JAVA 7

Bancos de dados e objetos remotos

ESTE MÓDULO APRESENTA A API JDBC, que permite o acesso a bancos de dados relacionais. Apresenta também a API CORBA e RMI, que oferecem acesso a objetos remotos e à chamada de métodos remotos.

Tópicos abordados neste módulo

- O pacote `java.sql` (JDBC)
- Visão geral de ODBC e SQL
- Acesso a bancos de dados usando Java
- Os pacotes de objetos remotos `java.rmi`, `javax.rmi` e `org.omg.CORBA`
- Uso de objetos remotos e invocação de métodos
- *RMI – Remote Method Invocation*
- *CORBA – Common Object Request Broker Architecture*

Índice

Objetivos

No final deste módulo você deverá ser capaz de:

- construir uma aplicação ou applet Java que realize acesso a bancos de dados relacionais
- construir uma camada intermediária usando CORBA ou RMI
- registrar objetos em um servidor para acesso remoto (CORBA ou RMI)
- obter referências para objetos remotos
- chamar métodos em objetos remotos

7.1. JDBC

Java Database Connectivity – JDBC, é uma interface baseada em Java para acesso a bancos de dados através de SQL. Oferece uma interface uniforme para

bancos de dados de fabricantes diferentes, permitindo que sejam manipulados de uma forma consistente. O suporte a JDBC é proporcionado por uma API Java padrão `java.sql` e faz parte da distribuição Java. Usando JDBC, pode-se obter acesso direto a bancos de dados através de applets e outras aplicações Java.

JDBC é uma interface de nível de código. Consiste de um conjunto de classes e interfaces que permitem embutir código SQL como argumentos na invocação de seus métodos. Por oferecer uma interface uniforme, independente de fabricante de banco de dados, é possível construir uma aplicação Java para acesso a qualquer banco de dados SQL. A aplicação poderá ser usada com qualquer banco de dados que possua um driver JDBC: *Sybase*, *Oracle*, *Informix*, ou qualquer outro que ainda não inventado, desde que implemente um driver JDBC.

Esta seção apresentará uma introdução a JDBC e as principais classes e interfaces do pacote `java.sql`. Na seção seguinte, desenvolveremos uma aplicação de banco de dados, semelhante à do capítulo anterior (acesso a dados em arquivo de texto) mas, desta vez, oferecendo acesso a um banco de dados relacional.

Um banco de dados relacional pode ser definido de maneira simples como um conjunto de tabelas (com linhas e colunas) onde as linhas de uma tabela podem ser relacionadas a linhas de outra tabela. Este tipo de organização permite a criação de modelos de dados compostos de várias tabelas. Umas contendo informação, outras contendo referências que definem relações entre as tabelas de informação. O acesso às informações armazenadas em bancos de dados relacionais é em geral bem mais eficiente que o acesso a dados organizados seqüencialmente ou em estruturas de árvore.

SQL

Para utilizar um banco de dados relacional, é preciso ter um conjunto de instruções para recuperar, atualizar e armazenar dados. A maior parte dos bancos de dados relacionais suportam uma linguagem padrão chamada SQL – *Structured Query Language*. O conjunto de instruções SQL foi padronizado em 1992 para que as mesmas instruções pudessem ser usadas por bancos de dados diferentes. Mas vários fabricantes possuem extensões e certas operações mais específicas possuem sintaxes diferentes em produtos de diferentes fabricantes.

Poucos sistemas implementam totalmente o SQL92. Existem vários níveis que foram definidos durante a padronização do SQL em 1992. O conjunto mínimo de instruções é chamado de *entry-level* e é suportado por JDBC.

Nas seções seguintes, apresentaremos os seis principais comandos da linguagem SQL. Este não é um guia completo. Apresentamos apenas o suficiente para permitir a compreensão dos exemplos que mostraremos em JDBC.

CREATE, DROP

Antes de ilustrar a recuperação e atualização de dados em uma tabela, precisamos ter uma tabela. Para criar uma nova tabela em um banco de dados, usamos a instrução `CREATE TABLE`. A sintaxe básica desta instrução é:

```
CREATE TABLE nome_da_tabela
     (nome_coluna tipo_de_dados [modificadores],
     [nome_coluna tipo_de_dados [modificadores], ... ])
```

Os modificadores são opcionais e geralmente dependentes de fabricante de banco de dados. A maior parte das implementações suporta: `NOT NULL`, `PRIMARY KEY` e `UNIQUE` como modificadores:

```
CREATE TABLE anuncios (numero INT PRIMARY KEY,
                       data DATE,
                       texto CHAR(8192),
                       autor CHAR(50))";
```

A sintaxe exata do `CREATE` é dependente de banco de dados. Todos suportam a sintaxe principal (`CREATE TABLE`). As incompatibilidades surgem no suporte a tipos de dados e modificadores. A instrução acima funciona com o driver ODBC do *Microsoft Access*, via ponte ODBC-JDBC. Não funciona, no entanto com o driver JDBC do banco de dados *mSQL*, já que o *mSQL* não suporta o tipo `DATE`. Também não funciona com o driver ODBC para arquivos de texto, da *Microsoft* que não suporta o modificador `PRIMARY KEY` nem campos com mais de 255 caracteres.

Para remover uma tabela do banco de dados, pode-se usar a instrução `DROP`. A sintaxe é simples:

```
DROP TABLE nome_da_tabela
```

INSERT, UPDATE, DELETE

Depois que uma tabela é criada, dados podem ser inseridos usando a instrução `INSERT`. É preciso respeitar os tipos de dados definidos para cada coluna de

acordo com a estrutura definida previamente para cada tabela. A sintaxe básica do INSERT é:

```
INSERT INTO nome_da_tabela (nome_da_coluna, ..., nome_da_coluna)
       VALUES (valor, ..., valor)
```

No lugar de valor, pode ser passado toda uma expressão SQL que resulte em um valor. Um exemplo do uso de INSERT, na tabela criada na seção anterior seria:

```
INSERT INTO anuncios
       VALUES (156, '13/10/1998', 'Novo anuncio!', 'Fulano');
```

O apóstrofe (') é usado para representar strings.

UPDATE permite que dados previamente inseridos sejam modificados. Sua sintaxe básica é:

```
UPDATE nome_da_tabela
       SET nome_da_coluna = valor,
           ...,
           nome_da_coluna = valor
       WHERE expressao_condicional
```

No lugar de valor, pode ser passada toda uma expressão SQL que resulte em um valor ou a palavra reservada NULL. Eis um exemplo do uso de UPDATE:

```
UPDATE anuncios
       SET texto = 'Em branco!',
           autor = ''
       WHERE codigo > 150
```

Para remover registros (linhas da tabela), usa-se DELETE:

```
DELETE FROM nome_da_tabela
       WHERE expressao_condicional
```

Por exemplo, a instrução:

```
DELETE FROM anuncios
       WHERE texto LIKE '%Para o Lixo%'
```

apaga todos os registros cujo texto contém 'Para o Lixo'.

SELECT

O comando SQL mais freqüentemente utilizado é SELECT. Com ele é possível selecionar linhas (registros) de um banco de dados de acordo com uma determinada condição. A sintaxe básica de SELECT é:


```
SELECT nome_da_coluna, ..., nome_da_coluna
       FROM nome_da_tabela
       WHERE expressão_condicional
```

A lista de colunas a serem selecionadas pode ser substituída por "*" se todas as colunas serão selecionadas. A sintaxe mostrara acima é básica. Para selecionar todos os números e textos de registros escritos pelo autor Mefisto, pode-se fazer:

```
SELECT codigo, texto
       FROM anuncios
       WHERE autor = 'Mefisto'
```

Junções

SQL permite a realização de procedimentos bem mais complexos que os listados acima. A declaração `SELECT`, por exemplo, pode ser utilizada para realizar pesquisas em mais de uma tabela ao mesmo tempo, por exemplo:

```
SELECT anuncios.texto
       FROM anuncios, cadastros
       WHERE anuncios.autor = cadastros.autor
```

retorna o texto da tabela `anúncios` apenas quando o autor consta também da tabela `cadastros`. No caso das junções, o nome da tabela precede o nome da coluna que é separada com um ponto ".".

Pesquisa entre resultados

Pesquisas mais refinadas podem ser realizadas entre os resultados de uma pesquisa anterior colocando em cascata várias instruções `SELECT`:

```
SELECT texto
    FROM anuncios
    WHERE autor IN
         (SELECT anuncios.autor
             FROM anuncios, departamentos
             WHERE anuncios.cod_autor = departamentos.cod_autor)
```

ODBC

Para criar e administrar bancos de dados relacionais precisamos ter um ambiente próprio, geralmente fornecido pelo fabricante. Para usar esses bancos de dados dentro de aplicações, precisamos de uma maneira de encapsular o SQL dentro de uma linguagem de programação, já que embora SQL seja eficiente na administração de bancos de dados, ela não possui recursos de uma linguagem de

programação de propósito geral. Usando SQL podemos ter acesso a bancos de dados de uma forma padrão dentro de programas escritos em C ou C++ através da interface ODBC.

ODBC – *Open Database Connectivity* é uma interface de baixo nível baseada na linguagem C que oferece uma interface consistente para a comunicação com um banco de dados usando SQL. Surgiu inicialmente como um padrão para computadores desktop, desenvolvido pela *Microsoft*, mas em pouco tempo tornou-se um padrão de fato da indústria. Todos os principais fabricantes de bancos de dados dispõem de drivers ODBC.

ODBC possui diversas limitações. As principais referem-se à dependência de plataforma da linguagem C, dificultando o porte de aplicações ODBC para outras plataformas. Diferentemente das aplicações *desktop*, onde praticamente domina a plataforma *Windows*, aplicações de rede e bancos de dados freqüentemente residem em máquinas bastante diferentes. A independência de plataforma nesses casos é altamente desejável. Java oferece as vantagens de ODBC juntamente com a independência de plataforma com sua interface JDBC, que apresentaremos na próxima seção.

Muitos bancos de dados já possuem drivers JDBC, porém é possível ainda encontrar bancos de dados que não os possuem, mas têm drivers ODBC. Também, devido a ubiqüidade da plataforma Windows, que contém um conjunto de drivers ODBC nativos, é interessante poder interagir com esses drivers em várias ocasiões, por exemplo, ao montar um banco de dados SQL baseado em arquivos de texto. Como a plataforma Java contém um driver JDBC para ODBC, podemos usar JDBC em praticamente qualquer banco de dados.

Arquitetura JDBC

JDBC é uma versão Java de ODBC. É uma alternativa que acrescenta a portabilidade entre plataformas ao ODBC.

Para que se possa usar JDBC na comunicação com um banco de dados, é preciso que exista um driver para o banco de dados que implemente os métodos JDBC. O driver é uma classe Java que implementa uma interface do pacote `java.sql` chamada `Driver` e um conjunto de interfaces comuns que definem métodos usados na conexão, requisições e resposta.

Para que uma aplicação se comunique com um banco de dados, precisa carregar o driver (pode ser em tempo de execução) e obter uma conexão ao mesmo através. Isto é conseguido através de um método fábrica fornecido pela classe `DriverManager`. Depois de obtida a conexão, pode-se enviar requisições de pesquisa e atualização e analisar os dados retornados usando métodos Java e passando instruções SQL como argumentos. Não é preciso conhecer detalhes do banco de dados em questão. Em uma segunda execução do programa, o programa pode carregar outro driver e utilizar os mesmos métodos para ter acesso a um banco de dados diferente.

Muitos bancos de dados não têm driver JDBC, mas têm driver ODBC. Por causa disso, um driver JDBC para bancos de dados ODBC é fornecido pela *Sun* e incluído na distribuição Java. Com essa ponte JDBC-ODBC é possível usar drivers ODBC através de drivers JDBC. Esse driver é somente um dos quatro tipos diferentes de drivers JDBC previstos pela especificação.

A figura 7-1 ilustra um diagrama em camadas da arquitetura JDBC ilustrando os diferentes tipos de drivers.

Tipos de drivers JDBC

Existem quatro tipos de drivers JDBC [SUN]:

Tipo 1 – drivers que usam uma ponte para ter acesso a um banco de dados. Este tipo de solução geralmente requer a instalação de software do lado do cliente. Um exemplo de driver do tipo 1 é a ponte JDBC-ODBC distribuída pela *Sun* na distribuição Java.

Tipo 2 – drivers que usam uma API nativa. Esses drivers contém métodos Java implementados em C ou C++. São Java na superfície e C/C++ no interior. Esta solução também requer software do lado do cliente. A tendência é que esses drivers evoluam para drivers do tipo 3

Tipo 3 – drivers que oferecem uma API de rede ao cliente para que ele possa ter acesso a uma aplicação *middleware* no servidor que traduz as requisições do cliente em uma API específica ao driver desejado. Esta solução não requer software do lado do cliente.

Tipo 4 – drivers que se comunicam diretamente com o banco de dados usando soquetes de rede. É uma solução puro Java. Não requer código do lado do

cliente. Este tipo de driver geralmente é distribuído pelo próprio fabricante do banco de dados.

A ponte JDBC-ODBC distribuída juntamente com o JDK é um driver do tipo 1. Nos exemplos apresentados neste trabalho, utilizamos esse driver apenas para acesso local através do ODBC nativo do *Windows*. É o mais ineficiente de todos pois não permite otimizações e é dependente das limitações do driver com o qual faz ponte.

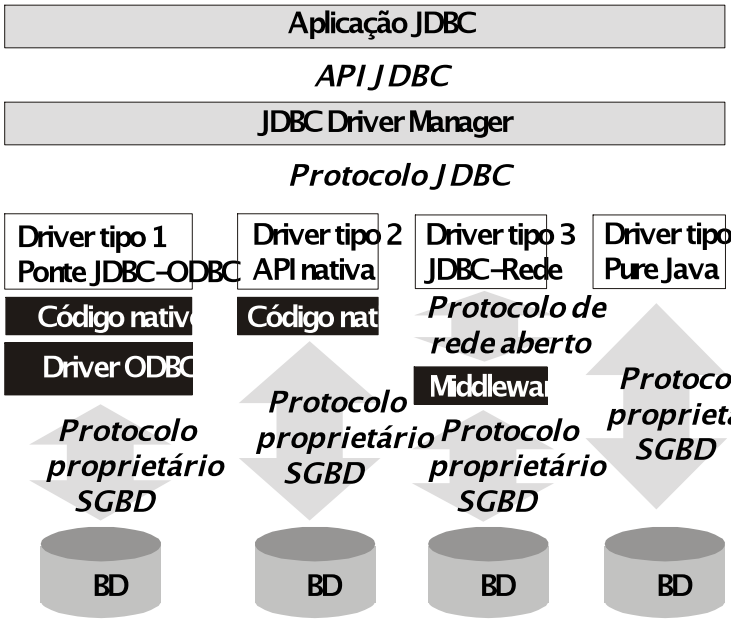

Figura 7-1: Arquitetura JDBC

Nos exemplos apresentados aqui, também utilizaremos um driver do tipo 4: o *Imaginary mSQL* driver. Com este driver, poderemos desenvolver aplicações remotas utilizando somente Java. Se no seu ambiente de trabalho houver um outro banco de dados (*Oracle*, *Informix*, *Sybase*, por exemplo), verifique se o mesmo não possui um driver JDBC próprio (tipos 2, 3 ou 4). A ponte ODBC-JDBC só deve ser usada em último caso já que carrega junto as desvantagens de falta de portabilidade e baixo desempenho.

URL JDBC

Uma aplicação JDBC pode carregar ao mesmo tempo diversos drivers. Para determinar qual driver será usado em uma conexão, uma URL é passada como argumento do método usado para obter uma conexão. Esta URL tem a sintaxe seguinte:

```
jdbc:subprotocolo:dsn
```

O `subprotocolo` é o nome do tipo de protocolo de banco de dados que está sendo usado para interpretar o SQL. É um nome dependente do fabricante. A aplicação usa o `subprotocolo` para identificar o driver a ser instanciado. O `dsn` é o nome que o `subprotocolo` utilizará para localizar um determinado servidor ou

base de dados. Pode ser o nome de uma fonte de dados do sistema local (*Data Source Name*) ou uma fonte de dados remota. Veja alguns exemplos:

```
jdbc:odbc:anuncios
jdbc:oracle:contas
jdbc:msql:clientes
jdbc:msql://alnitak.orion.org/clientes
```

A última URL, que contém um endereço Internet, não pode ser usada em drivers tipo 1. Veja na documentação do seu driver qual o nome correto para o `subprotocolo` e se o mesmo aceita acesso remoto.

Classes essenciais do pacote java.sql

Nesta seção apresentaremos as principais classes e interfaces do pacote `java.sql`, ilustradas na figura 7-2 abaixo. As classes e interfaces abaixo compõem todo o pacote `java.sql` na plataforma Java 2. As interfaces das duas primeiras colunas foram introduzidas somente nesta versão (*Java 2*). São várias interfaces para lidar com BLOBs (*Binary Large Objects*), e utilitários para melhorar a eficiência da transferência de dados.

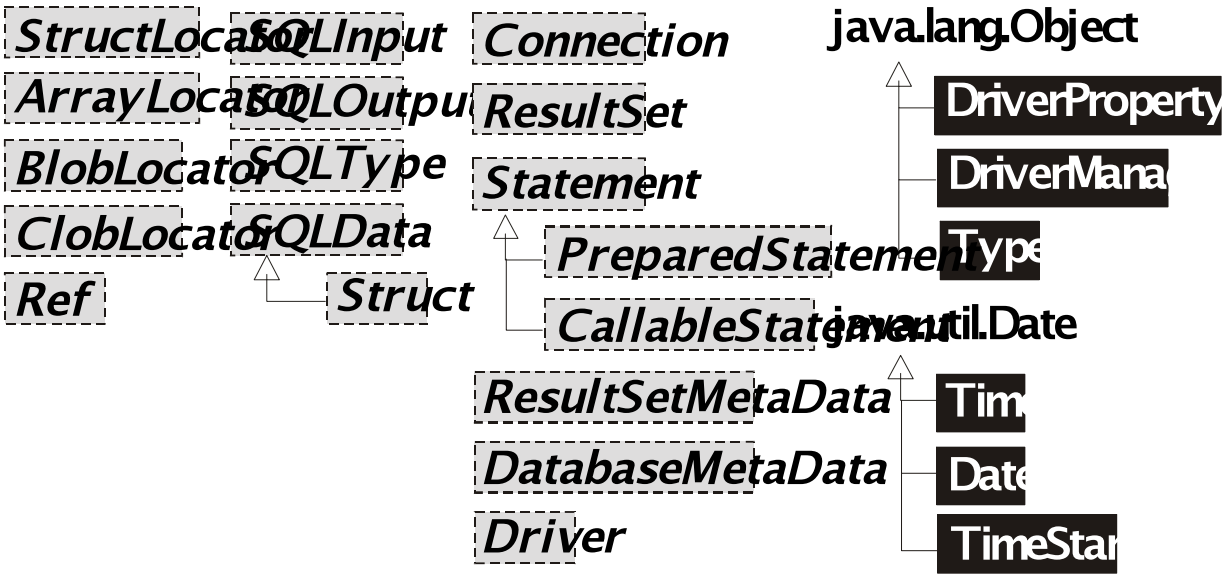

Figura 7-2: Hierarquia de classes java.sql

DriverManager e Driver

A interface `Driver` é utilizada apenas por implementações de drivers (todo driver JDBC implementa `Driver`). A classe `DriverManager` manipula objetos do tipo `Driver`. É utilizada em aplicações de acesso a banco de dados para manter

uma lista de implementações de `Driver` que podem ser usadas pela aplicação. Possui métodos para registrar novos drivers, removê-los ou listá-los. O seu método mais usado é estático e retorna um objeto do tipo `Connection`, que representa uma conexão a um banco de dados, a partir de uma URL JDBC recebida como parâmetro:

```
Connection con =
       DriverManager.getConnection("jdbc:odbc:dados","nome","senha");
```

Connection, ResultSet e Statement

As interfaces `Connection`, `Statement` e `ResultSet` contém métodos que são implementados em todos os drivers JDBC. `Connection` representa uma conexão, que é retornada pelo `DriverManager` na forma de um objeto. `Statement` oferece meios de passar instruções SQL para o sistema de bancos de dados. `ResultSet` lida com os dados recebidos.

Obtendo-se um objeto `Connection`, invoca-se sobre ele o método `createStatement()` para obter um objeto do tipo `Statement`:

```
Statement stmt = con.createStatement()
```

que poderá usar métodos como `execute()`, `executeQuery()`, `executeBatch()` e `executeUpdate()` para enviar instruções SQL ao BD. Existem duas subinterfaces de `Statement` que suportam procedimentos preparados previamente ou e armazenados no servidor: `PreparedStatement` e `CallableStatement`. Eles podem ser obtidos usando:

```
PreparedStatement pstmt = con.prepareStatement();
CallableStatement cstmt = con.prepareCall();
```

Depois de obtido um objeto `Statement`, instruções SQL podem ser enviadas para o servidor que as processará a medida em que as receber.

```
stmt.execute("CREATE TABLE dinossauros "
          + "(codigo INT PRIMARY KEY, genero CHAR(20), especie CHAR(20))");

int linhasModificadas =
    stmt.executeUpdate("INSERT INTO dinossauros "
                     + "(codigo, genero, especie) VALUES "
                     + "(499, 'Fernandosaurus', 'brasiliensis'");

ResultSet cursor =
    stmt.executeQuery("SELECT genero, especie FROM dinossauros "
                    + "WHERE codigo = 355");
```


Em vez de processar as instruções uma por uma, pode ser desejável ou necessário montar várias requisições e processá-las de uma vez. Isto pode ser obtido através do controle da lógica de transações proporcionado pela interface `Connection` através dos métodos `commit()`, `rollback()` e `setAutoCommit(boolean autoCommit)`. Por *default*, as informações são processadas a medida em que são recebidas. Isto pode ser mudado fazendo:

```
con.setAutoCommit(false).
```

Agora várias instruções podem ser acumuladas. Elas só serão processadas quando houver uma instrução `COMMIT` no banco de dados, implementada em Java da forma:

```
con.commit();
```

Se houver algum erro e todo o processo necessitar ser cancelado, pode-se emitir um `ROLLBACK` usando:

```
con.rollback();
```

O método `executeQuery()`, da interface `Statement`, retorna um objeto `ResultSet`. Este objeto implementa um ponteiro para as linhas de uma tabela. Através dele, pode-se navegar pelas linhas da tabela (registros) e recuperar as informações armazenadas nas colunas (campos).

```
ResultSet rs =
      stmt.executeQuery("SELECT Numero, Texto, Data FROM Anuncios");
while (rs.next()) {
    int x = getInt("Numero");
    String s = getString("Texto");
    java.sql.Date d = getDate("Data");
    // faça algo com os valores x, s e d obtidos...
}
```

Os métodos de navegação são `next()`, `previous()`, `absolute(int` *`linha`*`)`, `first()` e `last()`. Existem ainda diversos métodos `getXXX()` e `updateXXX()` para recuperar ou atualizar campos individuais. Os tipos retornados através desses métodos são convertidos de tipos SQL para tipos Java de acordo com a tabela 7-1.

Tabela 7-1

Método ResultSet	**Tipo de dados SQL92**
`getInt()`	`INTEGER`
`getLong()`	`BIG INT`
`getFloat()`	`REAL`

Método ResultSet	Tipo de dados SQL92
`getDouble()`	`FLOAT`
`getBignum()`	`DECIMAL`
`getBoolean()`	`BIT`
`getString()`	`CHAR, VARCHAR`
`getDate()`	`DATE`
`getTime()`	`TIME`
`getTimestamp()`	`TIME STAMP`
`getObject()`	Qualquer tipo

Após o uso, os objetos `Connection`, `Statement` e `ResultSet` devem ser fechados. Isto pode ser feito com o método `close()`:

```
con.close();
stmt.close();
rs.close();
```

DatabaseMetaData

Às vezes é preciso saber informações sobre os dados como: quantas colunas e quantas linhas existem na tabela, qual o nome das colunas, etc. Essas informações podem ser obtidas usando as classes `DatabaseMetaData` e `ResultSetMetaData`, que permitem que se obtenha informações sobre todo o bando de dados ou sobre o conjunto de dados devolvido por um `ResultSet`, respectivamente.

Para obter um objeto de metadados sobre o banco de dados, usa-se o método `getMetaData()` de `Connection`:

```
Connection con;
(...)
DatabaseMetaData dbdata = con.getMetaData();
```

Com este objeto, pode-se obter diversas informações sobre o banco de dados, como nome das tabelas, versão e nome do banco de dados, etc.:

```
String nomeDoProduto = dbdata.getDatabaseProductName();
```

Quando um `ResultSet` é retornado, podemos não saber o número e nomes das colunas. Usando um `ResultSetMetaData`, obtemos essas informações:

```
ResultSet rs;
(...)
ResultSetMetaData meta = rs.getMetaData();
int colunas = meta.getColumnCount();
String[] nomesColunas = new String[colunas];
for (int i = 0; i < colunas; i++) {
    nomesColunas[i] = meta.getColumnName(i);
```



```
}
```

Exercícios

1. Construa uma aplicação Java simples que permita que o usuário envie comandos SQL para um banco de dados e tenha os resultados listados na tela.
2. Crie uma aplicação com o mesmo objetivo que a aplicação do exercício 1, mas, desta vez, faça com que os dados sejam exibidos dentro de um `TextArea` (aplicação gráfica).
3. Crie uma interface completa para o banco de dados anuncios.mdb (disponível em disco) com três `TextFields` e uma `TextArea` para incluir, respectivamente, os campos `número`, `data`, `autor` e `texto` da tabela `anuncios`. O primeiro registro deverá ser mostrado. Implemente quatro botões de navegação. Dois para avançar e voltar um registro e dois para voltar ao início e ao fim do banco de informações.
4. Usando qualquer banco de dados disponível (que tenha driver JDBC ou ODBC) implemente uma tabela `Banco`, com código e nome de correntistas, uma tabela `Correntistas`, com código e saldo de vários correntistas. Crie agora uma interface JDBC (gráfica, se possível) que permita visualizar os dados e realizar operações bancárias (transferências, saques, depósitos, extratos e balanço).

7.2. Construção de uma aplicação JDBC

Nesta seção, apresentamos uma aplicação JDBC usando o mesmo banco de dados do capítulo anterior, desta vez organizado em um sistema de BD relacional. Para reaproveitar toda a interface do usuário e as classes que representam os conceitos fundamentais do programa, criamos uma classe `bancodados.tier2.local.BancoDadosJDBC`, que implementa a interface `bancodados.BancoDados` (veja módulo *Java 10*). Como a interface do usuário usa a interface `BancoDados`, podemos utilizar a classe `BancoDadosJDBC` preservando a mesma interface do usuário que utilizamos no módulo anterior.

Criação das tabelas

Os dados utilizados por esta aplicação serão do mesmo tipo que aqueles manipulados pela aplicação do módulo anterior. Sendo assim, teremos apenas uma tabela no banco de dados com a seguinte estrutura:

Tabela 7-2

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 comuns para garantir a compatibilidade com uma quantidade maior de bancos de dados. Poderíamos ter usado o tipo SQL `DATE`, por exemplo, no lugar de `CHAR (long)`, mas perderíamos compatibilidade com o *mSQL* que não suporta o tipo.

Antes de construir a tabela, é preciso criar uma fonte de dados ODBC e vinculá-las a uma base de dados previamente criada. Veja no apêndice A como criar bases de dados ODBC no *Windows* vinculadas a arquivos *Access* (`.mdb`) e ao sistema de arquivos (texto). Para usar uma base de dados *mSQL*, é preciso criá-la usando as ferramentas disponíveis na aplicação. Ela será acessada diretamente.

Construída a base de dados, podemos executar o programa `bancodados.util.CriaTabelas` ou `CriaTabelas2` para os bancos de dados ODBC baseados em arquivo de texto (as aplicações estão disponíveis no subdiretório `jad/apps/bancodados/util`). Qualquer uma das aplicações requer como parâmetros:

- uma URL JDBC do tipo `jdbc:odbc:`*<nome da fonte de dados>* para acesso ODBC ou `jdbc:msql://localhost/`*<nome da base de dados>* para acesso via *mSQL*.
- nome do usuário
- a senha do usuário

Por exemplo, para criar as tabelas em uma fonte de dados ODBC local chamada `classif`:

```
java bancodados.util.CriaTabelas jdbc:odbc:classif scott tiger
```


Se as tabelas forem criadas com sucesso, será impressa uma mensagem:

```
Tabelas criadas com sucesso!"
```

A listagem abaixo esclarece as partes mais importantes do código-fonte. Veja o código completo com comentários em `jad/apps/bancodados/util`.

O método `main()`, que inicia a aplicação, primeiro faz uma verificação para checar se os parâmetros requeridos foram passados na linha de comando. Caso positivo, inicializa o objeto passando os argumentos de linha de comando para o construtor. Quando o construtor terminar, será chamado o método `criaTabela()`.

```
    public static void main(String[] args) {
       if (args.length < 1) {
          System.out.println
            ("Sintaxe correta: java CriaTabelas url_jdbc [nome] [senha]");
       (...)
          System.exit(1);
       }
       String url = args[0];
       String nome = "";
       String senha = "";

       if (args.length >= 2) {
          nome = args[1];
       }
       if (args.length >= 3) {
          senha = args[2];
       }

       CriaTabelas ct = new CriaTabelas(url, nome, senha);
       ct.criaTabela();
    }
}
```

A aplicação localiza os drivers disponíveis através da informação armazenada em um arquivo `drivers.jdbc` armazenado no diretório de trabalho da aplicação (`jad/apps`). Define ainda variáveis de instância para representar a conexão e o contexto de envio de declarações SQL. O arquivo `drivers.jdbc` permite que novos bancos de dados sejam utilizados e que seus drivers JDBC (classes Java) sejam carregados sem que seja necessária a recompilação do programa. O arquivo relaciona nomes de classes e subprotocolos. Eis a listagem da classe `CriaTabelas.java`:


```
package bancodados.util;
 (...)
public class CriaTabelas {

   // arquivo com drivers disponíveis
   // localizado no diretorio onde roda a aplicacao
   private static final String DRIVERS = "drivers.jdbc";

   private Connection con;              // conexao
   private Statement stmt;;             // declaracao SQL
   private Vector numeros;              // vetor com numeros de registro
```

O construtor da aplicação tem duas partes. Inicialmente, tenta carregar o arquivo onde estão listados os drivers disponíveis. Isto é feito através de um método local chamado `loadDrivers()`. Se consegue, o método retorna uma matriz com o subprotocolo e nome completo da classe Java que implementa o driver. Se falha, usa o driver ODBC (*default*).

```
   public CriaTabelas(String url, String nome, String senha) {

      // nomes de drivers disponíveis: default ponte JDBC-ODBC da Sun
      String driv = "odbc";
      String[][] drivers = {{"odbc", "sun.jdbc.odbc.JdbcOdbcDriver"}};
      boolean temDriver = false;

      try {
         try {
            // método loadDriverNames é local (veja código fonte).
            // Ele lê arquivo de drivers e devolve matriz com
            // [subprotocolo] [nome do driver]
            String[][] listaDrivers = loadDriverNames(DRIVERS);

            // Leu arquivo...  redefine drivers
            drivers = listaDrivers;
            listaDrivers = null;    // libera o objeto

            // agora faça validação da URL recebida pelo cliente
            StringTokenizer st = new StringTokenizer(url, ":");
            String prot = st.nextToken();   // obtem primeiro token
            if (!prot.equals("jdbc"))
               throw new java.net.MalformedURLException
                         ("A URL tem que ser do tipo 'jdbc:'!");

            driv = st.nextToken();  // obtem segundo token

         } catch (FileNotFoundException e) {
```



```
            // se nao existe arquivo de configuracao de drivers,
            // continue com o driver default
            (...)
            temDriver = true;
            // continua...
         }
```

A segunda parte do construtor prossegue verificando se o protocolo da URL passada na linha de comando coincide com o subprotocolo de algum driver disponível. Se coincide, o driver é carregado, caso contrário, uma exceção é provocada e o programa é terminado.

```
         for (int i = 0; i < drivers.length; i++) {
            // verifica se subprotocolo coincide com driver disponivel
            if (drivers[i][0].equals(driv)) {
               Class.forName(drivers[i][1]);    // carrega driver
               temDriver = true; // pelo menos um driver foi carregado
            }
         }

         if (temDriver == false) {
            throw new UnsupportedOperationException
                ("Não há drivers disponíveis para " + url + "!");
         }
```

No final, tendo-se carregado o driver, a URL, nome e senha passados como argumentos de linha de comando são utilizados para se obter uma conexão ao banco de dados (através do objeto `Connection`). Logo em seguida, é obtido um objeto `Statement`. A partir de agora, métodos que executam SQL podem ser chamados.

```
            con = DriverManager.getConnection(url, nome, senha);  // obtem conexao
            stmt = con.createStatement();                         // cria statement

         } catch (Exception e) {
            System.out.println(e);
            System.exit(0);
         }
      }
```

Logo que o construtor termina seu trabalho, o método `criaTabela()` é chamado. Este método simplesmente define um procedimento SQL e o passa como argumento do método execute invocado sobre o objeto `stmt`. Se não ocor-

rer exceção, as tabelas serão criadas e a aplicação imprimirá uma mensagem indicando sucesso.

```
public void criaTabela() {
   String create = "CREATE TABLE anuncios (numero int primary key, " +
                                          " data char(24)," +
                                          " texto char(8192)," +
                                          " autor char(50))";

   try {
      stmt.execute (create);
      System.out.println("Tabelas criadas com sucesso!");
   } catch (SQLException e) {
      e.printStackTrace();
   }
}
```

Inicialização

A figura 7-3 ilustra o diagrama da aplicação. Somente as interfaces que estão na parte superior do desenho (`BancoDados` e `BancoDadosJDBC`) são acessíveis e alteráveis dentro da nossa aplicação. O código que usa `java.sql` está somente na classe `BancoDadosJDBC` que usa `DriverManager` para obter um driver (objeto do tipo `Driver`), através do qual interage com o banco de dados ODBC.

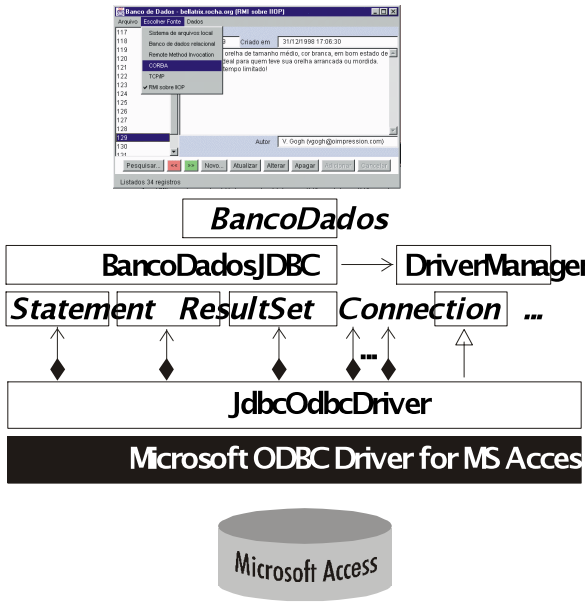

Figura 7-3: diagrama da aplicação mostrando JDBC interfa-

Na classe `BancoDadosJDBC` carregamos a classe e criamos uma instância de um driver de acordo com a URL passada pelo usuário, da mesma forma como fizemos na classe `CriaTabelas`.

O construtor da classe `BancoDadosJDBC` é bem parecido com o construtor da aplicação `CriaTabelas`. Contém as seguintes instruções:

```
public BancoDadosJDBC(String url, String nome, String senha) throws IOException

      // Define formato para todas as datas (dd/mm/aaaa hh:mm:ss)
      df = DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM);
(...)
      try {
         try {
```



```
            String dir = System.getProperty("app.home");
            String[][] listaDrivers = loadDriverNames(dir +
                                      File.separator + DRIVERS);
            // Leu arquivo...  redefine drivers
            drivers = listaDrivers;
(...)
         for (int i = 0; i < drivers.length; i++) {
            if (drivers[i][0].equals(driv)) {
               Class.forName(drivers[i][1]);    // carrega driver
               temDriver = true;
            }
         }
 (...)
         con = DriverManager.getConnection(url, nome, senha);  // obtem conexao
         stmt = con.createStatement();                         // cria statement

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

Os objetos `con` e `stmt` são do tipo `Connection` e `Statement`, respectivamente (veja o código). Foram declarados como variáveis de instância do objeto `BancoDadosJDBC`.

A instrução `Class.forName()` carrega a classe do driver pelo nome. O `DriverManager` recebe uma URL JDBC, informada pelo cliente, localiza um driver, cria uma instância dele e obtém um objeto do tipo `Connection`. Depois, usa o método `createStatement()` para obter um objeto `Statement` que será usado na implementação dos métodos de `bancodados.BancoDados`.

Acesso ao banco de dados

A referência `stmt` é usada em todos os métodos. Alguns retornam uma tabela com valores, outros não. O método `executeUpdate` pode ser usado para inserir registros na implementação de `addRegistro()`, que acrescenta um novo registro no banco de dados:

```
public synchronized void addRegistro(String texto, String autor) {
      ResultSet rs;
      int numero = getProximoNumeroLivre();
      java.util.Date data = new java.util.Date();
      String quando = df.format(data);
      String insert = "INSERT INTO anuncios VALUES (" + numero + ", '"
                                                     + quando + "', '"
```



```
                                                    + texto +  "', '"
                                                    + autor +  "')";
      try {
         stmt.executeUpdate(insert);
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
```

O método `getRegistros` seleciona todos os números de registro do servidor e os coloca em um `Vector`. Veja a implementação JDBC e compare com a implementação mostrada no ultimo capítulo. Neste caso, espera-se o retorno de uma tabela de dados como resposta, usamos o método `executeQuery()`, que retorna um `ResultSet` cujo ponteiro pode ser adiantado com o seu método `next()`. Para cada posição (que corresponde a uma linha da tabela `ResultSet`), usamos os métodos `getXXX()` apropriados para ler inteiros, strings e datas.

```
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 implementado de forma mais simples ainda. É só 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;
        }
     }
```

Interface do usuário

A única alteração necessária na interface do usuário para o acesso ao banco de dados JDBC, é o fornecimento de uma interface de entrada de dados para que o cliente possa escolher a fonte de dados ao qual quer conectar-se, seu nome e senha de acesso. Na aplicação gráfica, utilizamos uma janela de diálogo para coletar esses dados (veja figura 10-2c). Na aplicação de texto, utilizamos a linha de comando:

```
---- Cliente de Banco de Dados ----
Digite o numero correspondente a forma de conexao:
1) Sistema de arquivos local
2) Banco de dados relacional
3) Remote Method Invocation
4) CORBA
5) TCP/IP
6) RMI sobre IIOP
10) Sair do programa
--> 2
Informe a URL JDBC para conexao: jdbc:odbc:txtdados1
Informe o id do usuario para acesso [nenhuma]: helder
Informe a senha para acesso [nenhuma]: kz9919
```

e passamos as informações recebidas na construção do objeto:

```
(...)
case 2:     // JDBC
    System.out.print("Informe a URL JDBC para conexao: ");
    System.out.flush();
    String url = br.readLine();
    System.out.print("Informe o id do usuario para acesso [nenhuma]: ");
    System.out.flush();
    String nome = br.readLine();
```



```
        System.out.print("Informe a senha para acesso [nenhuma]: ");
        System.out.flush();
        String senha = br.readLine();
        System.out.print("Conectando a " + url + "...");
        System.out.flush();
        client = new BancoDadosJDBC(url, nome, senha);
        System.out.println("Conectado a " + url + "!");
        menuOpcoes();
    break;
    (...)
```

A referência `client`, no código acima, é do tipo `bancodados.BancoDados`.

7.3. CORBA

A última seção apresentou uma solução que permitia transformar uma aplicação de uso local em uma aplicação distribuída. Para atingir esse objetivo, tivemos que elaborar regras de comunicações próprias e implementar um cliente e servidor específicos, conforme o protocolo estabelecido.

Utilizando uma tecnologia de objetos distribuídos, podemos obter os mesmos resultados usando um protocolo padrão e desenvolvendo um código mais simples e com maiores possibilidades de reutilização. Esta seção apresenta uma breve introdução à tecnologia de objetos distribuídos CORBA e como poderíamos utilizá-la com a linguagem Java. Na seção seguinte, mostraremos como, através de um procedimento passo-a-passo, podemos implementar uma camada intermediária baseada em CORBA para a aplicação de banco de dados explorada nos módulos e seções anteriores.

Por que usar CORBA?

No módulo anterior vimos que é possível implementar aplicações distribuídas independentes de plataforma em Java usando as classes do pacote `java.net`. Para atingir nossa meta tivemos que desenvolver um protocolo e criar clientes e servidores que se utilizassem desse protocolo para estabelecer um canal de comunicação. Tivemos que definir portas, administrar *threads* individuais para cada conexão e garantir a correta abertura e fechamento dos soquetes e fluxos de dados. Usando agora a tecnologia CORBA, nossa meta é obter a mesma conectividade, mas reduzindo de forma significativa a complexidade.

CORBA oferece uma infraestrutura que permite a invocação de operações em objetos localizados em qualquer lugar da rede como se fossem locais à aplicação que os utiliza. Assim sendo, poderemos invocar métodos remotos como se fossem locais e não precisar se preocupar com protocolos, portas, localidades, *threads* ou controle de fluxos de dados.

Mas CORBA oferece mais que isto. Oferece também independência em relação à linguagem na qual foram desenvolvidos os objetos. A especificação CORBA 2.2 permite que programas distribuídos tenham objetos escritos em *C*, *C++*, *Smalltalk*, *COBOL* e *Java*, e que todos possam se comunicar entre si. Portanto, CORBA é uma ótima solução para desenvolver clientes Java para aplicações legadas, escritas em *C*, *C++*, *COBOL* ou *Smalltalk* e situadas em máquinas remotas.

Fundamentos de CORBA e IDL

CORBA – Common ORB Architecture (arquitetura comum do ORB) é uma especificação que estabelece uma forma de comunicação entre aplicações independente da plataforma onde residem e da linguagem em que foram escritas. O coração da arquitetura CORBA é o *ORB – Object Request Broker* (corretor de requisição de objetos), um "barramento" de mensagens que intercepta as requisições dos clientes CORBA e invoca operações em objetos remotos, passando parâmetros e retornando os resultados da operação ao cliente. Em suma, ORBs servem para transferir objetos de um lugar para outro. Eles ORBs cuidam da forma como os objetos são transferidos deixando os detalhes ocultos ao programador ou usuário. Um ORB, roda em uma única máquina, mas pode se conectar a todos os outros ORBs disponíveis em uma rede TCP/IP, usando o protocolo IIOP (*Internet Inter-ORB Protocol*).

A especificação CORBA é desenvolvida pelo *OMG – Object Management Group*, o maior consórcio da indústria de computadores com mais de 750 membros (1997). A OMG é uma organização sem fins lucrativos cujos objetivos são promover teoria e prática da engenharia de software orientada a objetos, oferecendo uma arquitetura comum para o desenvolvimento de aplicações independente de plataformas de hardware, sistemas operacionais e linguagens de programação. A meta da OMG visa reduzir a complexidade, custos e como conseqüência levar ao desenvolvimento de novas aplicações.

Essencial na tecnologia CORBA está a noção de transparência que ocorre de duas formas. *Transparência em relação à localidade* estabelece que deve ser tão simples invocar operações em um objeto remoto como seria fazer o mesmo se aquele objeto estivesse disponível localmente. *Transparência em relação à linguagem de programação* garante a liberdade para implementar a funcionalidade encapsulada em um objeto usando a linguagem mais adequada, seja qual for o motivo. Isto é conseguido separando a implementação do objeto da sua interface pública de acesso, que é definida usando uma linguagem de definição de interface (IDL – *Interface Definition Language*) neutra quanto à implementação.

IDL – Interface Definition Language

A linguagem OMG IDL é uma linguagem declarativa que oferece uma sintaxe e nomes genéricos para representar estruturas comuns como módulos, interfaces, métodos, tipos de dados, etc. em objetos CORBA. O objeto é completamente especificado na IDL. Para ter utilidade, deve haver um *mapeamento* entre o IDL e a linguagem de programação na qual os objetos estão implementados. Fabricantes de ORBs fornecem um *compilador IDL* cuja finalidade é gerar módulos de suporte às operações declaradas na interface na linguagem de implementação.

Um arquivo IDL declara a interface pública de um objeto. Informa o nome das operações suportadas pelo objeto, o tipo de seus parâmetros, o tipo de suas variáveis públicas e os tipos que são retornados após a invocação de um método. A listagem a seguir mostra um exemplo de OMG IDL com as palavras reservadas em negrito.

```
module BDImagens {       // módulo

   exception ErroEntradaSaida{};          // exceções
   exception FimDeArquivoPrematuro{};

   interface Registro {        // interface
      long getCodigo();
      string getNome();
      string getTipo();        // operações
   };

   interface BancoDados {
      void setRegistro(in long cod, in string nome, in string tipo)
          raises (ErroEntradaSaida);
```



```
        Registro getRegistro()
            raises (ErroEntradaSaida, FimDeArquivoPrematuro);
    };

    interface Prancheta {
        typedef sequence <octet> Color;      // definição de tipos
        typedef sequence <long>  IntArray;

        // Retorna a imagem do desenho.
        IntArray getDesenho();

        // Retorna a cor atual.
        Color getCorAtual();

        // Redefine a imagem do desenho.
        void setDesenho(in IntArray newBitmap);
    };
};
```

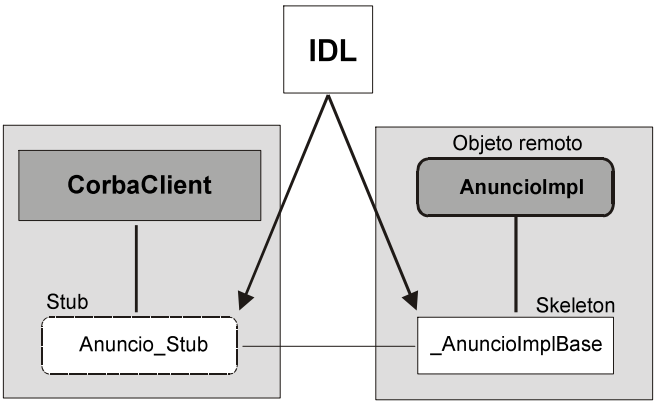

Figura 7-4 Um mesmo IDL define o stub na linguagem do cliente e um skeleton na linguagem do objeto remoto, permitindo que o cliente invoque métodos no objeto remoto mesmo estando escrito em outra linguagem.

A IDL da OMG tem sua sintaxe derivada do C++, mas contém palavras-chave diferentes e específicas. Um arquivo IDL é tudo que um programador necessita para implementar o cliente para um determinado objeto. O compilador IDL, cria os arquivos fonte que implementam a interface para o cliente e o servidor do objeto, na linguagem específica do ORB que fornece o compilador. Cada compilador obedece a um determinado mapeamento IDL-linguagem.

Para o cliente, são gerados arquivos *stub* – que contêm código que realiza a tarefa de intermediar, de forma transparente ao programador e usuário, as operações entre o cliente e o ORB. O *stub* dá ao cliente a ilusão de estar acessando diretamente o objeto remoto, transformando os tipos de dados da linguagem de programação no qual foi criado em um formato de dados interpretado pelo ORB como uma mensagem ao objeto remoto.

Do outro lado, no servidor, arquivos *skeleton* fazem o trabalho inverso, convertendo a requisição na linguagem de programação e tipos de dados locais ao servidor. Arquivos auxiliares também podem ser gerados pelo compilador de forma a tornar a implementação do servidor de objetos transparente em relação à existência do *skeleton*.

Um mesmo IDL pode ser usado na geração de *stubs* e *skeletons* de linguagens diferentes (figura 7-4), permitindo a comunicação de aplicações implementadas em linguagens distintas. Por exemplo, pode-se gerar um *stub* de cliente Java para um objeto com interface definida em uma determinada IDL usando um compilador com mapeamento IDL-Java. Usando o mesmo IDL, e um compilador com mapeamento IDL-COBOL, pode-se criar um *skeleton* para um servidor de objetos remotos em COBOL, fazendo assim uma aplicação Java interagir de forma transparente com uma aplicação COBOL.

Geralmente, objetos CORBA em um ambiente distribuído são servidos por ORBs diferentes. A comunicação entre ORBs é realizada através de protocolos Inter-ORB (*IOP – Inter-ORB Protocol*). Na Internet e redes TCP/IP a conectividade é possível via IIOP – *Internet Inter ORB Protocol*.

Usando CORBA com Java

Para utilizar CORBA com Java é preciso ter um ORB que implemente um mapeamento IDL-Java. Há uma correspondência entre construções IDL e estruturas presentes na linguagem Java. A tabela abaixo mostra o mapeamento entre tipos de dados, definido na especificação CORBA 2.2. Este mapeamento é utilizado pelo compilador na geração de *stubs* e *skeletons*.

Tabela 7-3

Tipo IDL	Tipo Java
boolean	boolean
char	char
wchar	char
octet	byte
short/unsigned short	short
long / unsigned long	int
long long / unsigned long long	long
float	float
double	double

Estruturas como pacotes, classes, interfaces, objetos serializados, exceções também têm uma correspondência simples em IDL. Como Java é uma linguagem orientada a objetos, a conversão de uma interface IDL em uma interface Java é muito simples. Veja na tabela 7-4, algumas estruturas Java e seus pares em IDL:

Tabela 7-4

IDL	Java
module	package
interface	class, interface
raises	throws
exception	Exception
string	String
void	void
enum, union, struct	class
sequence <octet>,	byte[], int[], ...
sequence <long>, ...	(vetores e objetos serializados)
:	extends
const (dentro de interfaces)	static final

A listagem a seguir representa três interfaces Java e duas exceções que podem ser representadas pela interface IDL listada no início desta seção. Veja e compare as duas listagens.

```
package BDImagens;

class ErroEntradaSaida extends Exception {}
class FimDeArquivoPrematuro extends Exception {}

interface Registro {
    public int getCodigo();
    public String getNome();
    public String getTipo();
}

interface BancoDados {
    public void setRegistro(int cod, String nome, String tipo)
           throws ErroEntradaSaida;

    public Registro getRegistro()
           throws ErroEntradaSaida, FimDeArquivoPrematuro;
}

interface Prancheta {
    public int[] getDesenho();
```



```
    public byte[] getCorAtual();
    public void setDesenho(int[] newBitmap);
}
```

Já existem vários ORBs que oferecem suporte a Java. Objetos CORBA desenvolvidos para uma ORB podem não rodar em outro ORB pois poderá haver diferenças na implementação do IDL em ORBs de fabricantes diferentes. O IDL, porém, sempre pode ser usado para desenvolver um objeto compatível com o ORB disponível. Neste trabalho, utilizaremos o ORB nativo do Java 2 por ser gratuito e disponível automaticamente com a distribuição Java.

O suporte a CORBA no Java 2 está nos pacotes e subpacotes de `org.omg.CORBA`, nos pacotes `org.omg.CosNaming` e `omg.CosNaming.NamingContext-Package` (serviço de registro de nomes), e nas ferramentas `idltojava`, `idlj` (compiladores IDL-Java) e `tnameserv` (servidor de registro de nomes).

Exercícios

5. Defina uma interface IDL para enviar requisições SQL remotas para aplicação proposta no exercício 1 (ou 2). Implementa a aplicação em Java e faça os testes, preferencialmente usando máquinas diferentes.
6. Crie uma interface IDL para a aplicação bancária e implemente o acesso remoto nos serviços bancários.

7.4. Construção de aplicações distribuídas com CORBA

A melhor maneira de compreender o funcionamento de Java com CORBA é através de exemplos. O restante desta seção será dedicada à implementação de uma camada CORBA na aplicação de banco de dados vista na seção anterior.

Aplicação de banco de dados

Esta aplicação consiste do quadro de anúncios classificados, semelhante ao apresentado nos capítulos anteriores, mas com um servidor intermediário CORBA. O cliente pode, através de um cliente remoto, realizar operações como acrescentar um novo anúncio, listar os anúncios existentes, alterá-los, apagá-los, pesquisar, etc. Para isto, é preciso que ele obtenha uma referência remota do objeto que representa o quadro de avisos e invocar os seus métodos.

Interface IDL

O primeiro passo é definir a interface IDL que irá descrever os objetos que serão usados ou implementados. Criamos um arquivo `bdcorba.idl` com o seguinte conteúdo, representando as classes `Registro` e `BancoDados` da aplicação no servidor pelas interfaces IDL `Anuncio` e `QuadroAvisos`, respectivamente:

```
module bancodados {
 module util {
  module corba {

   // definição de tipo Date serializado (byte[])
   typedef sequence <octet> Date;
   typedef sequence <octet> AnuncioArray;

   // tratamento para IOException e RegistroInexistente
   exception ErroES{};
   exception NaoExiste{};

   // definição de objeto utilizado pelo quadro de avisos
   interface Anuncio {
      long getNumero();        // retorna numero do anuncio
      string getTexto();       // retorna conteudo do anuncio
      Date getData();          // retorna data postagem do anuncio
      string getAutor();       // retorna endereco do anunciante
   };

   // quadro de avisos
   interface QuadroAvisos {
      long length() raises (ErroES);
      void addRegistro(in string anuncio,
                       in string contato) raises (ErroES);
      Anuncio getRegistro(in long numero)
           raises (ErroES, NaoExiste);      // obtem registro
      boolean setRegistro(in long numero,
                          in string anuncio,
                          in string contato) raises (NaoExiste);
      boolean removeRegistro(in long numero) raises (NaoExiste);
      AnuncioArray getRegistros(in string textoProcurado);
      AnuncioArray getAllRegistros();
   };
  };
 };
};
```


Anuncio e `QuadroAvisos` são interfaces equivalentes a `Registro` e `BancoDados`. Usamos nomes diferentes para evitar conflitos (pois o IDL irá gerar classes com esses nomes) e podermos importar todo o pacote `bancodados` nas classes utilizadas.

O arquivo IDL possui quatro partes. Todas estão dentro do submódulo "`corba`" que será implementado como um pacote Java pelo compilador IDL. A primeira declaração:

```
typedef sequence <octet> Date;
```

define o tipo `Date`, como uma seqüência de bytes. Esta seqüência, em Java, poderá ser convertida em um objeto `java.util.Date` na nossa implementação. As declarações:

```
exception ErroES{};
exception NaoExiste{};
```

definem exceções provocadas pelos métodos da interface `QuadroAvisos`. As exceções definidas pelo usuário serão implementadas pelo compilador IDL como classes finais que estendem `org.omg.CORBA.UserException`.

As duas declarações seguintes

```
interface Anuncio { ... };
interface QuadroAvisos { ... };
```

definem as interfaces aos objetos que o cliente terá acesso. A definição de `Anuncio` é necessária porque o `QuadroAvisos` retorna um objeto deste tipo no método `getRegistro()`. Dentro de cada definição estão declaradas as operações (métodos) que poderão ser invocadas remotamente sobre os objetos `QuadroAvisos` e `Anuncio`.

Vejamos algumas das operações declaradas em `QuadroAvisos`:

```
boolean length() raises (ErroES);

void addRegistro(in string anuncio,
                 in string contato) raises (ErroES);

Anuncio getRegistro(in long numero) raises (ErroES, NaoExiste);
```

A semelhança com métodos Java é bastante grande. Observe que `raises` tem a mesma função que `throws`. Veja o formato dos parâmetros da operação `setRegistro`. Os parâmetros precedidos por "`in`" são semelhantes à parâmetros de métodos Java, passados pelo objeto que invoca a operação. Os parâmetros pre-

cedidos por "`out`" e "`inout`" (não utilizados aqui) não têm equivalente em Java. Esse indicador significa que o valor para o parâmetro será fornecido pelo objeto invocado. O suporte a esse tipo de operação é feito através de objetos especiais (*Holder*) criados pelo compilador IDL-Java.

Compilação IDL-Java

O segundo passo é compilar o arquivo IDL para gerar o código de suporte ao cliente (*stub*) e código de suporte aos objetos no servidor (*skeleton*). No nosso exemplo, tanto o cliente quanto o servidor serão desenvolvidos em Java, mas, nada impede que se utilize o mesmo IDL para gerar um servidor em C, C++, Smalltalk ou COBOL para a comunicação com um cliente Java e vice-versa.
O Java 2 fornece um compilador IDL chamado `idltojava` (`idltojava.exe` em *Windows*). Para compilar o `bdcorba.idl`, faça:

```
idltojava -fno-cpp bdcorba.idl
```

Com o arquivo IDL no subdiretório `jad/apps/`, o compilador terá criado o subdiretório "`bancodados/util/corba`" abaixo do diretório atual. Este subdiretório representa um pacote que contém as seguintes classes:

`DateHolder.java` `DateHelper.java`	Classes de suporte ao tipo definido `Date`
`ErroES.java`	Implementação Java da exceção `ErroES`
`ErroESHelper.java` `ErroESHolder.java`	Classes de suporte ao tipo `ErroES`
`NaoExiste.java`	Implementação Java da exceção `NaoExiste`
`NaoExisteHelper.java` `NaoExisteHolder.java`	Classes de suporte ao tipo `NaoExiste`
`_AnuncioStub.java`	*Stub* do cliente para o objeto remoto `Anuncio`
`Anuncio.java`	Interface Java isomórfica à interface IDL `Anuncio`
`AnuncioHolder.java` `AnuncioHelper.java`	Classes de suporte ao tipo `Anuncio`
`_QuadroAvisosStub.java`	*Stub* do cliente para o objeto remoto `QuadroAvisos`
`QuadroAvisos.java`	Interface Java isomórfica à interface IDL `QuadroAvisos`
`QuadroAvisosHolder.java` `QuadroAvisosHelper.java`	Classes de suporte ao tipo `QuadroAvisos`
`_AnuncioImplBase.java`	*Skeleton* do servidor. Implementa a interface `Anuncio`
`_QuadroAvisosImplBase.java`	*Skeleton* do servidor. Implementa `QuadroAvisos`

Implementação do servidor

A próxima etapa é implementar os objetos remotos utilizando como base os *skeletons* gerados na compilação do IDL. Neste modelo de implementação CORBA, precisamos estender a classe esqueleto. No nosso exemplo, devemos estender `_AnuncioImplBase` e `_QuadroAvisosImplBase`. A listagem abaixo (parcial) mostra a implementação deste último. Os outros arquivos-fonte da implementação CORBA podem ser encontrados em `jad/apps/bancodados/tier3/corba`.

```
package bancodados.tier3.corba;

import bancodados.*;
import bancodados.util.*;
import bancodados.util.corba.*;

public class QuadroAvisosImpl extends _QuadroAvisosImplBase {

   private BancoDados fonte;       // acesso a operacoes de cliente local

   public QuadroAvisosImpl(BancoDados fonte) {
      this.fonte = fonte;
   }

   public int length() throws ErroES {
      return fonte.length();
   }

   public synchronized void addRegistro(String anuncio, String contato)
      throws ErroES {
      fonte.addRegistro(anuncio, contato);
   }

   public boolean setRegistro(int numero, String anuncio, String contato)
      throws NaoExiste {
      try {
         return fonte.setRegistro(numero, anuncio, contato);
      } catch (RegistroInexistente ri) {
         throw new NaoExiste();
      }
   }

   public boolean removeRegistro(int numero) throws NaoExiste {
      try {
         return fonte.removeRegistro(numero);
      } catch (RegistroInexistente ri) {
```



```
            throw new NaoExiste();
         }
      }
(...)
}
```

Observe que a classe acima usa a classe `bancodados.BancoDados` para ter acesso, como cliente, à terceira camada da aplicação, ficando, desta forma, totalmente independente dos detalhes de implementação da terceira camada. O cliente (`fonte`) é passado como argumento na construção do objeto, através do servidor.

A próxima etapa é a implementação do servidor que exportará e oferecerá acesso aos objetos remotos. Ele terá que criar uma instância do objeto a ser distribuído (`QuadroAvisosImpl`) e registrá-lo no serviço de nomes. O servidor é uma classe executável que deverá permanecer no ar enquanto houver interesse de servir clientes remotos. É uma aplicação gráfica semelhante ao servidor TCP/IP do último capítulo, pois estende `bancodados.user.DadosServerFrame`. Usa as classes do pacote `org.omg.CosNaming` para exportar e registrar o objeto no sistema de nomes. Mostramos abaixo o construtor e o método `init()` que contém a inicialização e exportação dos objetos:

```
package bancodados.user;

import java.io.*;
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

(...)
public class CorbaDadosServerUI extends DadosServerFrame {

   public final static String NOMESERV = "anuncios";
   private BancoDados client;
   private QuadroAvisosImpl bdc;
   private ORB orb;

   public CorbaDadosServerUI(String[] args) {
      super(frameTitle);

      //inicializacao do ORB
      Properties props = new Properties();
      props.put("org.omg.CORBA.ORBClass", "com.sun.CORBA.iiop.ORB");
      orb = ORB.init( args, props );
```



```
    }

    protected boolean init(BancoDados client) {
       if (client == null) return false;
       try {
          // cria objeto
          bdc = new QuadroAvisosImpl(client); // 1a
          // exporta a referencia do objeto
          orb.connect(bdc);  // 1b
          // registra objeto no sistema de nomes
          org.omg.CORBA.Object objRef =
               orb.resolve_initial_references("NameService");  // 2
          NamingContext nctx = NamingContextHelper.narrow(objRef); // 3

          NameComponent ncomp = new NameComponent(NOMESERV, ""); // 4
          NameComponent[] raiz = {ncomp};
          nctx.rebind(raiz, bdc); // 5

          // imprime referencia do objeto no formato String (opcional)
          System.out.println(orb.object_to_string(bdc));
          System.out.println("Servidor no ar. Nome do servico: \'"+
                             NOMESERV +"\'");

          // espera requisicoes de objetos
          Thread waiting = new Thread(new Runner());
          waiting.start();
          return true;
       } catch(Exception e) { // (...)
       }
    }
    (...)
    public static void main(String[] args) {
       new CorbaDadosServerUI(args);
    }
}
```

Registrar um objeto remoto com CORBA exige um certo trabalho, já que os objetos são registrados no servidor de nomes dentro de um contexto hierárquico. O primeiro passo é inicializar o ORB e exportar o objeto para ele (linhas 1a e 1b acima). Depois, é preciso registrar o objeto em um sistema de nomes para que possa ser localizado por clientes remotos. É preciso encontrar um serviço chamado "`NameService`" no ORB e recuperar a referência `NamingContext` que ele retorna (linhas 2 e 3). Esta referência é necessária para termos acesso aos métodos do serviço de nomes será usado.

O servidor de nomes permite vários níveis de hierarquia. Como só temos um objeto a registrar, o criaremos na raiz (linha 4) de um componente de nome. Na linha 5, usamos o contexto de nomes para ligar este componente ao objeto.

Depois de implementado o servidor acima, o banco de dados torna-se acessível via CORBA,. Veja no capítulo 4 como executar a aplicação servidora.

Implementação do Cliente

O cliente CORBA precisa localizar o objeto no servidor através do nome que este último usou para se registrar. Não precisa saber onde está o objeto, mas precisa saber pelo menos o endereço de um servidor de nomes que possa localizá-lo. Obtendo-se uma referência ao objeto remoto, este precisa ser convertido no seu formato original. Depois disso, usá-lo é uma tarefa trivial. O acesso a métodos remotos é idêntico a invocações de métodos locais.

O cliente é mais uma implementação de `bancodados.BancoDados` e portanto pode ser usado por qualquer interface usuário. O código pode ser encontrado no diretório `bancodados/tier2/remote`. A listagem abaixo é parcial e mostra que, embora haja algum trabalho na obtenção da referência para o objeto remoto, a posterior invocação dos métodos remotos é trivial e igual à chamada de métodos locais.

```
public class CorbaClient implements BancoDados {

   public final static String NOMESERV = "anuncios";
   private QuadroAvisos bd;  // interface remota

   public CorbaClient(String[] args, Properties props) throws IOException {
      try {
         // inicializacao do ORB
         ORB orb = ORB.init(args, props);
         // registro no sistema de nomes
         org.omg.CORBA.Object objRef =
                   orb.resolve_initial_references("NameService");
         NamingContext nctx = NamingContextHelper.narrow(objRef);
         NameComponent ncomp = new NameComponent(NOMESERV, "");
         NameComponent[] raiz = {ncomp};
         org.omg.CORBA.Object obj = nctx.resolve(raiz);

         // Estreitamento do tipo
         bd = (QuadroAvisos)QuadroAvisosHelper.narrow(obj);
      } catch (Exception e) { // (...)
      }
```



```
(...)
   public void addRegistro(String texto, String autor)  {
      try {
         bd.addRegistro(texto, autor);
      } catch (ErroES e) {
         e.printStackTrace();
      }
   }
   (...)
}
```

Implantação do serviço

Para implantar o serviço, usando Java 2, é preciso executar primeiro o servidor de nomes:

```
tnameserv & (ou start tnameserv no Windows)
```

Depois, já pode-se rodar o servidor:

```
java bancodados.tier3.corba.CorbaDadosServerUI
```

Depois que a aplicação servidora estiver no ar, pode-se escolher o tipo de banco de dados que será disponibilizado aos clientes e selecionar a opção de menu "Conectar", informando o arquivo ou URL JDBC. Quando a aplicação informar que está a espera de clientes, estará pronta para receber requisições dos clientes CORBA.

Glossário

CORBA – *Common ORB Architecture*. Arquitetura Comum do ORB. Especificação que estabelece uma estrutura comum de comunicação entre aplicações independente da plataforma onde residem e da linguagem em que foram escritas.

OMA – *Object Management Architecture (OMA)*. Arquitetura de Gerência de Objetos. É uma representação de alto nível de um ambiente distribuído. Constitui-se de quatro componentes. ORBs, *Object Services* (objetos da aplicação), *Application Objects* (objetos da aplicação) e *Common Facilities* (recursos comuns).

OMG – *Object Management Group*. Grupo de Gerência de Objetos. Consórcio sem fins lucrativos criado em 1989 com o objetivo de promover a teoria e a prática da engenharia de software orientada a objetos, oferecendo uma arquitetura comum para o desenvolvimento de aplicações independente de plataformas de hardware, sistemas operacionais e linguagens de programação.

ORB – *Object Request Broker*. Corretor de Requisição de Objetos. Barramento comum que conduz as informações trocadas entre objetos.
IDL – *Interface Definition Language*. Linguagem de Definição de Interfaces. Define os tipos dos objetos, especificando suas interfaces[49]. A IDL da OMG tem uma sintaxe derivada do C++, sem as partes usadas na implementação de sistemas e acrescentando novas palavras-chave necessárias à especificação de sistemas distribuídos. Um mesmo IDL pode ser usado na geração de *stubs* e *skeletons* de linguagens diferentes, permitindo a comunicação de aplicações implementadas em linguagens distintas.
IIOP – *Internet Inter ORB Protocol*. Protocolo Internet Inter-ORB. Protocolo que estabelece pontes de comunicação entre ORBs diferentes em redes TCP/IP.
Stub. *Proxy* do cliente. Código intermediário que traduz e adapta as operações requisitadas pelo cliente ao formato de comunicação do ORB.
Skeleton. Esqueleto do servidor. Faz o trabalho inverso do *stub*. Transforma a requisição conuzida pelo ORB nos tipos nativos da implementação do servidor.
tnameserv – *Transient Name Server*. Servidor de nomes fornecido pelo JDK1.2. Deve ser executado na máquina servidora antes que qualquer servidor de objetos seja iniciado. Cada servidor de objetos deve registrar seu serviço no servidor de nomes.
idltojava – Compilador IDL fornecido pelo JDK1.2. Deve receber um arquivo IDL como entrada. Gera os *stubs*, *skeletons* para cada objeto declarado na IDL e cria pacotes e código auxiliar necessários ao desenvolvimento de clientes ou servidores de objetos CORBA em Java.

7.5. RMI

CORBA é uma solução para a comunicação entre aplicações escritas em Java ou em outras linguagens. Quando os dois lados falam Java, uma solução de desenvolvimento bem mais simples é RMI (*Remote Method Invocation*). RMI é 100% Java mas tem uma arquitetura muito parecida com CORBA. Este capítulo é uma versão do capítulo anterior, apresentando os mesmos exemplos, porém usando desta vez RMI em vez de CORBA.

Com RMI, um programa cliente poderá invocar um método em um objeto remoto da mesma maneira como faz com um método de um objeto local. Todos

os detalhes da conexão de rede são ocultados, permitindo que o modelo de objetos tenha sua interface pública mantida através da rede, sem precisar expor detalhes irrelevantes a ele, como conexões, portas ou endereços.

Há duas maneiras de usar RMI em Java. Uma é através de um protocolo chamado JRMP - *Java Remote Method Protocol*. Outra maneira é através de IIOP – *Internet Inter-ORB Protocol*. A primeira opção é adequada a aplicações 100% Java. Já a segunda, pode ser usada em ambientes heterogêneos. A forma de desenvolvimento, porém, é bastante parecida.

Um objeto remoto é obtido, usando JRMP, através de um servidor de nomes especial chamado *RMI Registry*. Ele deve estar rodando no servidor e permite que os objetos publicamente acessíveis através da rede sejam referenciados através de um nome. A classe `java.rmi.Naming` possui um método de classe `lookup()` que consulta um servidor de nomes RMI e obtém uma instância de um objeto remoto que poderá usar para invocar seus métodos.

Por exemplo, um determinado servidor de nomes RMI, situado na máquina `pleione.taurus.net` possui um registro chamado "Oceano" para um servidor de objetos remotos. Este servidor define os métodos `atira(x,y)` e `mapeia(x,y)`, numa interface `Territorio` ambos publicamente acessíveis. Para usar este objeto, pode-se fazer o seguinte:

```
Territorio mar;
mar = (Territorio)Naming.lookup("rmi://pleione.taurus.net/Oceano");
```

Com esses passos, obtemos um objeto `mar`, do tipo `Território`. O método `lookup` devolve `Object`, daí a necessidade da transformação em `Território`. Agora é possível invocar métodos remotos de `mar`:

```
tentativa[i] = mar.atira("C", 9);
```

É assim: tão simples quanto chamar um método local. O registro de nomes de objetos e sua posterior localização também é extremamente simples, já que não existe a organização em hierarquias de nomes como em CORBA.

Nos trechos de código acima, omitimos alguns detalhes, como a necessidade de tratar da exceção `RemoteException`. Este e outros detalhes sobre como usar o RMI serão ilustrados mais adiante.

Como funciona o RMI

Quando este trabalho foi iniciado, só havia uma forma de usar RMI: sobre os protocolos nativos Java. Pouco antes deste trabalho ser concluído, foi liberado aos desenvolvedores a primeira versão preliminar de RMI sobre IIOP (o mesmo protocolo usando nas comunicações CORBA). Incluímos neste capítulo algumas seções que irão destacar diferenças entre o RMI tradicional, chamado RMI sobre *Java Remote Method Protocol* (JRMP) e o novo RMI sobre IIOP.

RMI sobre JRMP

RMI/JRMP tem um funcionamento semelhante a CORBA, porém é muito mais simples e limitado à linguagem Java. É ideal para invocar objetos remotos quando tanto o cliente quanto o servidor são programas escritos em Java.

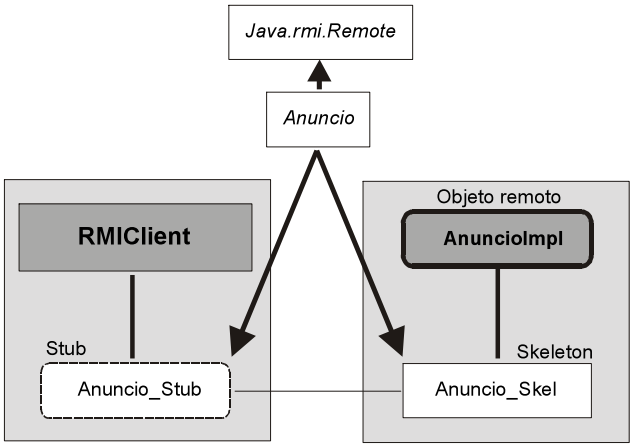

Figura 7-5

Para usar RMI, é necessário criar os objetos do servidor, definir um arquivo de interface Java implementado pela classe que define os objetos do servidor e rodar um compilador especial (análogo a um compilador IDL) que irá gerar os *stubs* e *skeletons* (figura 7-5). Um servidor de registro de nomes faz um papel semelhante ao repositório de interfaces CORBA, mantendo um registro dos objetos remotos disponíveis naquela máquina. Os clientes usam os arquivos de *stub* , em máquinas remotas, e passam a ter acesso aos objetos no servidor.

Além as classes da API, localizadas no pacote `java.rmi`, a plataforma Java 2 vêm com alguns aplicativos específicos para o RMI. O aplicativo `rmic` é o compilador gerador de *stubs* e *skeletons*. Ele deve ser rodado nas classes que implementam uma interface contendo os métodos acessíveis remotamente. Outro aplicativo é o `rmiregistry`. Ele deve estar rodando em cada máquina que tiver objetos remotos e desejar disponibilizá-los.

RMI sobre IIOP

Em dezembro de 1998 a *JavaSoft* liberou uma versão beta de RMI sobre IIOP. Esta nova arquitetura une a facilidade de desenvolvimento RMI com a portabilidade de CORBA. Objetos remotos RMI construídos de acordo com essa arquitetura são também objetos remotos CORBA. A nova ferramenta `rmic` distribuída com o pacote pode ser usada para gerar interfaces IDL a partir de implementações Java. As interfaces IDL geradas podem ser compiladas em outras linguagens.

A seção seguinte será, quase toda dedicada a RMI sobre JRMP. Porém, no final será mostrada uma maneira de realizar a conversão de aplicações RMI/JRMP para aplicações RMI/IIOP.

7.6. Construção de uma aplicação distribuída com RMI

Como nas seções anteriores, criaremos uma camada intermediária para o acesso remoto da aplicação de banco de dados. Nesta seção, usaremos RMI (sobre JRMP).

Interface Remote

O primeiro passo é escrever a interface pública que será acessível através da rede. Assim como na versão CORBA, teremos duas interfaces: `QuadroAvisos` e `Anuncio`. A interface RMI é uma interface Java e deve estender a interface `java.rmi.Remote`. Esta interface não têm novos métodos a serem implementados. É apenas uma indicação de que a nossa classe é uma interface RMI. Compare as interfaces com a versão IDL mostrada na última seção.

Classe bancodados/tier3/rmi/Anuncio.java

```
package bancodados.tier3.rmi;

import java.util.Date;
import java.rmi.Remote;
import java.rmi.RemoteException;

   public interface Anuncio extends Remote {
   public int getNumero() throws RemoteException;
   public String getTexto()  throws RemoteException;
   public Date getData()  throws RemoteException;
   public String getAutor() throws RemoteException;
}
```


Classe bancodados/tier3/rmi/QuadroAvisos.java

```
package bancodados.tier.rmi;

import java.rmi.Remote;
import java.rmi.RemoteException;
import bancodados.server.RegistroInexistente;

public interface QuadroAvisos extends Remote {

   public int length() throws RemoteException ;

   public Anuncio getRegistro(int numero)
       throws RegistroInexistente, RemoteException ;

   public void addRegistro(String texto, String autor)
       throws RemoteException;
(...)
}
```

Diferentemente de CORBA, em RMI não compilamos as interfaces. Precisamos *implementá-las* primeiro em classes que definem objetos remotos para só depois usar o gerador de *stubs* e *skeletons*.

Para que funcione como objeto remoto, uma classe deve estender a classe da API `java.rmi.server.UnicastRemoteObject`. Deve também implementar a sua interface `Remote` correspondente. Os métodos do objeto remoto todos podem provocar `java.rmi.RemoteException` e sua ocorrência deve ser declarada (ou tratada) em cada um dos métodos definidos nas subclasses de `UnicastRemoteObject`. Não somente os métodos. O construtor do objeto remoto também pode provocar uma exceção quando o objeto estiver sendo criado, portanto, mesmo que o construtor seja vazio, ele precisa ser definido mesmo que só para acrescentar a declaração "`throws java.rmi.RemoteException`".

Implementamos as interfaces acima nos arquivos `AnuncioImpl.java` e `QuadroAvisosImpl.java`. A listagem abaixo mostra a implementação parcial de `QuadroAvisosImpl`:

```
package bancodados.tier3.rmi;

import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

import bancodados.*;
```



```
public class QuadroAvisosImpl extends UnicastRemoteObject
                              implements QuadroAvisos {

   private BancoDados fonte;       // acesso a operacoes de cliente local

   public QuadroAvisosImpl(BancoDados fonte) throws RemoteException {
      this.fonte = fonte;
   }

   public int length() throws RemoteException {
      return fonte.length();
   }


   public void addRegistro(String texto, String autor) throws RemoteException {
      fonte.addRegistro(texto, autor);
   }

   public boolean setRegistro(int numero, String texto, String autor)
      throws RegistroInexistente, RemoteException  {
      return fonte.setRegistro(numero, texto, autor);
   }
(...)
```

Observe que para a implementação, não interessa a forma de armazenamento do banco de dados porque novamente, esta aplicação usa a interface `bancodados.BancoDados`..

O próximo passo é a geração de *stubs* e *skeletons*, mas antes, vamos tratar da criação do servidor que irá servir os objetos aos clientes remotos.

Implementação do servidor

Para colocar os objetos remotos "no ar" e permitir que aplicações cliente remotas tenham acesso a eles, é preciso construir um servidor. O servidor RMI é uma aplicação *middleware* que vai criar o objeto remoto (`QuadroAvisosImpl`, por exemplo), que por sua vez vai acessar, como cliente, a terceira camada da aplicação.

Implementamos o servidor como uma subclasse de `DadosServerFrame`. O método `init()` do servidor tem como principal objetivo criar uma instância do objeto remoto e registrá-lo no serviço de nomes: o *RMI Registry*. Para registrar o nome no *RMI Registry*, usamos o método de classe `Naming.rebind`. Veja o código (parcial) do servidor RMI `RMIDadosServerUI`:


```
package bancodados.user;

import java.io.*;
import java.rmi.*;

import bancodados.*;
import bancodados.tier3.rmi.*;

(...)
public class RMIDadosServerUI extends DadosServerFrame {

   /** Constante com nome do serviço */
   public final static String NOMESERV = "anuncios";

   private static String frameTitle = "Servidor RMI";  // Titulo da aplicação
   private BancoDados client;
   private QuadroAvisosImpl bdrmi;      // implementação do servidor

   public RMIDadosServerUI() {
      super(frameTitle);
   }

   protected boolean init(BancoDados client) {
      if (client == null) return false;
      try {

          // cria objeto
         bdrmi = new QuadroAvisosImpl(client);
         // exporta a referencia e registra objeto no sistema de nomes
         Naming.rebind(NOMESERV, bdrmi);
         // servidor no ar aguardando conexoes
         System.out.println("Servidor no ar. Nome do servico: \'"
                            + NOMESERV +"\'");
         return true;
      } catch(Exception e) {        // (...)
      }
   }
(...)
public static void main(String[] args) {
      new RMIDadosServerUI();
   }
}
```


Geração dos Stubs e Skeletons

Antes de gerar os *stubs* e *skeletons*, é necessário compilar as classes do servidor.

```
javac AnuncioImpl.java
javac QuadroAvisosImpl.java
```

Em seguida, executa-se o `rmic` (gerador de *stubs* e *skeletons*) sobre a classe compilada do servidor:

```
rmic –d /jad/apps/ bancodados.tier3.rmi.AnuncioImpl
rmic –d /jad/apps/ bancodados.tier3.rmi.QuadroAvisosImpl
```

Esta operação irá criar quatro novos arquivos. Dois arquivos para cada implementação. Um é o *stub* do cliente. O outro, o esqueleto do servidor. Eles devem estar no mesmo pacote (diretório) que as classes `AnuncioImpl` e `QuadroAvisosImpl`:

```
QuadroAvisosImpl_stub.class
QuadroAvisosImpl_skel.class

AnuncioImpl_stub.class
AnuncioImpl_skel.class
```

Desenvolvendo o cliente

O cliente é muito simples. Sua principal função é localizar o objeto. Depois disso, é só usar a referência. Uma instância do objeto remoto é obtida usando o método `lookup()`, que consulta o *RMI Registry* do servidor escolhido e retorna um objeto de acordo com a URL definida. O cliente é mais uma implementação da interface `bancodados.BancoDados` (pode ser usado por qualquer interface do usuário). A listagem abaixo é parcial.

```
package bancodados.tier2.remote;

import java.io.IOException;
import java.util.Date;
import java.rmi.*;

import bancodados.*;
import bancodados.tier3.rmi.*;

public class RMIClient implements BancoDados {

   public final static String NOMESERV = "anuncios";

   private QuadroAvisos bd;
```



```
   public RMIClient(String hostname) throws IOException {

      try {
        Object obj = Naming.lookup("rmi://" + hostname + "/" + NOMESERV);
        bd = (QuadroAvisos) obj;

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

   }

   public void addRegistro(String texto, String autor)  {
      try {
         bd.addRegistro(texto, autor);
      } catch (RemoteException e) { // (...)
      }
   }

(...)
```

Inicialização do serviço de nomes

Para implantar o serviço, deve-se iniciar o *RMI Registry* a máquina que será servidora. Este é o verdadeiro servidor de objetos que fica escutando a porta 1099 (*default*) onde clientes podem conectar e tentar descobrir quais objetos remotos estão disponíveis.

```
rmiregistry &       (Unix)
start rmiregistry  (WinNT/95/98)
```

Estando o *RMI Registry* no ar, pode-se executar o servidor

```
java bancodados.tier3.rmi.RMIDadosServerUI
```

Para iniciar o servidor, escolha uma fonte de dados (arquivo ou JDBC) e depois selecione a opção de menu "Conectar". Quando o servidor informar que o objeto foi exportado, estará pronto para receber requisições de clientes.

Conversão de uma aplicação RMI/JRMP em RMI/IIOP

São poucas as mudanças necessárias para transformar uma aplicação JRMP (que só usa objetos remotos escritos em Java) em aplicação IIOP (que pode usar qualquer objeto remoto CORBA) são realizadas principalmente nas classes que implementam clientes e servidores. Os objetos remotos precisam mudar a classe de quem herdam. Não é preciso mexer nas interfaces `Remote`.

Os novos objetos remotos devem estender **javax.**rmi.**PortableRemoteObject** e não mais **java.**rmi.server.**UnicastRemoteObject**:

```
public class QuadroAvisosImpl extends PortableRemoteObject
                               implements QuadroAvisos { ... }

public class AnuncioImpl extends PortableRemoteObject
                          implements Anuncio { ... }
```

Os servidores não devem usar o serviço de nomes do *RMIRegistry* (que usa JRMP). Deve usar o serviço de nomes nativo do Java (pacote `java.naming`) para registrar objetos remotos RMI em um ORB:

```
import javax.rmi.*;     // RMI sobre IIOP
import javax.naming.*;  // JNDI (Naming services)

public class RMIIIOPDadosServerUI extends DadosServerFrame {

(...)
   private QuadroAvisosImpl bdrmi;      // implementação do servidor
   private Context namingContext;       // ambiente RMI-IIOP

   public RMIIIOPDadosServerUI() {
      super(frameTitle);
      try {
         Hashtable env = new Hashtable();
         env.put("java.naming.factory.initial",
                 "com.sun.jndi.cosnaming.CNCtxFactory");
         env.put("java.naming.provider.url",
                 "iiop://localhost:900");   // nameserver ñ precisa ser local!

         // obtem contexto de serviço de nomes
         namingContext = new InitialContext(env);
      } catch (NamingException e) {
         e.printStackTrace();
      }
   }

   protected boolean init(BancoDados client) {
      try {
         bdrmi = new QuadroAvisosImpl(client);
         // exporta a referencia e registra objeto no sistema de nomes
         namingContext.rebind(NOMESERV, bdrmi);
         System.out.println("Servidor no ar. Nome do servico: \'"
                            + NOMESERV +"\'");
         return true;
      } catch(Exception e) { // (...)
      }
```



```
    }
```

O cliente deve fazer o mesmo. Obter um contexto, localizar o objeto, obter a referência.

```
import javax.rmi.*;     // PortableRemoteObject
import javax.naming.*;  // pacote JNDI (nomes)

public class RMIIIOPClient implements BancoDados {
   public final static String NOMESERV = "anuncios";
   private QuadroAvisos bd;

   public RMIIIOPClient(Hashtable env) throws IOException {
       try {
         // Obtenção do contexto inicial (JNDI) usando por RMI sobre IIOP
         Context namingContext = new InitialContext(env);
         // obtem objeto via sistema de nomes (usando JNDI)
         Object obj = namingContext.lookup(NOMESERV);
         // Estreita (converte) o tipo
         bd = (QuadroAvisos)PortableRemoteObject.narrow(obj,
                                                QuadroAvisos.class);

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

   public void addRegistro(String texto, String autor)  {
      try {
         bd.addRegistro(texto, autor);
      } catch (RemoteException e) {
         e.printStackTrace();
      }
   }
```

Para gerar os stubs e skeletons para o RMI/IIOP, é preciso rodar o novo `rmic` com a opção `-iiop`:

```
rmic -iiop -d /jad/apps/ bancodados.tier3.riiop.AnuncioImpl
rmic -iiop -d /jad/apps/ bancodados.tier3.riiop.QuadroAvisosImpl
```

Isto deve gerar quatro arquivos dentro do pacote `bancodados.tier3.riiop`:

`QuadroAvisos_Stub.class`	- *Stub*
`QuadroAvisosImpl_Tie.class`	- *Skeleton*
`Anuncio_Stub.class`	- *Stub*
`AnuncioImpl_Tie.class`	- *Skeleton*

Para iniciar o servidor, é preciso que o serviço de nomes do JNDI esteja no ar:

`tnameserv & (ou start tnameserv).`

Depois, pode-se executar o servidor como foi feito com as outras tecnologias.

Resumo

Os passos a seguir resumem o processo de desenvolvimento e implantação de um servidor de objetos usando RMI/JRMP:

- 1. Definir a interface: declare todos os métodos que serão acessíveis remotamente em um arquivo de interface Java que estende `java.rmi.Remote`. Todos os métodos devem declarar `throws java.rmi.RemoteException`. Isto deve ser feito para cada objeto que será acessível através da rede.
- 2. Escrever a implementação dos objetos remotos - classes que estendem `java.rmi.server.UnicastRemoteObject` e que implementam a interface criada no passo 1. Todos os métodos devem declarar `throws java.rmi.RemoteException` inclusive o construtor, mesmo que seja vazio.
- 3. Estabelecer um gerente de segurança (`SecurityManager`[1]), obter uma instância do objeto a ser servido e implementar o mecanismo de registro do objeto do *RMIRegistry* no servidor.
- 4. Compilar o código-fonte dos objetos remotos com `javac`, gerando um arquivo de classe do servidor.
- 5. Compilar o arquivo de classe dos objetos remotos com `rmic`, gerando um arquivo stub, e um arquivo skeleton. (`ObjImpl_stub.class` e `ObjImpl_skel.class`)
- 6. Escrever, compilar e instalar o(s) cliente(s)
- 7. Instalar o stub no(s) cliente(s) (copiar o stub gerado no passo 5 para as máquinas e CLASSPATH acessíveis pelos clientes ou prover mecanismo que permita ao cliente fazer o download do stub.)
- 8. Iniciar o *RMI Registry* no servidor
- 9. Iniciar o servidor de objetos
- 10. Iniciar os clientes informando o endereço do servidor.

---

[1] No nosso exemplo não definimos uma instância de `RMISecurityManager`. Sem um `SecurityManager` não é permitido a um cliente carregar stubs e outras classes dinamicamente do servidor. Para instalar um `RMISecurityManager`, faça: `System.setSecurityManager(new RMISecurityManager());`

Glossário

RMI - *Remote Method Invocation*. Invocação Remota de Método.
Stub. *Proxy* do cliente. Código intermediário que traduz e adapta as operações requisitadas pelo cliente em formato de dados adequado à transmissão via rede.
Skeleton. Esqueleto do servidor. Faz o trabalho inverso do *stub*. Repassa a requisição para o servidor.
rmic – *RMI Compiler*. Compilador (gerador) de *stubs* e *skeletons* fornecido pelo JDK1.2. Deve receber o objeto remoto (que implementa uma subinterface de `java.rmi.Remote`) como entrada (código compilado). Gera os *stubs*, *skeletons* para cada objeto.
rmiregistry – Servidor de nomes RMI fornecido pelo JDK1.2. Deve ser executado na máquina servidora antes que qualquer servidor de objetos seja iniciado. Cada servidor de objetos deve registrar seu serviço no servidor de nomes. Depende da existência de um servidor ou serviço de nomes (DNS ou arquivo de *hosts*) TCP/IP.

7.7. Comparação entre as tecnologias

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

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 .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.

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 transparente 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.

7.8. Resumo

Para organizar todas as informações apresentadas, construímos uma tabela comparativa, comparando diversos aspectos das tecnologias analisadas e mostrando também onde (em que pacote nativo ou extensão) cada tecnologia é fornecida. A tabela 7-5 compara tecnologias de objetos distribuídos. Essas tecnologias podem ser usadas na construção de camadas (*tiers*) adicionais entre o cliente e o servidor.

Tabela 7-5 – 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[2]	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[3] 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)	-	-

Utilizar uma solução baseada no pacote `java.net` traz algumas vantagens: maior compatibilidade (por ser nativa ao núcleo da plataforma Java), flexibilidade

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

[3] 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.

(permite conectividade com agentes escritos em outras linguagens) e velocidade. A complexidade da programação em baixo nível e a baixa reutilização de código são motivos que nos levam a procurar soluções de objetos distribuídos. Em estudos realizados com as três tecnologias de objetos distribuídos, comparamos três tecnologias diferentes: RMI sobre JRMP (nativa desde o JDK1.1), CORBA sobre IIOP (nativa desde o JDK1.2[4]) e RMI sobre IIOP (extensão ao JDK1.2 ainda em beta). Nos casos e ambientes estudados, 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.

[4] Plataforma Java 2

