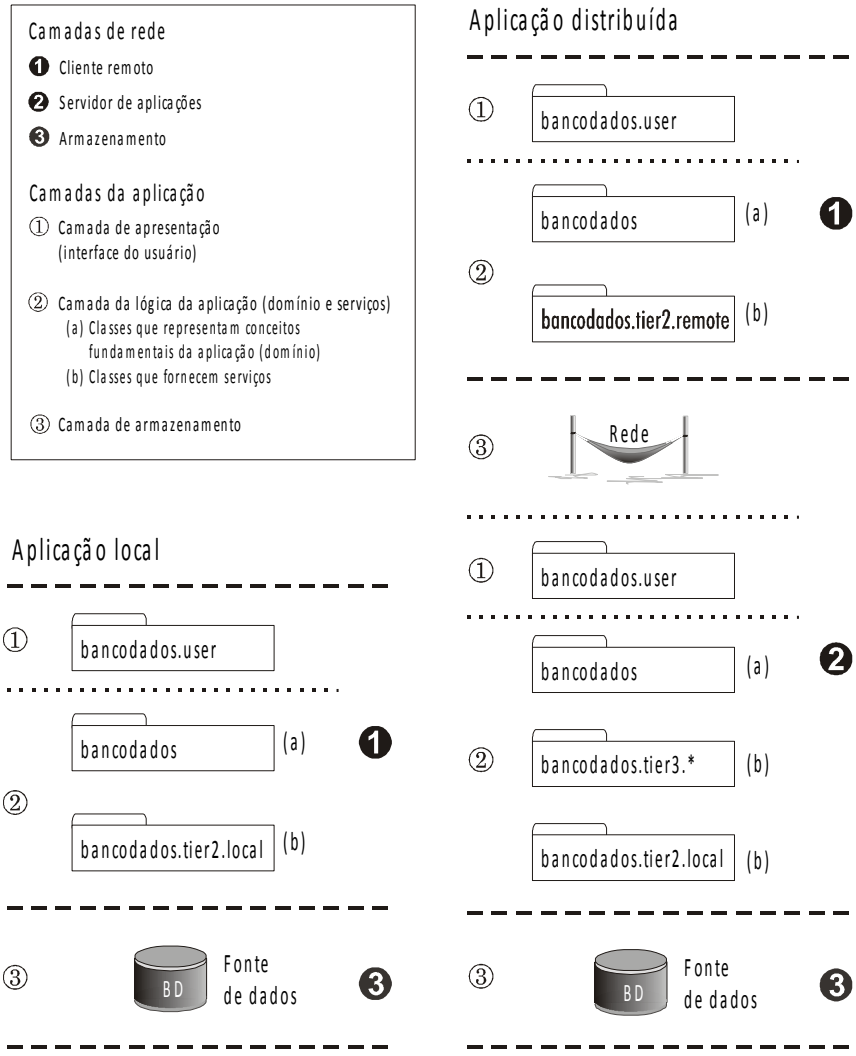
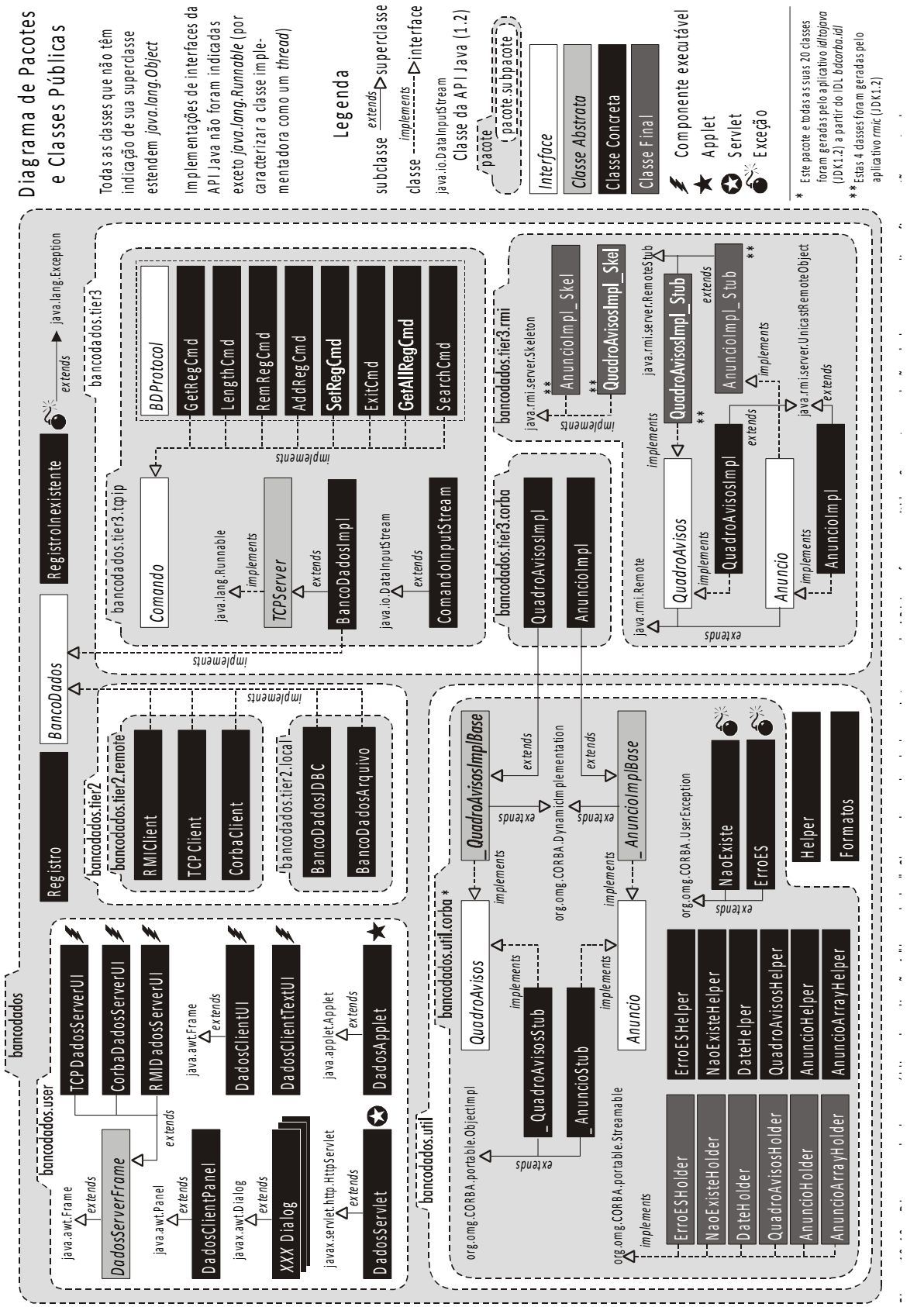


Figura 10-9 – Arquitetura em camadas e pacotes Java/UML das aplicações de banco de dados

As classes restantes do pacote `bancodados.user` são janelas de dialogo e adaptadoras de eventos usadas pelas aplicações gráficas. A figura 10-10 mostra todas as classes e todos os pacotes da aplicação.



A figura 10-11 mostra um cenário, de uma aplicação rodando como applet e usando CORBA para intermediar o acesso ao banco de dados.

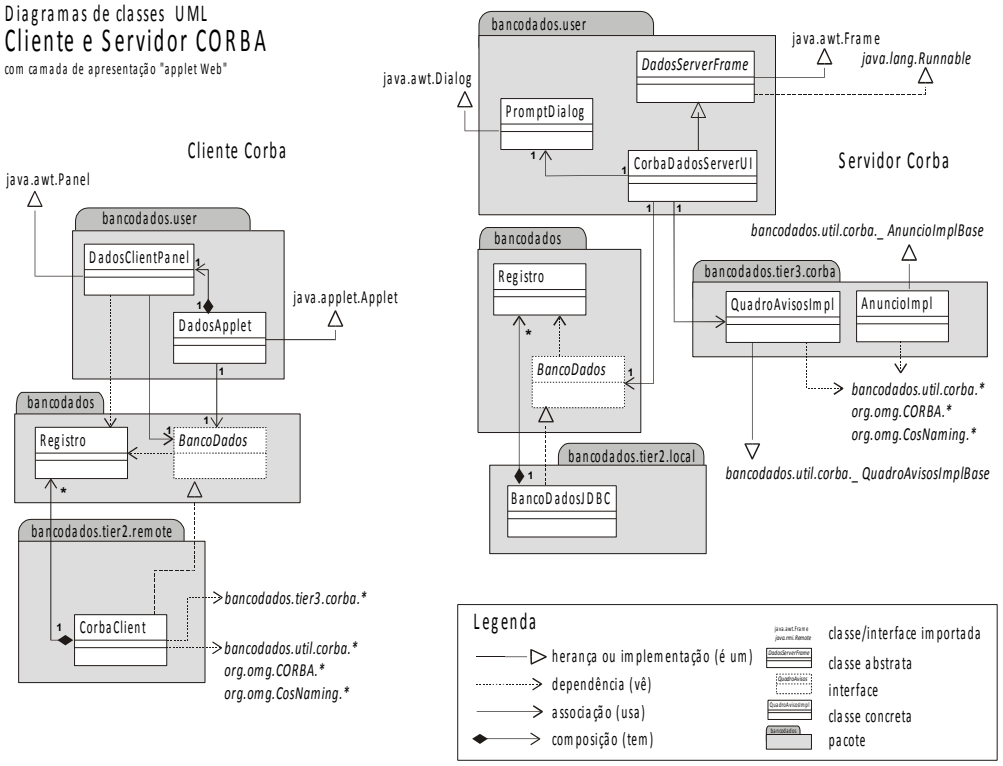


Figura 10-11 - Diagramas de classes. Cliente e Servidor CORBA com camada de apresentação "applet Web".

10.5.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.

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.

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 10-1 – 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;
      }
  }
```


10.6.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.

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 10-2 – 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.

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

[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.

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 -->
(...)
<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 10-4):

```
<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, ...
também!</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.

10.7.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/`.

10.8. Onde e quando usar cada cenário?

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 intermedeia 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[2], 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.

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.

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

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[3] 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.

10.9.Resumo

Este módulo analisou um exemplo de aplicação Java que utiliza as tecnologias exploradas em módulos 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.

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

Os resultados da análise 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.

Tabela 10-3

	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.
Flexibilidade de desenvolvimento[4]	Total (por ser aplicação de referência).	Limitada por herança (precisa estender a classe Applet).	Limitada por polimorfismo (precisa implementar interface Servlet)[5].
Facilidade de desenvolvimento[6]	Maior (referência).	Mesmo nível que aplicação de referência.	Requer conhecimentos de HTML e HTTP.
Quantidade de modificações e código adicional a escrever[7]	Nenhum (referência).	Poucas modificações (usa mesma interface gráfica).	Algumas modificações (precisa gerar HTML).
Suporte em JDK 1.2 (Java 2)	java.awt e javax.swing (ambos do núcleo JDK)	java.applet (núcleo)	javax.servlet (extensão)
Suporte em JDK 1.1	java.awt (núcleo) e com.sun.java.swing (extensão)	java.applet (núcleo)	javax.servlet (extensão)
Suporte em JDK 1.0	java.awt (núcleo)	java.applet (núcleo)	Não suportado.

[4] 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.

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

[6] 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.

[7] Em relação à aplicação *Windows*.

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 grande interação em tempo real no cliente (desenhos, por exemplo). O inconveniente é o download de um programa grande e a falta de suporte por alguns browsers. As restrições podem ser atenuadas por applets assinados.

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 *download* e *instalação* por parte do cliente (o que é raro em aplicações Web, mesmo em intranets).

Fonte:

Helder L. S. da Rocha. *Dissertação de Mestrado. Apêndice C.* Universidade Federal da Paraíba, Campus de Campina Grande, 1999.

