UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PROTÓTIPO DE SISTEMA PARA ACESSO A BANCO DE DADOS DISTRIBUÍDOS E HETEROGÊNEOS EM REDES TCP/IP

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

ADEMIR JOSÉ FINK

BLUMENAU, JUNHO/2000.

2000/1-1

PROTÓTIPO DE SISTEMA PARA ACESSO A BANCO DE DADOS DISTRIBUÍDOS E HETEROGÊNEOS EM REDES TCP/IP

ADEMIR JOSÉ FINK

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Marcel Hugo — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Marcel Hugo

_____

Prof. Everaldo Artur Grahl

_____

Prof. Sérgio Stringari

Dedico este trabalho aos meus pais e meus irmãos por terem depositado confiança na minha capacidade e pela compreensão e apoio em todos os momentos de minha trajetória.

AGRADECIMENTOS

Agradeço a todos os professores do curso de Bacharelado em Ciências da Computação da Universidade Regional de Blumenau, que de alguma forma contribuíram para o meu aperfeiçoamento acadêmico.

Ao professor Marcel Hugo, pelo incentivo, orientação e atenção dispensada durante todo o desenvolvimento do trabalho.

Agradeço a meus avós e toda minha família pelos momentos de felicidade que eles me proporcionam, que foi um incentivo muito grande para eu ter chegado até aqui.

Agradeço aos meus amigos Denis Alberto Dalmolin, Jones Cássio Poffo, Marcelo Dalpiaz e Maurício Dalpiaz pela amizade, companheirismo e compreensão, principalmente nos momentos que mais foram necessários.

SUMÁRIO

LISTA DE ABREVIATURAS

ANSI – *American National Standard Institute* (Instituto de Padrão Nacional Americano)

API – *Application Programming Interface* (Interface de Programação de Aplicação)

BD – Banco de Dados

BDD – Banco de Dados Distribuído

DBA – *Data Base Administrator* (Administrador de Banco de Dados)

JDBC – *Java Database Component* (Componente Java para Base de Dados)

JVM – *Java Virtual Machine* (Máquina Virtual Java)

SGBD – Sistema Gerenciador de Banco de Dados

SQL – *Strutured Query Language* (Linguagem de Consulta Estruturada)

ODBC – *Open Database Connectivity* (Conectividade com Base de Dados Aberta)

TCP/IP – *Transmission Control Protocol / Internet Protocol* (Protocolo de Controle de Transmissão / Protocolo de Internet)

UML – *Unified Modeling Language* (Linguagem Unificada de Modelagem)

URL – *Uniform Resource Locator* (Localizador de Recurso Uniforme)

LISTA DE FIGURAS

RESUMO

Este trabalho enfoca bancos de dados distribuídos e heterogêneos, Java e a API JDBC própria do Java. Apresenta as características, vantagens e desvantagens de um banco de dados distribuído e heterogêneo, bem como as facilidades na utilização da linguagem Java e de sua API JDBC para a implementação de um protótipo de sistema que tem a finalidade de tornar transparente para o usuário o acesso a bancos de dados distribuídos e heterogêneos em redes TCP/IP.

ABSTRACT

This study addresses the use heterogeneous and distributed databases, Java and API JDBC own of the Java. Presents the characteristics, advantages and disadvantages of a heterogeneous and distributed databases, as well as the utilization of the language Java and of your API JDBC for the implementation of a prototype of system that has the purpose of turning transparent for the user the access to heterogeneous and distributed databases in nets TCP/IP.

1 INTRODUÇÃO

Em muitas instituições, é comum encontrar as informações dispersas em vários setores e armazenadas em vários locais distintos, isto é, os dados estão distribuídos, permitindo que cada setor mesmo que distante geograficamente mantenha controle de seus próprios dados e ofereça um compartilhamento global no uso destes por outros setores da organização.

Um dos pontos fortes das redes corporativas é o compartilhamento de dados, o que traz grande performance no deslocamento de informações dentro da corporação. Embora exista este recurso de redes corporativas na grande parte das corporações, cada empresa possui a sua própria base de dados independente ([KOR1995]). A falta de integração entre as bases de dados reflete diretamente no setor administrativo, inviabilizando uma visão concentrada de todas as informações da empresa. Este é um dos grandes motivos porque a tecnologia de banco de dados distribuídos está crescendo dia-a-dia, a necessidade de se ter o maior número de informações possíveis para que se tome uma decisão rápida e precisa.

Neste contexto, em muitos casos o usuário não possui uma interface única para acesso aos dados que estão distribuídos nos vários setores de uma empresa, mas ao contrário, tem que utilizar várias aplicações distintas para fazê-lo. Com um ambiente de múltiplos bancos de dados independentes e heterogêneos, as aplicações que foram escritas para utilizar a base de dados não estarão integradas entre as filiais da corporação, ou seja, cada aplicação será executada em uma das filiais da corporação independente das informações existentes nas outras bases de dados ([HAC1993]).

Conforme Couceiro ([COU1991]), sem a existência de um ambiente distribuído as consultas aos dados que estão nas outras bases de dados tornam-se extremamente ineficazes, porque necessitará um contato prévio com as outras partes da corporação para discutir a maneira de transportar o dado de uma empresa para outra. Esta forma de acesso aos dados é lenta e em alguns casos não se obtém o resultado esperado.

Existem diversas propostas para o problema de se desenvolver aplicações que necessitem acessar dados residentes em diferentes Sistemas Gerenciadores de Bancos de dados (SGBD). Uma proposta geral e viável para a integração destes diferentes SGBDs é a

utilização de uma camada de software intermediária entre as aplicações cliente e os vários SGBDs.

O principal objetivo da camada intermediária, que também pode ser chamada de *middleware*, é manter a transparência do tipo de servidor para o cliente. Em outras palavras, os clientes conectam-se ao *middleware* e submetem-lhe requisições. O *middleware* interpreta essas requisições e endereça ao servidor adequado. Assim, o cliente não interage diretamente com o servidor, ficando transparente tanto o tipo de SGBD e sua localização, quanto sua distribuição.

A proposta deste trabalho é desenvolver um sistema de acesso a SGBDs Distribuídos e Heterogêneos, que consiste na implementação da camada de software intermediária entre Cliente/Interface e SGBDs.

Para implementação do protótipo foi utilizado o ambiente JBuilder 2.0, bem como os SGBDs Oracle 8i, Interbase 5.5 e Access 7.0. Foi utilizado o protocolo de comunicação de redes TCP/IP (*Transmission Control Protocol* / *Internet Protocol*). O protótipo foi especificado utilizando a metodologia UML (*Unified Modeling Language*), enquanto que o modelo de dados necessário para a implementação do protótipo foi especificado para um SGBD relacional segundo a técnica de Entidade e Relacionamento.

1.1 OBJETIVOS

Este trabalho tem por objetivo principal a especificação e implementação de um protótipo de sistema para acesso a Banco de Dados Distribuídos e Heterogêneos em Redes TCP/IP, que torne transparente ao usuário a utilização do SGBD.

1.2 ORGANIZAÇÃO

O capítulo 1 apresenta uma rápida introdução sobre o trabalho.

O capítulo 2 apresenta banco de dados, seus conceitos e características.

O capítulo 3 descreve a linguagem de programação Java com conceitos e características.

O capítulo 4 apresenta JDBC, seus conceitos, funcionalidade e principais comandos.

O capítulo 5 descreve as especificações de implementação do protótipo.

O capítulo 6 apresenta as conclusões e sugestões sobre o protótipo desenvolvido.

No capítulo 7 encontram-se os anexos do trabalho.

2 BANCOS DE DADOS DISTRIBUÍDOS

Banco de dados, segundo Cerícola ([CER1995]), é o arquivo físico, em dispositivos periféricos, onde estão armazenados os dados de diversos sistemas, para consulta e atualização pelo usuário.

Um Sistema de Banco de Dados é um sistema cujo objetivo global é manter as informações e torná-las disponíveis quando solicitadas, e um Sistema Gerenciador de Banco de Dados (SGBD) é o software que manipula todos os acessos ao Banco de Dados (BD) ([DAT1991]).

Dado é o valor do campo fisicamente registrado no Banco de Dados e informação refere-se ao significado destes valores para determinado usuário ([DAT1991]).

Segundo Date ([DAT1991]), “um Sistema Distribuído é qualquer sistema que envolve múltiplas localidades conectadas juntas em uma espécie de redes de comunicações, nas quais o usuário de qualquer localidade pode acessar os dados armazenados em outro local”.

Cada localidade pode ser considerada como um SGDB em si: tem seu próprio banco e seu próprio DBA (*Data Base Administrator* ou Administrador do Banco de Dados), seus próprios terminais e usuários, o seu próprio armazenamento local e o seu próprio computador.

Couceiro ([COU1991]) diz que um Sistema de Banco de Dados Distribuídos existe quando um Banco de Dados integrado logicamente (integração lógica significa que qualquer nó tem acesso potencial a todo o banco de dados) é fisicamente distribuído sobre diferentes nós de computação interligados por uma rede. Idealmente, a distribuição física deve ser transparente aos programas de aplicação. Define-se um nó de computação como um computador localizado numa área de organização com certas facilidades de processamento.

Se todos os SGBDs locais oferecem interfaces idênticas ou pelo menos da mesma família, então se diz que o sistema é **homogêneo**; em caso contrário é **heterogêneo** ([COU1991]).

Os sistemas heterogêneos surgem usualmente quando há necessidade de integrar sistemas já existentes. A escolha entre uma tecnologia de hardware e software é influenciada pelo aproveitamento das tecnologias já existentes e pelo próprio hábito e grau de cooperação

esperado dos usuários em caso de uma mudança. Surge então a alternativa de adotar uma arquitetura híbrida ([CAS1985]).

Hoje é comum em várias empresas ter grande diversidade de ambientes operacionais e SGBDs. Dentro desse contexto, principalmente quando se trata de Banco de Dados distribuídos, fica difícil recuperar as informações distribuídas em Bancos de Dados de diferentes fornecedores ([HAC1993]).

Como principais vantagens dos Bancos de Dados Distribuídos (BDDs), Cerícola ([CER1995]) apresenta entre outros, as seguintes:

a) tempo de resposta mais rápido, em função da redução do tráfego no acesso a base de dados – as informações ficam mais próximas do local onde serão utilizadas normalmente;

b) compartilhamento através de interfaces padrões entre partições do banco de dados;

c) melhoria da confiabilidade, pois nem todos os SGBDs locais deverão estar funcionando para que o sistema como um todo esteja disponível;

d) autonomia local, a distribuição do sistema permite aos grupos individuais exercerem um controle local sobre os seus próprios dados, com contabilidade local e, de maneira mais geral, que se tornem menos dependentes de um centro de processamento de dados.

Os principais problemas dos BDDs conforme Casanova ([CAS1985]) são:

a) a heterogeneidade global;

b) introdução de padrões globais sem que seja comprometida a autonomia local;

c) critérios de alocação de custos tendo em vista acessos locais e remotos;

d) complexidade requerida para assegurar a adequada coordenação entre os nós.

2.1 SQL

Structured Query Language (SQL) é uma linguagem de controle e acesso a dados padrão da indústria. É o meio mais usado pelas linguagens de desenvolvimento para acesso à base de dados. Por ser uma linguagem estruturada e ser de fácil entendimento, seu uso está cada vez mais comum entre os sistemas de informações das empresas ([HUR1990]).

A linguagem é controlada pelo ANSI (*American National Standard Institute*) que tem a finalidade de padronizar o SQL de tal forma que uma aplicação escrita em SQL ANSI seja capaz de acessar qualquer banco de dados padrão SQL do mercado. Mas as empresas que desenvolvem sistemas de banco de dados implementam extensões à linguagem para ter mais poderes que às outras. Por isso, os desenvolvedores tem que estar atentos a que comandos usar para que seus sistemas fiquem compatíveis a todos os bancos de dados que utilizam SQL ([HUR1990]).

A linguagem SQL, segundo Date ([DAT1989]), é composta por um grupo de facilidades para definição, manipulação e controle de dados em um banco de dados relacional. Entre as principais facilidades da linguagem, pode ser citada a sua fácil compreensão, por ser uma linguagem de mais alto nível e principalmente sua padronização, que por si só justifica seu uso no protótipo.

2.2 CLIENTE/SERVIDOR

Uma arquitetura cliente/servidor é uma abordagem da computação que separa os processo em plataformas independentes que interagem, permitindo que os recursos sejam compartilhados enquanto se obtém o máximo de benefício de cada dispositivo diferente. É basicamente uma forma de computação distribuída ou em rede ([BOC1995]).

O sistema cliente/servidor é um paradigma lógico para o processamento distribuído, sendo que o cliente e o servidor podem estar ou não em uma mesma máquina física. Quando os processos cliente e servidor são executados em máquinas diferentes, estes são conectados através de redes locais ou remotas, sendo que as redes locais são as implementações mais comuns de cliente/servidor. O usuário do sistema interage com um cliente, que por sua vez emite pedidos e recebe resultados do servidor ([REN1994], [VAS1995]).

Segundo ([BOC1995]), as principais características de um sistema cliente/servidor são:

a) uma arquitetura cliente/servidor consiste em um processo cliente e um processo servidor, que podem ser distinguidos um do outro, embora possam interagir totalmente;

b) a parte cliente e as partes servidor podem operar em diferentes plataformas de computador;

c) tanto a plataforma do cliente quanto a plataforma do servidor podem ser atualizadas sem que se tenha que atualizar a outra plataforma;

d) o servidor pode atender vários clientes simultaneamente, e os clientes podem acessar vários servidores;

e) os sistemas cliente\servidor incluem algum tipo de capacidade de operar em rede.

Ao utilizar um SGBD de duas camadas, que compreenderia o modelo Cliente/Servidor, o usuário teria que acessar os dados interagindo diretamente com o SGBD disponível no servidor ([HAC1993]), conforme a figura 1.

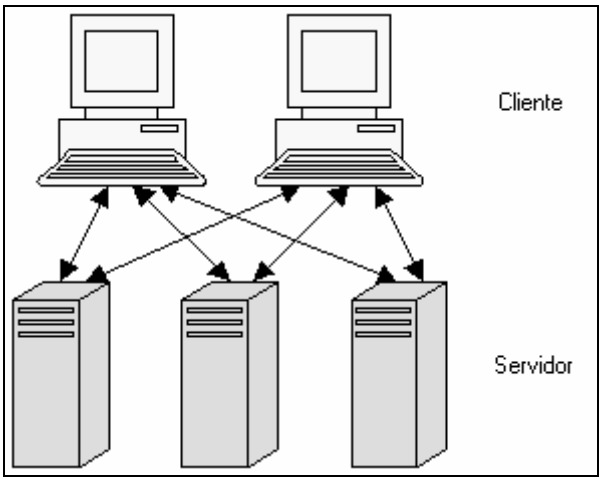

Figura 1 - Modelo de duas camadas

Neste protótipo de sistema, será utilizado o modelo de três camadas, que permite que o cliente faça conexão com o *middleware*, que é um software que conecta dois módulos permitindo que estes dois módulos se comuniquem entre si ([BOC1995]), como se fosse com um SGBD, e este, por sua vez, verifica a quais SGBD o usuário tem acesso, e através de uma distribuição pré-definida acessa o computador pertinente ao comando enviado pelo cliente ([HAC1993]). O próprio *middleware* recebe a resposta do SGBD e devolve para o cliente o resultado da consulta. Dessa forma, o cliente tem a impressão de que faz acesso com um único SGBD, independente da multiplicidade de SGBD existentes nos servidores, conforme é mostrado na figura 2.

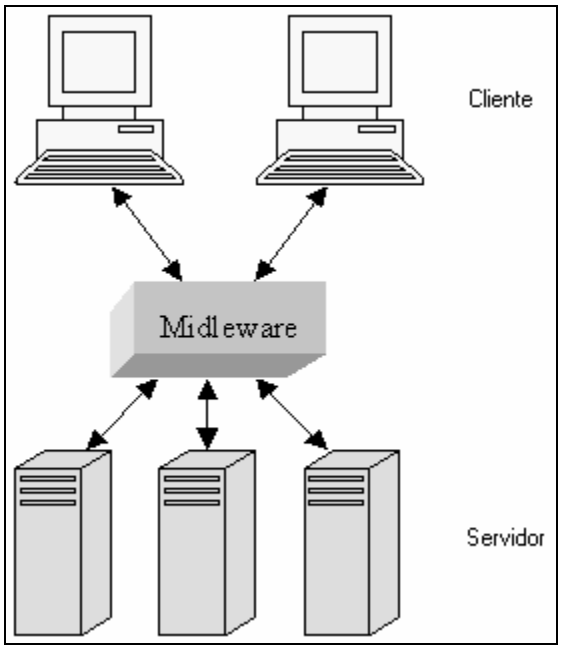

Figura 2 - Modelo de três camadas

Segundo Sybase ([SYB1999]), o sistema de três camadas apresenta várias vantagens, entre elas:

a) poupa os usuários da complexidade de protocolos e acessos a BD com comandos específicos do SGBD;

b) simplifica a integração de fontes de dados heterogêneas e permite construir sistemas mais facilmente;

c) permite ter o *middleware* num único servidor, pois os clientes se conectam a ele para acessar os SGBDs dos demais servidores.

2.3 TRANSAÇÕES

Segundo Casanova ([CAS1985]), a finalidade do sistema de banco de dados distribuído é realizar transações. A transação é uma unidade de trabalho. Ela consiste na execução de uma seqüência de operações especificadas pela aplicação, começando com uma operação especial *BEGIN TRANSACTION*, e terminando ou com uma operação *COMMIT* ou com uma operação *ROLLBACK*. *COMMIT* é utilizado para sinalizar o término bem-sucedido (a unidade de trabalho foi completada com sucesso); *ROLLBACK* é usado para sinalizar o término mal-sucedido (a unidade de trabalho não pode ser completada com sucesso por haver ocorrido alguma situação excepcional – por exemplo, um registro necessário não pode ser localizado). Término aqui, se refere ao término da transação, e não necessariamente ao término do programa; uma execução de programa poderá corresponder a uma seqüência de diversas transações, uma após a outra.

Para controlar as transações, instrumentos de SGBD distribuídos clássicos baseiam-se em um mecanismo chamado *two-phase commit*. Uma conclusão *two-phase commit* deve garantir que todos os participantes façam da mesma forma uma determinada transação. Ou todos a aceitam ou todos a rejeitam ([BOC1995]). Esse mecanismo é dividido em duas fases ([DAT1988] [BOC1995]):

a) Fase 1 – um processo coordenador envia o comando para todos os SGBDs participantes, onde cada um determina se pode concluir a transação. Caso possa concluir com sucesso então o SGBD retornará verdadeiro, caso contrário retornará falso. Após o envio da mensagem o SGBD permanece em um estado em que é possível tanto concluir a transação (*COMMIT*) quanto desfazê-la (*ROLLBACK*);

b) Fase 2 – o processo coordenador recebe a mensagem de cada um dos SGBDs participantes. Mesmo que apenas um SGBD tenha retornado falso, o coordenador envia uma mensagem cancelando a transação para todos os SGBDs participantes, mas se todos retornarem verdadeiro o coordenador transmite a mensagem concluir.

3 JAVA

Java é uma linguagem de programação com propósito geral (seu uso pode ser tanto científico quanto comercial), *multi-thread* (capaz de realizar várias tarefas ao mesmo tempo), orientada a objetos e possui a API JDBC para fazer acesso às bases de dados. Por conta da máquina virtual Java é também uma plataforma de software independente de sistema operacional e do hardware subjacente ([CAM1996]).

Principalmente por estas características, que serão melhor apresentadas no decorrer deste capítulo, a linguagem de programação Java aparece como sendo a mais indicada para a implementação do protótipo.

3.1 MÁQUINA VIRTUAL JAVA

É uma máquina imaginária que é implementada pela emulação (em software) de uma máquina real. O código para a *Java Virtual Machine* (JVM) é armazenado em arquivos **.class** ([RAM2000]).

Os arquivos **.class** são gerados através da compilação e linkedição, semelhante a um programa na linguagem C. O código binário gerado na compilação (também chamado de ***byte-codes***) é comum a todos os sistemas operacionais ([RAM2000]).

Um programa Java é multiplataforma porque um mesmo binário Java pode ser executado nas diferentes plataformas que implementam JVM, conforme mostrado na figura 3 ([RAM2000]).

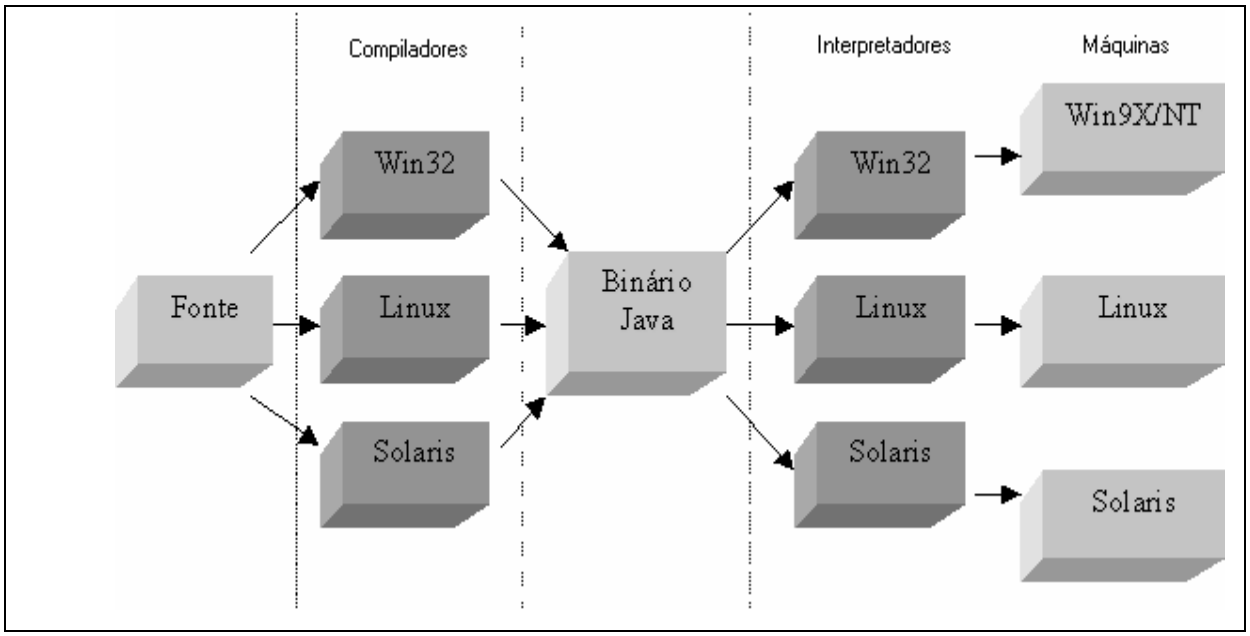

Figura 3 - Funcionamento da JVM

3.2 CARACTERÍSTICAS DA LINGUAGEM

Java é fortemente tipada. Esta especificação torna claras as diferenças entre os erros detectados pelo compilador em tempo de compilação e aqueles que ocorrem em tempo de execução. A compilação consiste em traduzir os programas Java para uma representação de máquina independente *byte-code*. As atividades em tempo de execução incluem a carga e a conexão das classes necessárias para executar um programa, código de máquina genérico, otimização do programa dinamicamente e a real execução do programa ([GOS1996]).

A maioria das linguagens orientadas a objetos requer a destruição dos objetos criados quando eles não forem mais utilizados. Java permite que você crie quantos objetos puder, mas não é exigida a destruição dos mesmos. O ambiente Java deleta os objetos assim que percebe que não serão mais utilizados. Este processo é conhecido como *Garbage Collector* ([CAM1996]).

3.3 ORIENTADA A OBJETOS

Com exceção das operações mais primitivas (*for*, *while*, *if*) e os tipos de dados *int*, *byte*, *char* e *boolean*, tudo em Java é um objeto. Objetos são instâncias de classes Java. Java

possui várias classes que podem ser usadas em programas e possibilita a criação de classes próprias ([CAM1996]).

Os objetos são criados a partir do momento em que as classes são instanciadas. A instância de uma classe acontece no momento em que o operador ***new*** é declarado. Nesse instante é alocada memória para um novo objeto daquela classe ou tipo. O ***new*** requer um argumento: a chamada do método construtor ([CAM1996]).

Em Java não existem funções desvinculadas de classes, funções isoladas. Isto implica que todo trecho de código que for escrito deve pertencer a uma classe, mais precisamente deve ser um método desta classe. O programa mais simples em Java deve conter pelo menos uma classe e um método de início do programa ([LEM1996]).

3.4 CLASSES

Uma classe é um protótipo que pode ser usada para criação de vários objetos. A implementação de uma classe é composta de duas partes: a declaração da classe e o corpo da classe ([CAM1996]).

A declaração de uma classe define o nome da classe e outros atributos, como a superclasse da classe. No mínimo a declaração deve conter a palavra reservada c*lass* e o nome da classe ([CAM1996]). Exemplo:

```
class <nome da classe>
{
        corpo da classe
}
```

Quando não declarada a superclasse de uma classe, será assumida como sendo a Object. Mas para especificar a superclasse usa-se o nome da classe, a palavra reservada *extends* seguida do nome da superclasse ([CAM1996]). Exemplo:

```
class <nome da classe> extends <nome da superclasse>
{
        corpo da classe
}
```

O corpo de uma classe é dividido em duas partes ([CAM1996], [NAU1996]):

a) declaração de variáveis membro, que representa os estados da classe (propriedades);

b) declaração dos métodos que implementam os comportamentos da classe.

As variáveis membro podem ser declaradas como qualquer um dos tipos de dados simples. Exemplo:

```
class <nome da classe>
{
        int <nome da variável>;

}
```

Os métodos de uma classe são declarados da seguinte forma:

```
class <nome da classe>
{
        tipo-do-método <nome do método> (lista de parâmetros formais)
        {
                corpo do método
        }

}
```

Onde tipo do método é qualquer tipo de retorno que este método queira retornar. No caso de não haver retorno então o tipo é *void*. A lista de parâmetros formais é uma seqüencia de pares de tipo e identificador separados por vírgulas. No caso em que não são usados parâmetros, a declaração de método deve incluir um par de parênteses vazios ([NAU1996]).

Para criar uma instância de uma classe, é utilizado o operador *new*. Exemplo:

```
<minha classe> <variável> = new <minha classe>;
```

3.5 TRATAMENTO DE EXCEÇÕES

O tratamento de exceções permite lidar com as condições anormais de funcionamento de um programa. Fazendo uso deste recurso o *software* passará a ser mais robusto, seguro e bem estruturado. Java é uma linguagem que faz forte uso de exceções. Em algumas linguagens que implementam tratamento é possível programar sem usar esse recurso, mas em Java não ([LEM1996]).

Um tratamento de exceção é feito a partir do bloco de código *try{} catch{}* no programa. Exemplo:

```
try
{
 <código do programa>

}

catch (<tipo da exceção>)

{

<tratamento dado ao erro>

}
```

O comando *try* é usado para tentar executar um determinado bloco de código, e caso acontecer algum erro a exceção é capturada pelo comando *catch*, que contém o bloco de código manipulador da exceção ([NAU1996])

3.6 THREADS

Threads são linhas de execução que rodam dentro de um processo. Normalmente as *threads* compartilham regiões de memória, mas não necessariamente. *Threads* permitem que sua aplicação execute mais de um método ao mesmo tempo ([LEM1996]). Cada JVM pode suportar muitas *threads* executando em conjunto, caracterizando um sistema *multi-thread*. Essas *threads* independentes executam código Java que operam em valores Java e objetos residentes em uma parte da memória principal ([GOS1996]).

Assim como acontece com todos os conceitos de Java, as *threads* são representadas em uma classe. A classe *Thread* encapsula todo o controle que você vai precisar ter sobre as linhas de execução ([NAU1996]).

Pode-se criar uma *thread* como extensão da classe *Thread*, sobrescrevendo o método *run()*, como no exemplo:

```
class <nome da nova thread> extends Thread
{
        public void run()
        {
                <corpo de execução da thread>
        }
```

}

O método *run()* é o corpo de uma *thread*. Ele é chamado depois que a *thread* foi inicializada. Sempre que o método *run()* retornar, a *thread* vai parar.

3.7 SOCKETS TCP/IP

O *Transmission Control Protocol / Internet Protocol* (TCP/IP) é um protocolo de comunicação de redes. Ele foi projetado pelo Departamento de Defesa norte-americano (DoD) para a ARPANET, uma rede construída para conectar vários locais do Órgão de Projetos de Pesquisa Avançada do DoD. No entanto, o TPC/IP é um protocolo público, e não proprietário. Devido a uma ordem do DoD para que o TCP/IP fosse utilizado para todos os contratos militares e civis do governo norte-americano, o TCP/IP passou a ser amplamente usado dentro de órgãos governamentais. No entanto, ele ganhou popularidade em universidades, institutos científicos e na Internet ([BOC1995]).

Abaixo estão relacionadas algumas características técnicas que fizeram com que o TCP/IP fosse tão amplamente difundido ([TAN1994], [GAS1993]):

a) grande portabilidade, pois vários tipos de plataformas podem ser interligados, sem preocupação com fabricante;

b) independência de hardware de rede específico o que possibilita a integração de diferentes padrões;

c) facilidade no desenvolvimento de aplicações e serviços.

Um *socket* pode ser considerado um ponto de referência para o qual as mensagens podem ser enviadas e a partir do qual as mensagens são recebidas. Qualquer processo pode criar um *socket* para se comunicar com outro processo, mas os dois processos devem criar seus próprios *sockets*, visto que os dois *sockets* são usados como um par ([BOC1995]).

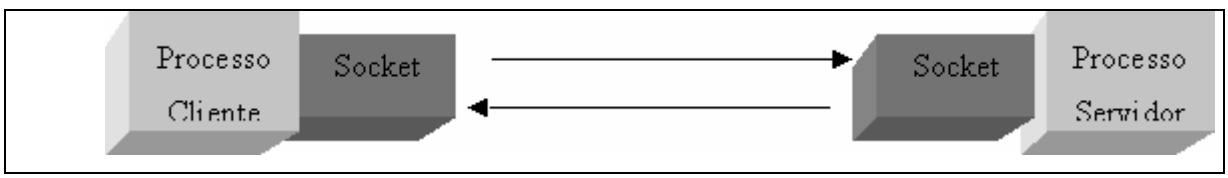

Figura 4 - Exemplo de comunicação via *Socket* TCP/IP

Os *sockets* são usados para implementar conexões confiáveis, bidirecionais e ponto a ponto, com base em *stream*, entre computadores da rede. Um *socket* pode ser usado para conectar o sistema de entrada/saída de Java a outros programas que podem residir na máquina local ou em qualquer outra máquina da rede. Os *sockets* implementam uma conexão persistente altamente confiável entre cliente e o servidor podendo especificar o endereço IP da máquina e sua porta ([CAM1996]).

A linguagem Java implementa a comunicação entre processos através de *sockets* TCP/IP, o que é muito interessante para a implementação do protótipo devido a flexibilidade propiciada.

4 JDBC

Java DataBase Connectivity (JDBC) é uma Interface de Programação de Aplicação ou *Applications Programming Interface* (API) exclusiva do Java, criada para executar comandos SQL em SGBD ([HAM1998]).

JDBC é uma camada de abstração que permite a um programa Java utilizar uma interface padrão (que adere ao ANSI-2 SQL) para acessar um banco de dados através da linguagem SQL ([RAM2000]).

No geral, existem dois níveis de interface na API JDBC: a camada de aplicação, onde o desenvolvedor usa a API para fazer chamadas para o SGBD via SQL e recebe o resultado, e a camada de *driver* que trata toda a comunicação com a implementação de um *driver* específico ([HEL1997]).

4.1 ARQUITETURA JDBC

Um programa Java utiliza uma API JDBC única que independe do SGBD ou *driver* que estiver sendo utilizado. Os *drivers* para conexão e acesso aos principais bancos de dados existentes são fornecidos pelos seus fabricantes ou por terceiros. O programador precisa apenas saber utilizar a API JDBC e a forma como o *driver* adequado se conecta ao SGBD ([RAM2000]).

Para um programa Java acessar um sistema gerenciador de banco de dados é necessário um *driver* JDBC que implemente a interface padrão da API JDBC ([RAM2000]).

Segundo Ramon ([RAM2000]), os *drivers* JDBC são, portanto, programas que fazem a intermediação entre um programa Java e um SGBD. Os *drivers* podem ser programas escritos com rotinas em linguagens diferentes de Java. Por conta disso são classificados em quatro tipos:

1. ponte JDBC-ODBC com *driver* ODBC;
2. *driver* com API parcialmente nativa;
3. *driver* puro-Java JDBC-rede;

4. *driver* puro-Java com protocolo nativo.

4.1.1 PONTE JDBC-ODBC COM DRIVER ODBC

Acesso ao SGBD via um *driver* ODBC convencional. A ponte possui código nativo a fim de fazer chamadas à API ODBC. Como pode ser visto na figura 5, tanto a ponte quanto o *driver* ODBC devem residir na máquina onde está o programa Java, uma vez que as chamadas são sempre locais ([RAM2000]).

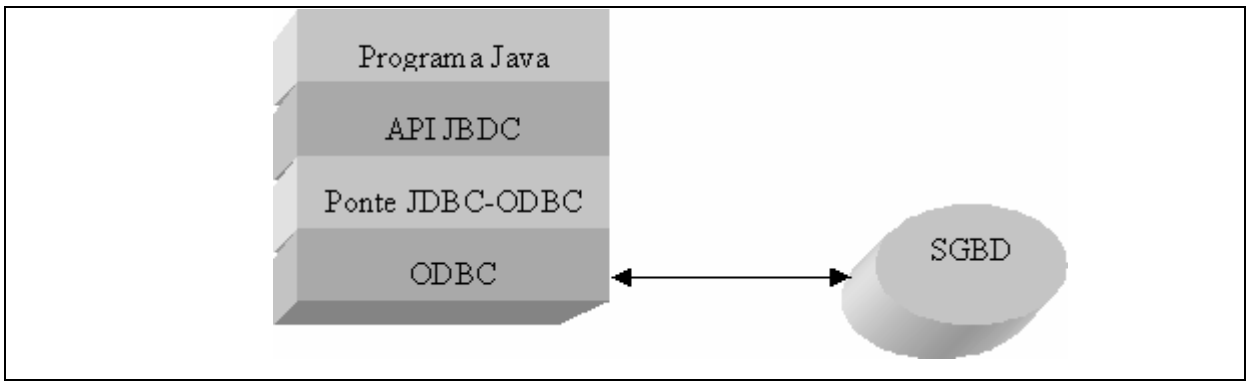

Figura 5 - Ponte JDBC-ODBC

4.1.2 DRIVER COM API PARCIALMENTE NATIVA

O *driver* JDBC acessa um programa cliente do SGBD (SQLNet, por exemplo, para o caso do Oracle). Normalmente este acesso é feito em código nativo. Cada estação deve ter instalado o programa cliente ([RAM2000]).

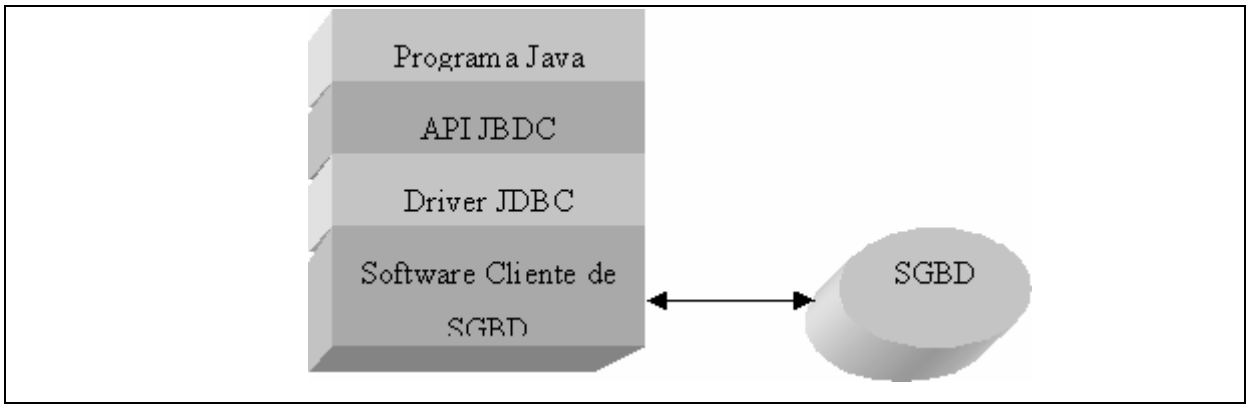

Figura 6 - API parcialmente nativa

4.1.3 DRIVER PURO-JAVA JDBC-REDE

Este é o tipo mais flexível de *driver*. O *driver* acessa um servidor e este servidor por sua vez estabelece a conexão com um ou mais SGBD ([RAM2000]).

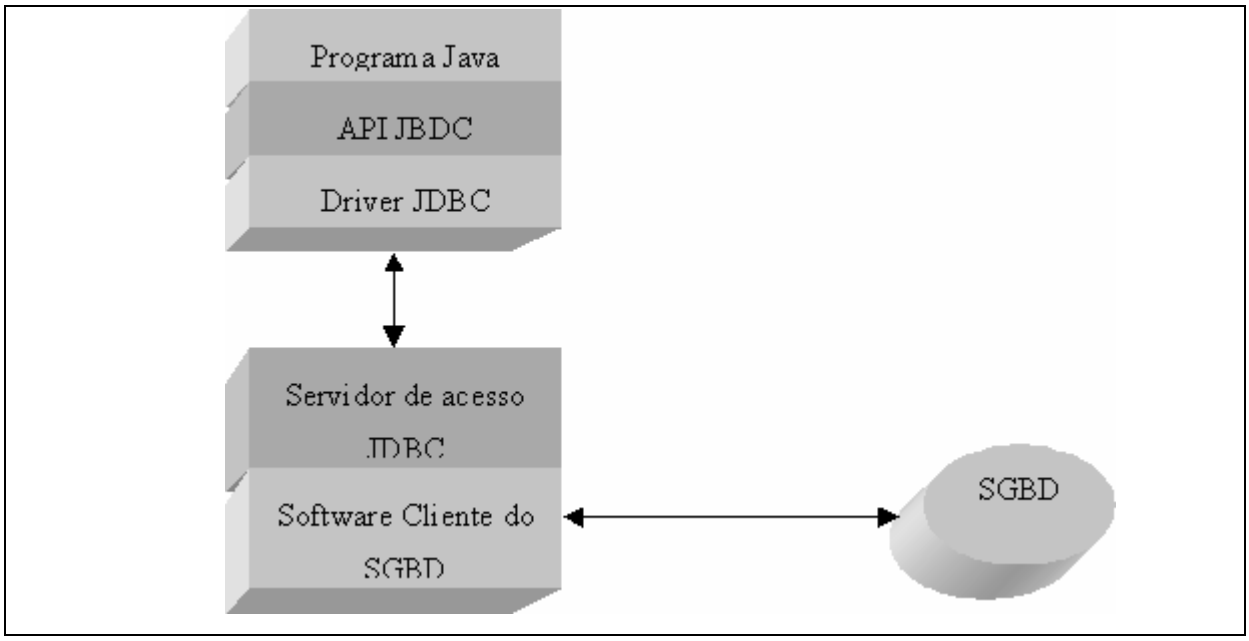

Figura 7 - Puro-Java JDBC-Rede

4.1.4 DRIVER PURO-JAVA COM PROTOCOLO NATIVO

O driver acessa diretamente o SGBD usando o protocolo nativo do mesmo. Por essa razão normalmente é fornecido pelos fabricantes de SGBD. Em geral é de tamanho reduzido e chamado de *thin driver* ([RAM2000]).

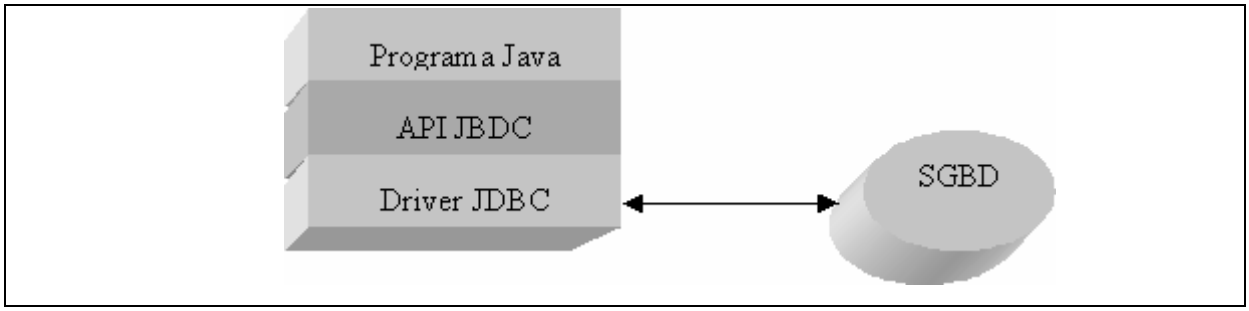

Figura 8 - Puro Java com protocolo nativo

4.2 DRIVER MANAGER

DriverManager é uma classe do pacote **java.sql** responsável principalmente por faxer a conexão dos *drivers* com o banco de dados. É uma camada de manutenção do JDBC, trabalhando entre o usuário e os *drivers* ([HAM1998]). Um programa Java pode utilizar simultaneamente vários *drivers* JDBC distintos. Todos devem estar registrados na classe *DriverManager* ([RAM2000]).

A classe *DriverManager* registra os *drivers* e estabelece a comunicação com o SGBD, selecionando o *driver* apropriado ([HAM1998]).

Um programador que utiliza um *driver* não precisa saber como ele foi codificado, mas deve saber que é suficiente carregar o *driver* para que ele se registre no *DriverManager* ([RAM2000]). Para carregar o *driver* pode-se usar o comando *Class.forName*. Exemplo:

Class.forName(“meu.driver.JDBC”);

4.3 ESTABELECENDO A CONEXÃO COM O SGBD

Para estabelecer a conexão com o SGBD é usado o método *getConnection()* da classe *DriverManager()*. O método tem a seguinte sintaxe ([HAM1998]):

DriverManager.getConnection(String URL, String Username, String Password);

A URL de conexão, utilizada para especificar o SGBD, tem o formato bem flexível. Normalmente a documentação da URL de conexão é fornecida junto com o driver ([RAM2000]). Basicamente a URL possui o seguinte formato:

jdbc:subprotocol:subname

a) *jdbc* – valor fixo que indica o protocolo JDBC;

b) *subprotocol* – mecanismo de conexão suportado pelo driver. Seu formato depende do fornecedor do driver;

c) *subname* – fonte de dados. Seu formato depende do *subprotocol* e portanto do fornecedor.

Exemplos:

Ponte JDBC-ODBC: *jdbc:odbc:ACCESS1*

Puro Java com protocolo nativo: *jdbc:oracle:thin:192.168.0.1:1521:inf2*

Exemplo do estabelecimento de uma conexão:

Connection conn = DriverManager.getConnection(“jdbc:odbc:acc”,“sys”,“master”);

4.4 ENVIANDO SQL E RECEBENDO RESULTADOS

A classe *Statement* é utilizada para enviar comandos SQL para um SGBD. Para criar um *Statement* é usado o comando *createStatement* ([RAM2000]). Exemplo:

Statement stmt = conn.createStatement();

Um *Statement* está associado a um contexto no SGBD, e uma mesma conexão pode suportar vários *Statements* simultaneamente ([HAM1998]).

A classe *Statement* possui entre outros, os métodos *executeQuery()* e *executeUpdate()*. Para executar consultas (*Select*) em um SGBD, é utilizado o método *executeQuery()*, que recebe o comando como parâmetro e retorna um *ResultSet* com o resultado da consulta. Para executar alterações na base de dados (Insert, Delete, Update, Create Table, Create Index, etc) e utilizado o método *executeUpdate()*, que recebe o comando como parâmetro e retorna um *int* que indica o número de linhas afetadas pelo comando ([HAM1998] [RAM2000]). Exemplo:

int numerolinhas = stmt. executeUpdate(“delete from table1”);

A classe *ResultSet*, que é retornada pelo *executeQuery()*, possui todos os métodos (*next(), getString(), etc*) necessários para obter os valores de todos os campos em cada linha ([HAM1998]). O *ResultSet* é uma tabela que contém o resultado da execução da consulta, em outras palavras, ele contém as linhas que satisfazem as condições da consulta ([RAM2000]). Exemplo:

ResultSet rs = stmt. executeQuery(“select cod from table1”);

4.5 ETAPAS DA PROGRAMAÇÃO COM JDBC

Segundo Ramon ([RAM2000]), a programação com JDBC pode ser dividida em sete etapas:

1) importar java.sql.*;

2) carregar driver JDBC;

3) especificar um SGBD para a conexão;

4) abrir uma conexão com o SGBD;

5) criar um *Statement*;

6) submeter o comando SQL;

7) receber o resultado.

5 DESENVOLVIMENTO DO PROTÓTIPO

O problema relatado na introdução deste trabalho é uma constante em muitas instituições. A distribuição dos dados pelos diversos setores de uma empresa é um problema muito grande quando surge a necessidade de integrar os múltiplos SGBD independentes e heterogêneos.

Para resolver este problema, propõe-se a construção de um protótipo de sistema que possibilitará a conexão do cliente com diversos SGBD de forma transparente ao usuário.

Pode-se imaginar que um usuário precisa acessar um sistema que esteja distribuído em vários departamentos de uma mesma empresa, sistema este que poderia ser o cadastro de clientes da empresa. O usuário tem duas opções, poderia ir até cada um dos departamentos e utilizar o sistema específico de cada departamento para acessar os dados, ou então poderia se conectar com um *middleware* que receberia os comandos do usuário e faria o acesso aos diversos SGBD que estão distribuídos como se fossem apenas um, de forma transparente, sem que o usuário precise necessariamente saber onde os dados estão armazenados fisicamente.

O protótipo foi especificado utilizando a técnica UML (*Unified Modeling Language*) e implementado na linguagem Java utilizando a ferramenta JBuilder 2.0.

Como SGBD para exemplo e teste no desenvolvimento do protótipo, foram utilizados o Oracle 8i, Interbase 5.5 e Access 7.0, justamente por serem SGBD de uso bastante difundido e conseqüentemente mais fácil de obtê-los para o desenvolvimento.

O Oracle 8i, desenvolvido pela Oracle Corporation, e o Interbase 5.5, desenvolvido pela Inprise Corporation, são bancos de dados relacionais de grande porte, que conseguem trabalhar com uma grande quantidade de dados sem comprometer a confiabilidade e o desempenho. O Access 7.0, desenvolvido pela Microsoft Corporation, já é um banco de dados bem mais modesto, desenvolvido para trabalhar com pequena quantidade de dados. A opção por estes SGBD foi principalmente a facilidade de obtê-los para o desenvolvimento do protótipo.

Para a especificação do protótipo foi levantado o diagrama de casos de uso apresentado na figura 9:

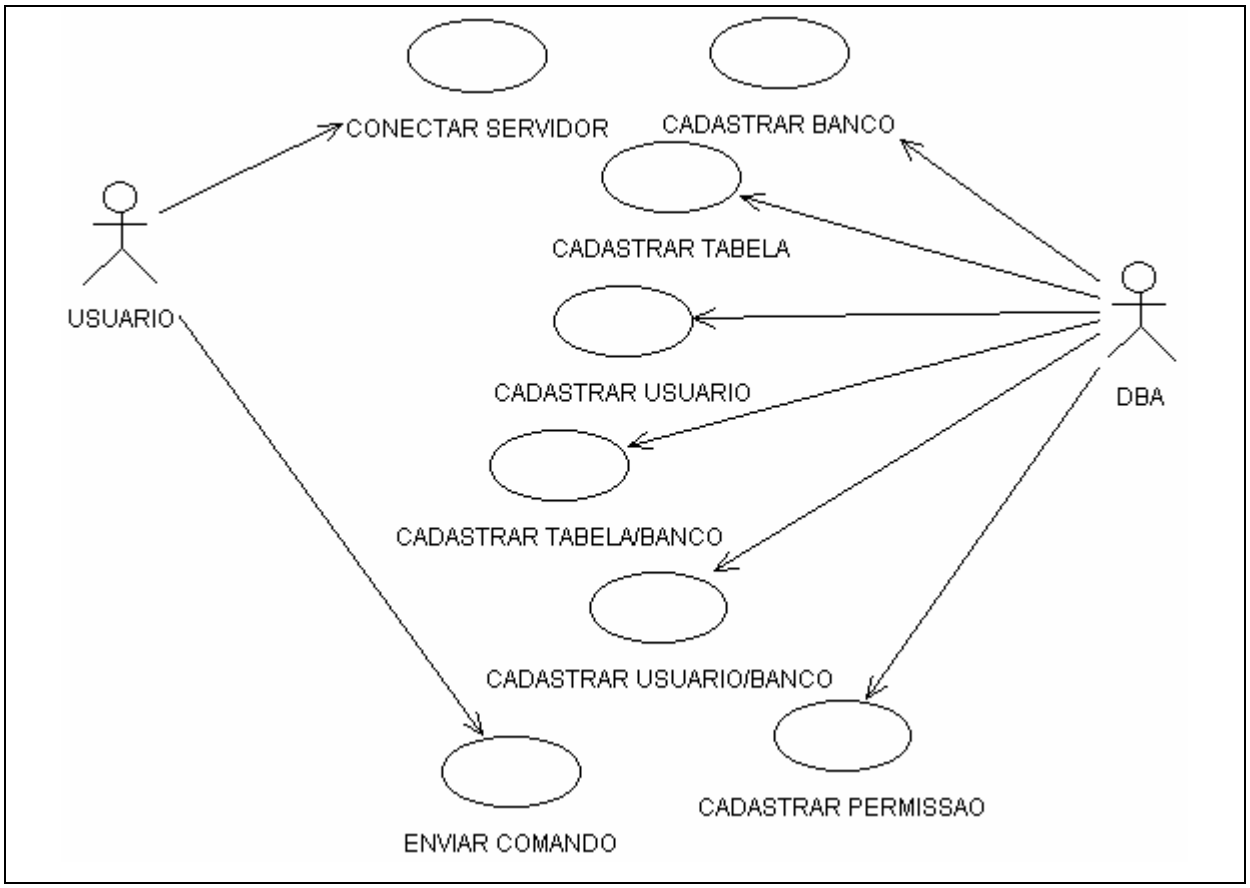

Figura 9 - Casos de uso

Descrição dos casos de uso:

a) cadastrar banco – o DBA cadastra os SGBD que serão possíveis de acessar através do protótipo;

b) cadastrar tabela – o DBA cadastra as tabelas que estão nos SGBD e que poderão ser utilizadas;

c) cadastrar usuário – o DBA cadastra os usuários do protótipo;

d) cadastrar tabela/banco – o DBA associa as tabelas com os devidos SGBD onde as mesmas estão localizadas, sendo que uma mesma tabela pode estar em mais de um SGBD;

e) cadastrar usuário/banco – o DBA associa o usuário aos diversos SGBD que ele terá acesso;

f) cadastrar permissão – o DBA define as permissões do usuário dentro do sistema. Ele pode controlar o modo de acesso para cada tabela em cada SGBD para cada usuário;

g) conectar servidor – o usuário efetua a conexão com o protótipo;

h) enviar comando – o usuário envia o comando para o protótipo, e recebe o resultado.

A partir do diagrama de casos de uso da figura 9, foi obtido o diagrama de classes da figura 10:

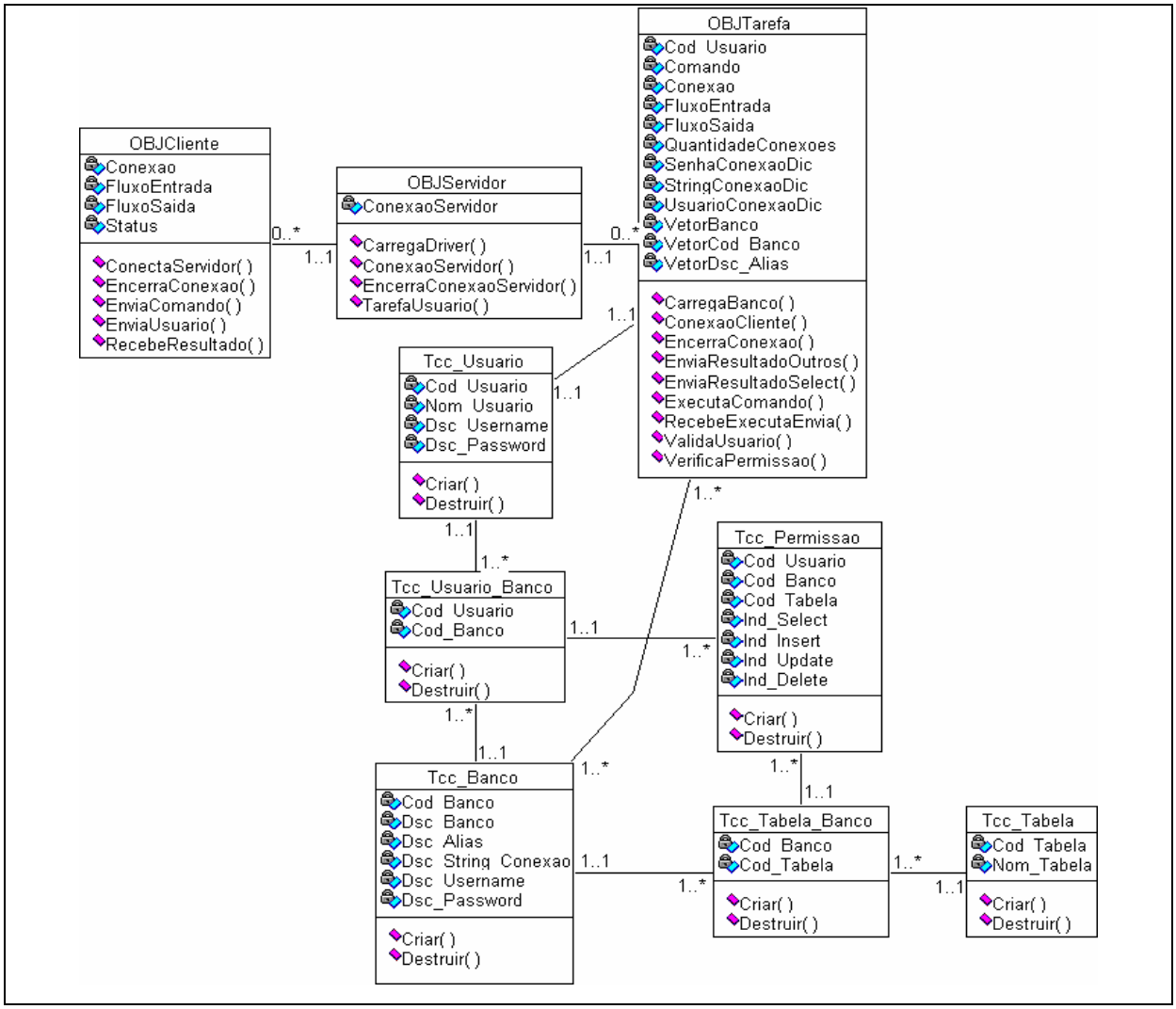

Figura 10 - Diagrama de Classes

Sobre diagrama de classes da figura 10, podemos dizer que as classes que serão efetivamente implementadas são a OBJCliente, OBJServidor e a OBJTarefa, sendo que as demais são classes persistentes e que originaram o Modelo Entidade Relacionamento da figura 11:

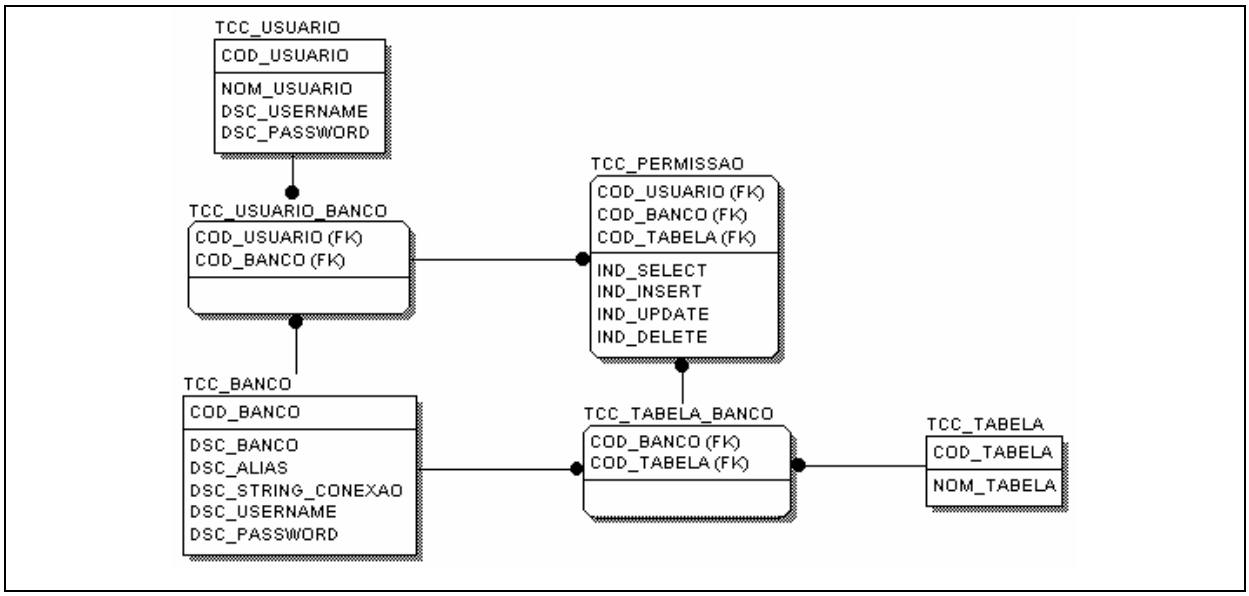

Figura 11 - Modelo entidade relacionamento

A partir do diagrama de casos de uso da figura 9 foram gerados os diagramas de seqüência apresentados nas figuras 12 e 13. Gerou-se diagramas de seqüência somente para os casos de uso do usuário, por serem os mais relevantes.

O diagrama de seqüência da figura 12 corresponde ao primeiro caso de uso do usuário, onde ele conecta ao servidor. O usuário estabelece uma conexão com o servidor, o qual cria uma nova tarefa, esta tarefa valida o usuário e efetua a carga dos SGBD que o usuário tem acesso.

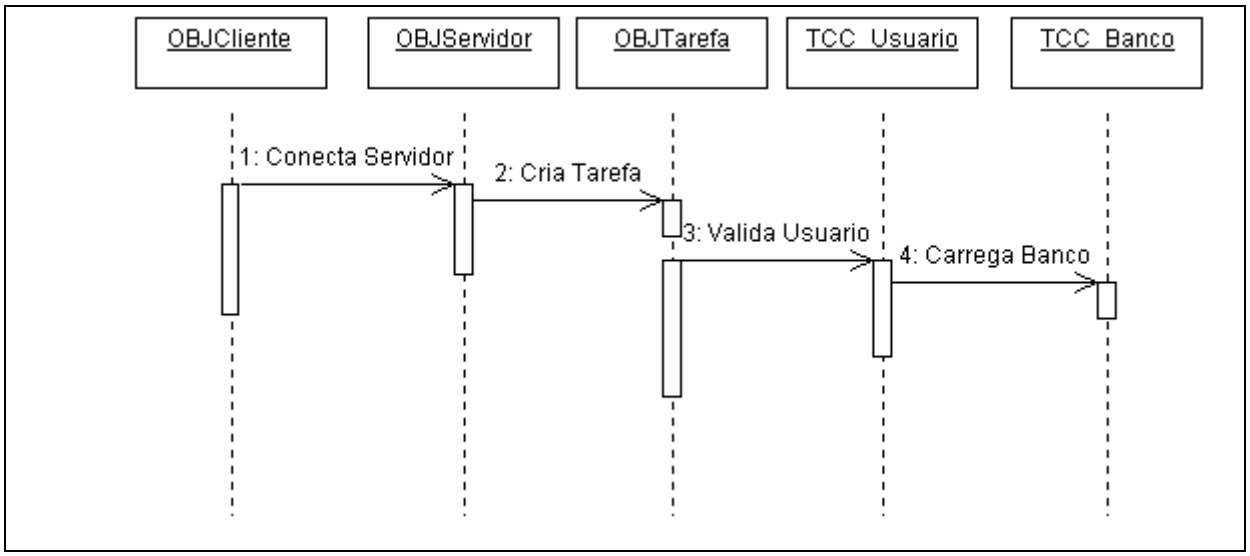

Figura 12 – Diagrama de seqüência

O diagrama de seqüência da figura 13 corresponde ao segundo caso de uso do usuário, quando ele envia o comando para a tarefa, a qual recebe o comando e verifica as permissões de acesso do usuário antes de executar o comando.

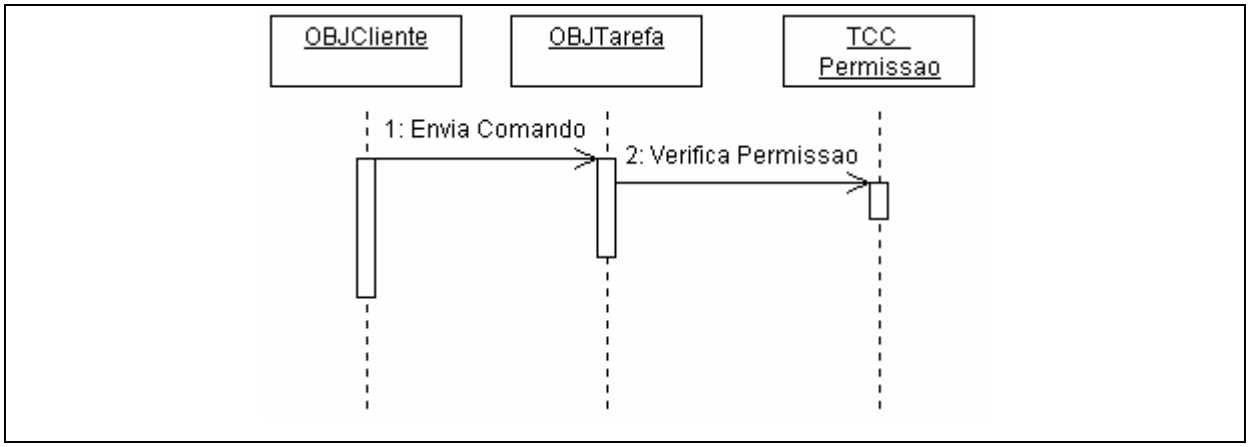

Figura 13 - Diagrama de Seqüencia

5.1 DICIONÁRIO DE DISTRIBUIÇÃO

Para administrar a distribuição dos SGBD, precisaram ser criadas as tabelas descritas no Modelo de Entidade Relacionamento da figura 11. O Dicionário de Distribuição é o coração do sistema, pois irá armazenar os Usuários, localização dos SGBD, Tabelas e identificará as permissões de cada usuário. Para manter o Dicionário de Distribuição foi criado um protótipo a parte.

Para acessar o SGBD é preciso saber a URL (*string* de conexão) do JDBC e ter definido um usuário e uma senha. Cada usuário que possuir acesso ao *middleware* terá que possuir os SGBD que necessitar associados a ele.

As permissões de acesso são configuráveis para cada usuário. É possível definir quais tabelas ele poderá ver, bem como o tipo de acesso (INSERT, DELETE, UPDATE, SELECT), sendo que esta configuração pode mudar de SGBD para SGBD.

5.1.1 MANUTENÇÃO DO DICIONÁRIO DE DISTRIBUIÇÃO

O dicionário de distribuição que foi gerado na especificação do protótipo necessita de um outro protótipo especifico para dar manutenção nos dados que serão armazenados. Para isso foi desenvolvido um protótipo a parte com as seguintes telas (figuras 14 a 20):

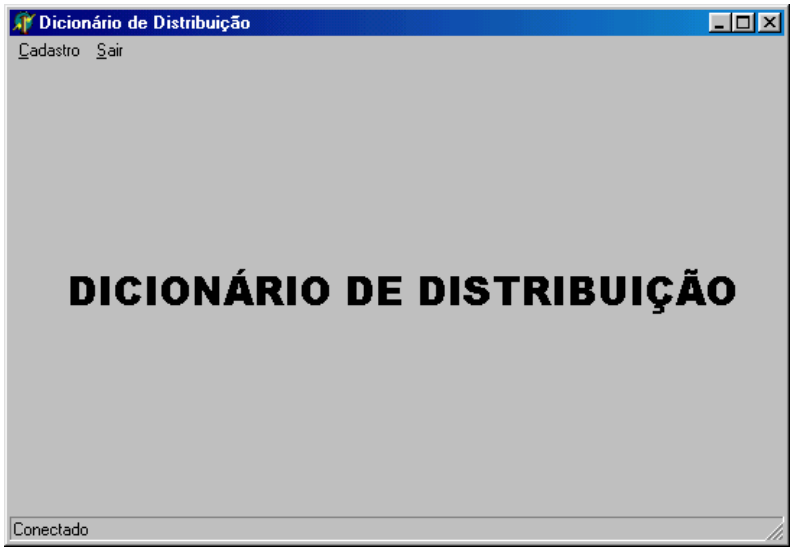

Figura 14 - Tela principal do sistema de manutenção do dicionário de distribuição

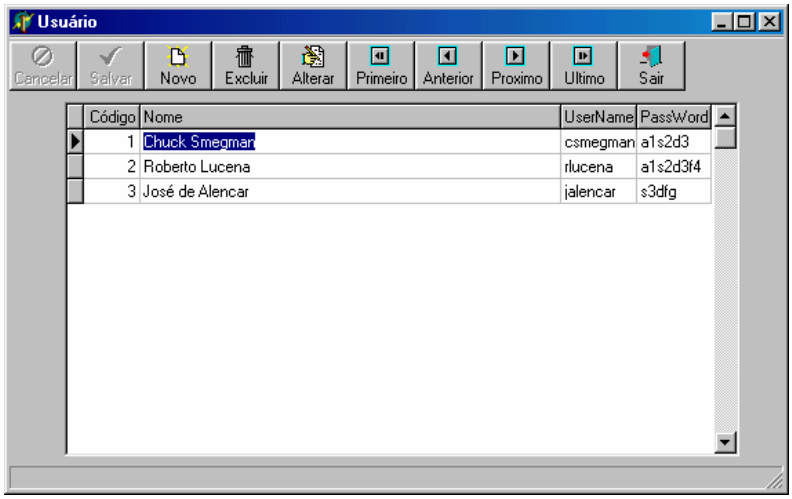

Figura 15 - Tela para cadastro dos usuários do protótipo

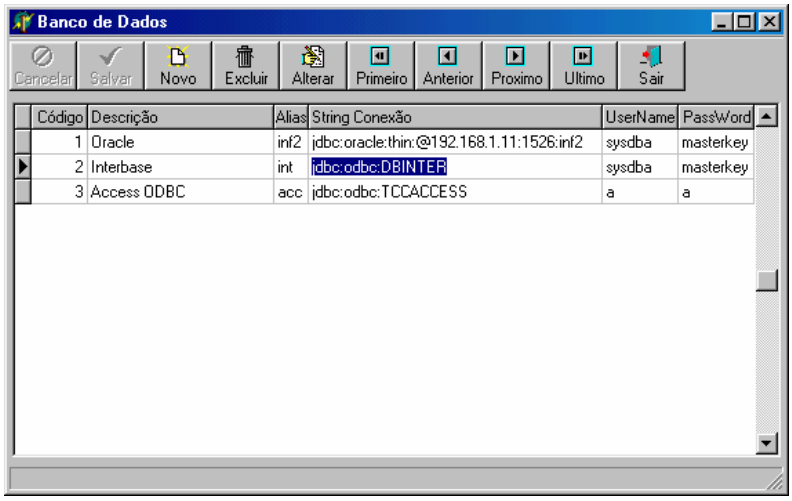

Figura 16 - Tela para cadastro dos SGBDs

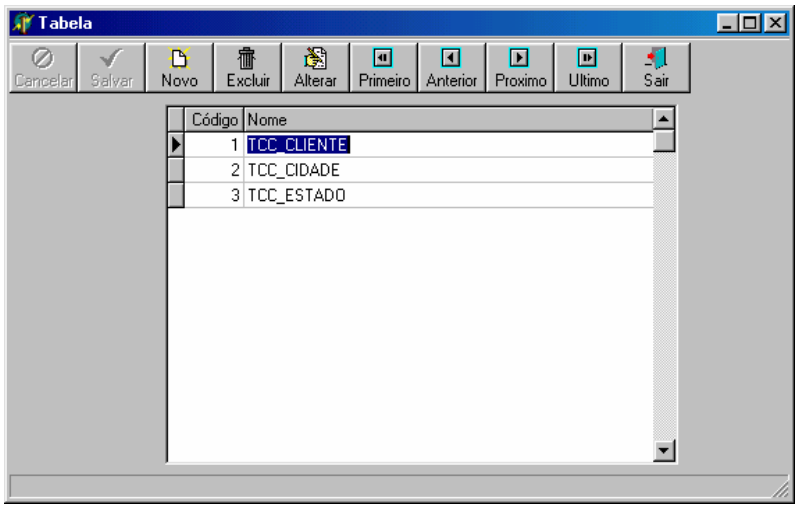

Figura 17 - Tela para cadastro das tabelas

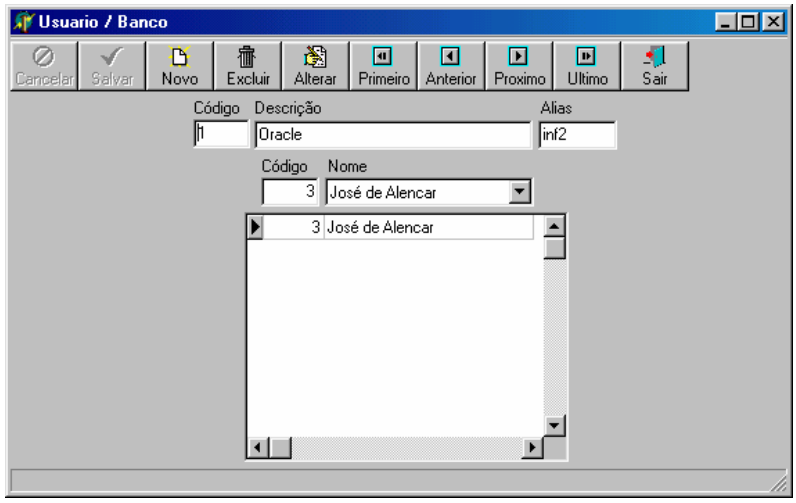

Figura 18 - Tela para associar os usuários aos SGBDs

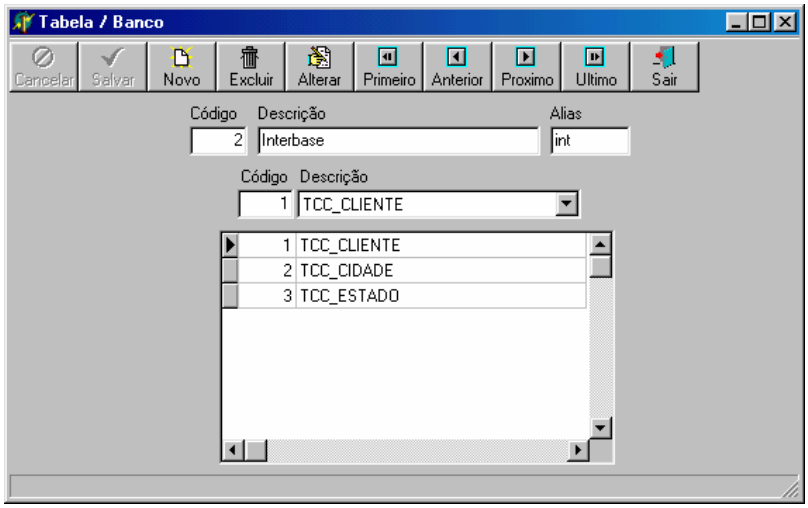

Figura 19 - Tela para associar as tabelas aos SGBDs

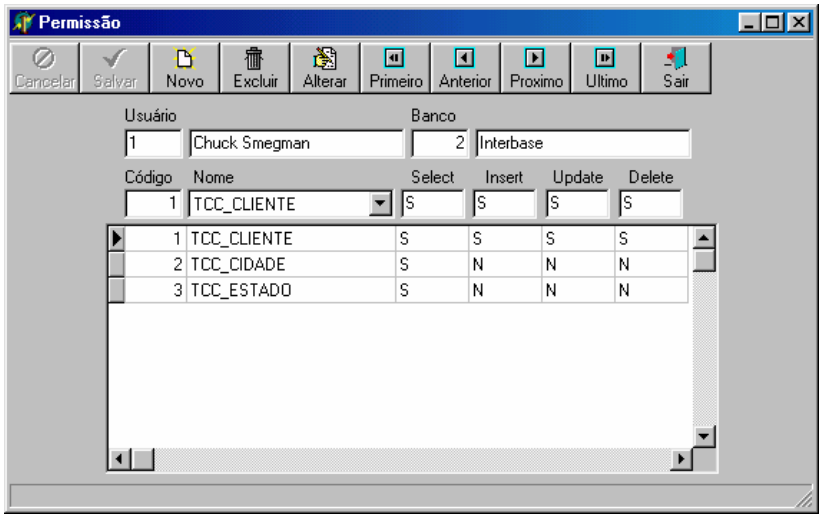

Figura 20 - Tela para definir as permissões de cada usuário

O protótipo para manutenção do dicionário de distribuição foi implementado na linguagem Delphi 5.0, enquanto que as tabelas ficam armazenadas no SGBD Interbase 5.5.

5.2 FUNCIONAMENTO DO PROTÓTIPO

O protótipo foi implementado utilizando três classes, a classe OBJCliente, a classe OBJServidor e a classe OBJTarefa, cujo código fonte encontra-se no capítulo 7.

Funcionamento básico de cada uma das classes:

a) classe OBJServidor – faz a monitoração de uma porta específica do computador, reservada para a conexão com os clientes. A cada conexão de um cliente (através do *socket* TCP/IP na porta definida) é instanciada um novo OBJTareja que passa a manter e a gerenciar esta conexão, permitindo então que o *middleware* mantenha o controle de vários OBJCliente simultaneamente. Esta classe não possui interface visual com o usuário e está descrita no anexo 7.2;

b) classe OBJTarefa – nesta classe estão as rotinas de atendimento ao OBJCliente e de acesso aos SGBDs. A classe é responsável por verificar se é um usuário válido que esta acessando o *middleware*, consultar a quais SGBDs o usuário tem acesso, estabelecer a conexão com cada um dos SGBDs, receber o comando do usuário, verificar as permissões de acesso à tabelas que o usuário possui, executar o comando e enviar o resultado para o usuário. Esta classe não possui interface visual com o usuário e está descrita no anexo 7.3;

c) classe OBJCliente – esta classe é responsável pela conexão entre o usuário e o *middleware*. Ela que faz a conexão com o OBJServidor, envia o login de acesso, envia comandos para o OBJTarefa e recebe os resultados do comando. É a classe que faz a interface visual com o usuário e está descrita no anexo 7.1.

O funcionamento do protótipo é relativamente simples:

a) entra em funcionamento o OBJServidor, que faz a carga dos *drivers* e fica aguardando que algum OBJCliente faça conexão com ele através de uma porta lógica pré-definida;

b) assim que o OBJCliente efetua a conexão com o OBJServidor, este imediatamente cria um novo OBJTarefa que fica responsável por toda a comunicação com o OBJCliente;

c) o OBJCliente envia o usuário e a senha para validação;

d) caso seja um usuário válido, então o OBJTarefa envia um sinal positivo e verifica quais os SGBDs que o usuário tem cadastrado para em seguida efetuar a conexão com eles;

e) estando os SGBDs conectados o OBJTarefa aguarda comandos para executar;

f) ao receber o sinal positivo, o OBJCliente está apto para enviar comandos para o OBJTarefa;

g) quando o OBJTarefa recebe um comando do OBJCliente, ele verifica qual comando foi enviado para fazer a verificação das permissões de acesso;

h) O OBJTarefa separa as tabelas do comando e verifica em todos os SGBDs se determinada tabela esta cadastrada e quais os tipos de acesso permitido;

i) Em seguida, caso seja um comando permitido o OBJTarefa executa o comando, enviando o resultado para o OBJCliente, que recebe o resultado e exibe na tela;

j) A conexão encerra quando o OBJCliente enviar um comando pedindo o fim da conexão ou caso o usuário não seja um usuário válido.

Um diagrama do ambiente em que o protótipo opera pode ser visto na figura 21. Os quadros em cinza representam o que foi criado neste trabalho.

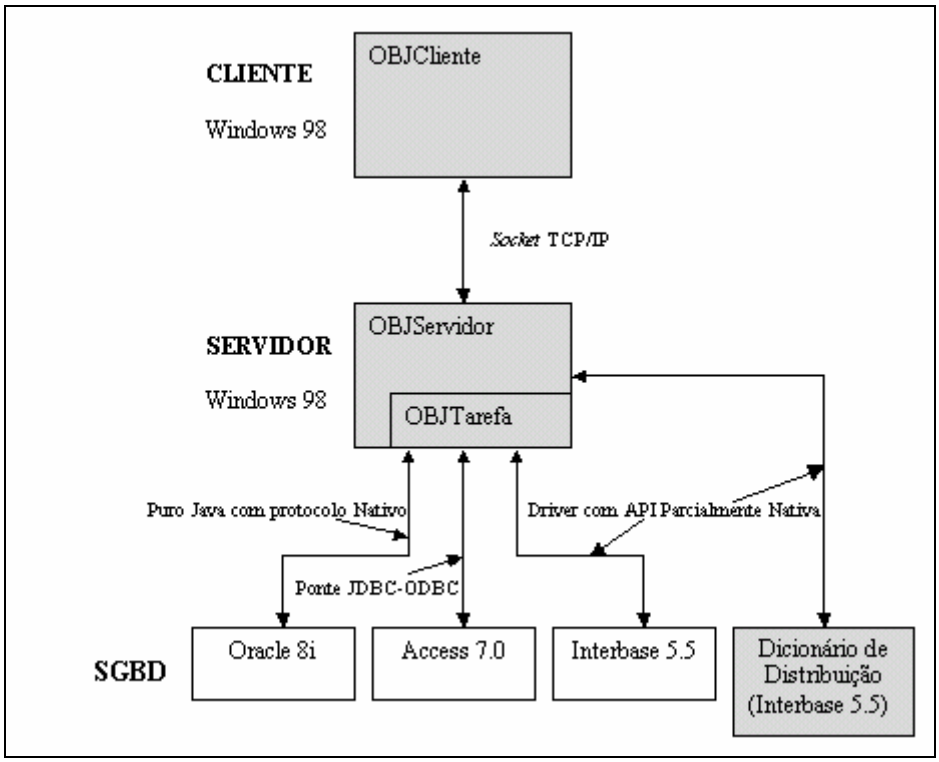

Figura 21 - Diagrama do ambiente de operação

As figuras 22 a 26 mostram um exemplo do funcionamento do protótipo, onde o usuário executa um *insert* no SGBD Interbase 5.5.

Na figura 22 o usuário conecta-se ao Servidor e envia o comando *select*. O servidor retorna os dados armazenados nos diversos SGBD.

Figura 22 - Select na base de dados através do protótipo

Na figura 23 o usuário executa o mesmo *select* diretamente no SGBD do Interbase 5.5, onde serão exibidos somente os dados que estão armazenados neste SGBD.

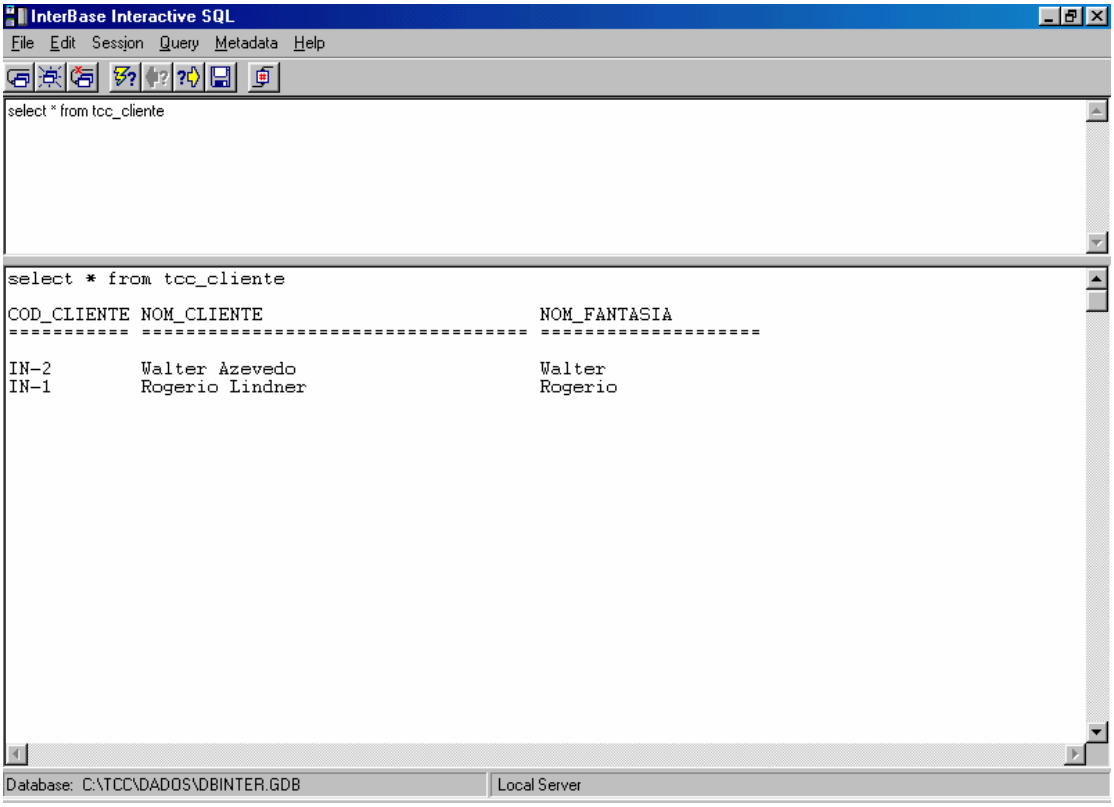

Figura 23 – Select no Interbase 5.5 através do próprio SGBD

Na figura 24 o usuário executa um *insert* no SGBD Interbase 5.5 através do protótipo.

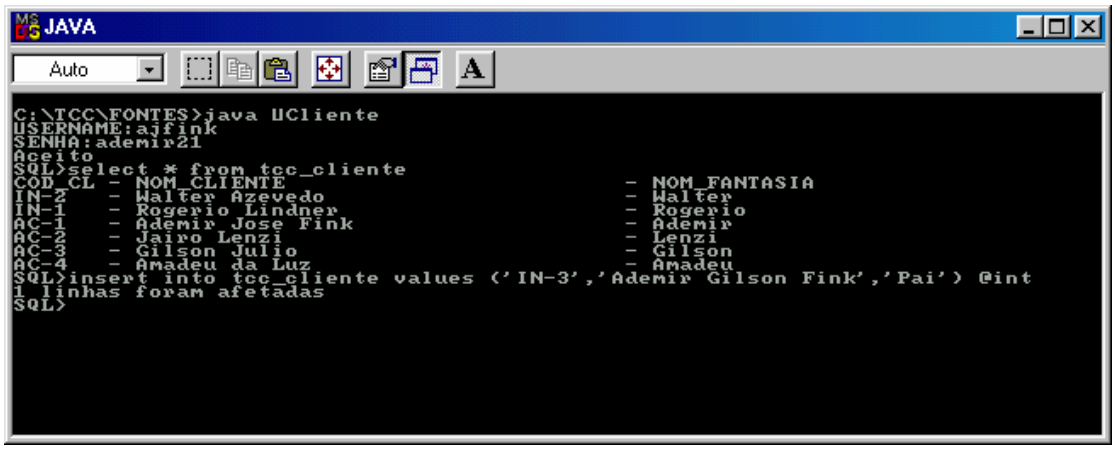

Figura 24 - Insert no Banco de Dados Interbase 5.5 através do protótipo

Na figura 25 o usuário executa um *select* através do protótipo o qual retorna novamente os dados armazenados nos diversos SGBD disponíveis, inclusive com os dados que ele acabou de inserir na figura 24.

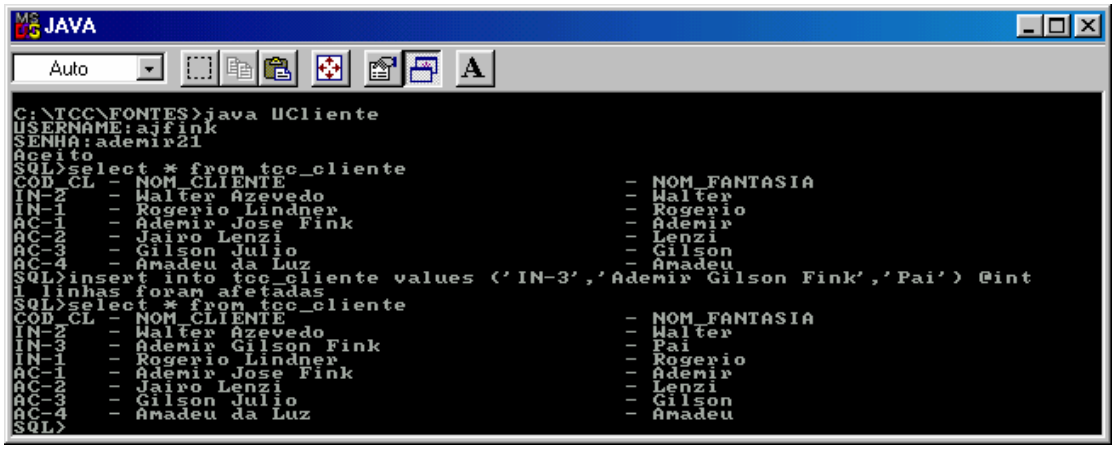

Figura 25 - Select pelo protótipo após o insert no Interbase 5.5

Na figura 26 o usuário executa o mesmo comando diretamente no SGBD Interbase 5.5 somente para confirmar se os dados foram mesmo cadastrados com sucesso.

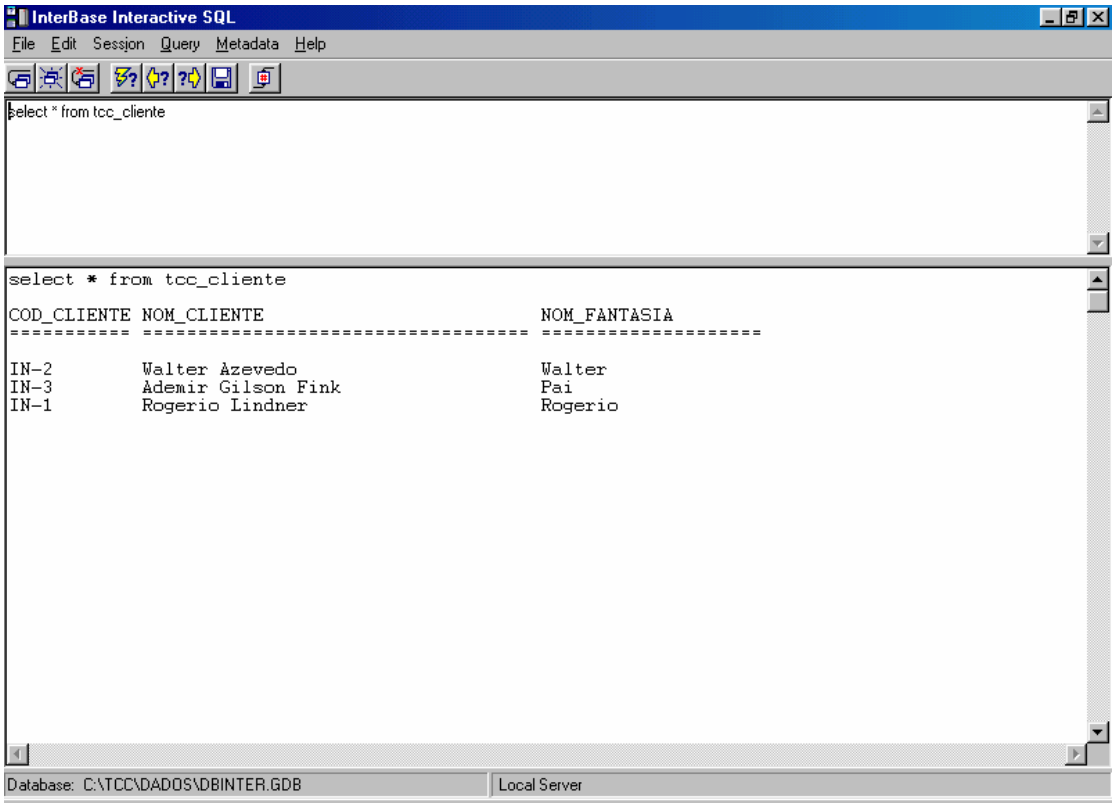

Figura 26 - Select no Interbase 5.5 através do próprio SGBD após o insert pelo protótipo

6 CONCLUSÃO

As empresas, dentro do mercado globalizado e competitivo de hoje, precisam de informações com alto grau de disponibilidade, rapidez, segurança e controle, para que elas tenham produtividade e qualidade nos serviços e produtos em que elas atuam.

Um sistema de banco de dados distribuído fornece um ambiente no qual novas aplicações de banco de dados podem fazer o acesso a dados a partir de uma variedade de banco de dados preexistentes localizados em ambientes de hardware e software heterogêneos.

A viabilidade de acesso a SGBD de diferentes fornecedores foi resultado da opção pela API JDBC. Utilizando esta API, é simples enviar comandos SQL a uma grande variedade de SGBD relacionais, a API JDBC possibilita que seja escrito um único programa que acesse os dados contidos em vários SGBD diferentes.

Para acessar os diferentes SGBD é necessário armazenar no dicionário de distribuição quais as URL, usuários e tabelas são utilizados em cada um dos SGBD. A manutenção no dicionário de distribuição é uma ação dinâmica, que pode ser realizada a qualquer momento em tempo de execução, sem necessidade de manutenção no protótipo.

Outro fato que corroborou para a interoperabilidade do sistema foi a opção por *sockets* TCP/IP na comunicação entre usuário e *middleware*. Com *sockets* TCP/IP, o programa do usuário pode ser implementado em qualquer linguagem que ofereça suporte a comunicação entre processos em redes TCP/IP, tendo em vista que o protocolo TCP/IP tem a vantagem sobre os demais de ser compatível com um grande número de diferentes sistemas de hardware e software.

Uma das principais constatações feitas durante o trabalho é que a linguagem Java é de fácil utilização, sendo apropriada para construção de aplicativos deste gênero principalmente pelos recursos que possui. A linguagem Java foi desenvolvida para ter as redes como seu ambiente operacional e especificada para ser independente de plataforma e tendo como trunfo a API JDBC para acesso a banco de dados.

É possível concluir que o objetivo deste trabalho foi alcançado, tendo em vista que a proposta de estudo e utilização de JDBC para conexão de SGBD distribuídos e heterogêneos

foi validada com a construção do protótipo, tornando possível o acesso a vários SGBDs simultaneamente, como se fossem apenas um. O exemplo de execução apresentado nas figuras 22 a 26, onde foi realizado *select* e *insert* em SGBD heterogêneos e localizados em plataformas diferentes, apenas confirma o sucesso do protótipo.

6.1 LIMITAÇÕES E SUGESTÕES PARA FUTUROS TRABALHOS

Existem principalmente três limitações neste *middleware*:

a) não é possível fazer JOIN entre tabelas equivalentes de SGBD diferentes, ou seja, este tipo de comando é válido apenas para tabelas que se encontram no mesmo SGBD. Poderia ser implementada a possibilidade de os SGBD poderem interagir um com o outro, criando assim a possibilidade de fazer o JOIN entre tabelas de diferentes SGBD;

b) tabelas que possuem o mesmo nome mas que estão em SGBD diferentes, terão que possuir a mesma estrutura ou pelo menos os mesmos campos para fornecer um resultado satisfatório. Imagine que exista a tabela FATURAMENTO em dois SGBD diferentes, em um SGBD a tabela armazena as duplicatas e em outro SGBD a tabela com o mesmo nome armazena um acumulado do faturamento. Embora as duas tabelas tenham o mesmo nome tem pouco a ver uma em relação a outra. Poderia ser implementado um mecanismo para administrar este problema;

c) o *middleware* faz uma carga estática dos drivers dos SGBDs acessados. Cada vez que for preciso adicionar um novo tipo de SGBD ao *middleware* terá que ser feito manutenção no código fonte. O protótipo poderia ser incrementado para fazer uma carga dinâmica de *drivers* JDBC, configurando o número de diferentes tipos de SGBD acessíveis em tempo de execução.

7 ANEXOS

7.1 ANEXO 1 – OBJCLIENTE

```
import java.net.*;
import java.io.*;
import java.lang.*;

class OBJCliente
{
  Socket          Conexao      = null;
  DataInputStream FluxoEntrada = null;
  PrintStream     FluxoSaida   = null;
  String          Status       = "Recusado";

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void ConectaServidor()
  {
    try
    {
      Conexao = new Socket("192.168.0.1",8086);
      FluxoEntrada =
            new DataInputStream(Conexao.getInputStream());
      FluxoSaida = new PrintStream(Conexao.getOutputStream());
    }
    catch (Exception Erro)
    {
      System.err.println("Erro na criacao do Cliente. "+Erro.getMessage());
      System.exit(1);
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void EnviaUsuario(String Dsc_Username,String Dsc_Password)
  {
    String Usuario = null;

    String OitoEspacos = "        ";
    try
    {
      Dsc_Username = Dsc_Username +
OitoEspacos.substring((Dsc_Username.length()),8);
      Dsc_Password = Dsc_Password +
OitoEspacos.substring((Dsc_Password.length()),8);
      Usuario = "##"+Dsc_Username+Dsc_Password+"##";
      FluxoSaida.println(Usuario);
      FluxoSaida.flush();
      Status = FluxoEntrada.readLine();
      System.out.println(Status);
    }
    catch (Exception Erro)
    {
      System.err.println("Erro ao enviar o Usuario. "+Erro.getMessage());
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
```

```
  void EnviaComando(String Comando)
  {
    try
    {
      if (Status.equals("Aceito"))
      {
        FluxoSaida.println(Comando);
        FluxoSaida.flush();
        RecebeResultado();
      }
    }
    catch (Exception Erro)
    {
      System.err.println("Erro enviando o comando. "+Erro.getMessage());
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void RecebeResultado()
  {
    String Resultado;
    try
    {
      Resultado = FluxoEntrada.readLine();
      while (!(Resultado.startsWith("FIM"))
          && !(Resultado.startsWith("ERRO"))
          && !(Resultado.startsWith("QUIT")))
      {
        System.out.println(Resultado);
        Resultado = FluxoEntrada.readLine();
      }
      if (Resultado.startsWith("ERRO"))
      {
        System.out.println(Resultado);
      }
    }
    catch (IOException Erro)
    {
      System.err.println("Erro recebendo o Resultado. "+Erro.getMessage());
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void EncerraConexao()
  {
    try
    {
      FluxoEntrada = new DataInputStream(Conexao.getInputStream());
      FluxoSaida = new PrintStream(Conexao.getOutputStream());
    }
    catch (Exception Erro)
    {
      System.err.println("Erro ao encerrar a Conexao. "+Erro.getMessage());
    }
  }

}

class UCliente
```

```
{
  public static void main(String[] args)
  {
    DataInputStream Teclado = null;
    String          Dsc_Username = null;
    String          Dsc_Password = null;
    int             Ind = 1;
    String          Comando = " ";

    Teclado = new DataInputStream(System.in);
    OBJCliente Cliente = new OBJCliente();
    Cliente.ConectaServidor();
    while (Ind <= 3 && Cliente.Status.equals("Recusado"))
    {
      try
      {
        System.out.print("USERNAME:");
        Dsc_Username = Teclado.readLine();
        System.out.print("SENHA:");
        Dsc_Password = Teclado.readLine();
      }
      catch (Exception Erro)
      {
        System.err.println("Erro ao ler o Teclado. "+Erro.getMessage());
      }
      Cliente.EnviaUsuario(Dsc_Username,Dsc_Password);
      Ind = Ind + 1;
    }
    try
    {
      while (!Comando.toUpperCase().trim().equals("FIM") &&
!Cliente.Status.equals("Recusado"))
      {
        System.out.print("SQL>");
        Comando = Teclado.readLine();
        Cliente.EnviaComando(Comando);
      }
    }
    catch (Exception Erro)
    {
      System.err.println("Erro tratando o comando. "+Erro.getMessage());
    }
    Cliente.EncerraConexao();
  }
}
```

7.2 ANEXO 2 – OBJSERVIDOR

```
import java.net.*;
import java.io.*;
import java.sql.*;

class OBJServidor
{
  // propriedades
  ServerSocket ConexaoServidor = null;
```

```
  //metodos e funcoes

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void ConexaoServidor(int PortaLogica)
  //procedure que cria a conexao do servidor com a porta logica
  {
    try
    {
      ConexaoServidor = new ServerSocket(PortaLogica); //a porta logica
    }
    catch (IOException Erro) //trata um possivel erro
    {
      System.err.println("Erro ao criar o Servidor " + PortaLogica + ", " +
Erro.getMessage());
      System.exit(1);
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void CarregaDriver()
  //procedure que carrega os drivers dos bancos que serao acessados
  {
    try
    {
      Class.forName("interbase.interclient.Driver"); //driver Interbase
      Class.forName("oracle.jdbc.driver.OracleDriver"); //driver Oracle
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //driver odbc
    }
    catch (Exception Erro) //trata um possivel erro
    {
      System.err.println("Erro ao carregar o Driver. "+Erro.getMessage());
      System.exit(1);
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void TarefaUsuario()
  //procedure que escuta a porta logica e cria as threads para os usuarios
  {
    Socket ConexaoCliente = null;
    while (true)
    {
      try
      {
        ConexaoCliente = ConexaoServidor.accept();
      }
      catch (IOException Erro)
      {
        System.err.println("Erro ao criar a nova Tarefa " + 8086 + ", " +
Erro.getMessage());
        continue;
      }
      new Tarefa(ConexaoCliente).start();
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void EncerraConexaoServidor()
  //procedure que encerra a conexao servidor
```

```
  {
    try
    {
      ConexaoServidor.close();
    }
    catch (Exception Erro)
    {
      System.err.println("Nao e possivel encerrar a conexao Servidor. " +
Erro.getMessage());
    }
  }

}


class UServidor
{
  public static void main(String[] args)
  {
    OBJServidor Servidor = new OBJServidor();
    Servidor.ConexaoServidor(8086);
    Servidor.CarregaDriver();
    Servidor.TarefaUsuario();
    Servidor.EncerraConexaoServidor();
  }
}
```

7.3 ANEXO 3 – OBJTAREFA

```
import java.net.*;
import java.io.*;
import java.sql.*;
import java.util.*;
import java.lang.*;

class Tarefa extends Thread
{
  Socket          Conexao            = null;
  DataInputStream FluxoEntrada       = null;
  PrintStream     FluxoSaida         = null;
  String          Cod_Usuario        = null;
  String          Comando            = null;
  Connection      VetorBanco[ ]      = null;
  String          VetorCod_Banco[ ]  = null;
  String          VetorDsc_Alias[ ]  = null;
  int             QuantidadeConexoes = 0;
  String          StringConexaoDic   = "jdbc:odbc:DBINTER";
  String          UsuarioConexaoDic  = "sysdba";
  String          SenhaConexaoDic    = "masterkey";


  Tarefa(Socket ConexaoCliente)
  {
    this.Conexao = ConexaoCliente;
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
```

```
  void ConexaoCliente()
  {
    try
    {
      FluxoEntrada = new DataInputStream(Conexao.getInputStream());
      FluxoSaida = new PrintStream(Conexao.getOutputStream());
    }
    catch (Exception Erro)
    {
      System.err.println("Erro na criacao da Tarefa. "+Erro.getMessage());
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void ValidaUsuario(String Usuario)
  {
    String     UserDic      = null;
    String     PassDic      = null;
    Connection BancoDic     = null;
    Statement  EnviaSQLDic  = null;
    ResultSet  ResultadoDic = null;


    try
    {
      if ((Usuario.startsWith("##")) && (Usuario.endsWith("##")))
      {
        UserDic = Usuario.substring(2,10).trim();
        PassDic = Usuario.substring(10,18).trim();
        BancoDic = DriverManager.getConnection (StringConexaoDic,
UsuarioConexaoDic, SenhaConexaoDic);
        EnviaSQLDic = BancoDic.createStatement();
        ResultadoDic = EnviaSQLDic.executeQuery("select cod_usuario "+
                                                "from tcc_usuario "+
                                                "where dsc_password =
'"+PassDic+"' "+
                                                  "and dsc_username =
'"+UserDic+"'");
        if (ResultadoDic.next())
        {
          Cod_Usuario = ResultadoDic.getString(1);
        }
        if (Cod_Usuario == null)
        {
          FluxoSaida.println("Recusado");
        }
        else
        {
          FluxoSaida.println("Aceito");
        }
        FluxoSaida.flush();
        BancoDic.close();
      }
      else
      {
        FluxoSaida.println("String de Usuario/Senha invalido");
        FluxoSaida.flush();
      }
    }
```

```
    catch (Exception Erro)
    {
      FluxoSaida.println("Erro na validacao do Usuario:
"+Erro.getMessage());
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*
  void CarregaBanco()
  {
    Connection BancoDic     = null;
    Statement  EnviaSQLDic  = null;
    ResultSet  ResultadoDic = null;
    String     AuxCod_Banco = null;


    if (Cod_Usuario != null)
    {
      try
      {
        BancoDic = DriverManager.getConnection (StringConexaoDic,
UsuarioConexaoDic, SenhaConexaoDic);
        EnviaSQLDic = BancoDic.createStatement();
        ResultadoDic = EnviaSQLDic.executeQuery("select tb.cod_banco "+
                                                ",tb.dsc_alias "+

",tb.dsc_string_conexao "+
                                                ",tb.dsc_username "+
                                                ",tb.dsc_password "+
                                              "from tcc_banco tb,
tcc_usuario_banco tub "+
                                              "where tb.cod_banco =
tub.cod_banco "+
                                                "and tub.cod_usuario =
"+Cod_Usuario);
        VetorBanco = new Connection[20];
        VetorCod_Banco = new String[20];
        VetorDsc_Alias = new String[20];
        while (ResultadoDic.next())
        {
          AuxCod_Banco = ResultadoDic.getString(1);
          if (AuxCod_Banco.length() > 0)
          {
            ++QuantidadeConexoes;
            VetorCod_Banco[QuantidadeConexoes] = AuxCod_Banco;
            VetorDsc_Alias[QuantidadeConexoes] =
ResultadoDic.getString(2).toUpperCase();
            try
            {
              VetorBanco[QuantidadeConexoes] =
DriverManager.getConnection(ResultadoDic.getString(3)

,ResultadoDic.getString(4)

,ResultadoDic.getString(5));
            }
            catch (Exception Erro)
            {
```

```
              FluxoSaida.println("Erro ao Conectar com o Banco do Usuario:
"+Erro.getMessage());
            }
          }
        }
        BancoDic.close();
      }
      catch (Exception Erro)
      {
        FluxoSaida.println("Erro na conexao com o Dicionario para carregar
os Bancos : "+Erro.getMessage());
      }
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void RecebeExecutaEnvia()
  {
    try
    {
      if (Cod_Usuario != null)
      {
        Comando = FluxoEntrada.readLine();
        while (!Comando.toUpperCase().trim().equals("FIM"))
        {
          if (Comando.toUpperCase().startsWith("SELECT")
            ||Comando.toUpperCase().startsWith("INSERT")
            ||Comando.toUpperCase().startsWith("DELETE")
            ||Comando.toUpperCase().startsWith("UPDATE"))
          {
            VerificaPermissao();
          }
          else
          {
            FluxoSaida.println("Comando invalido.");
            FluxoSaida.println("FIM");
          }
          Comando = FluxoEntrada.readLine();
        }
        FluxoSaida.println("QUIT");
      }
    }
    catch (IOException Erro)
    {
      System.err.println("Erro ao receber o Comando. "+Erro.getMessage());
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void VerificaPermissao()
  {
    Connection BancoDic           = null;
    Statement  EnviaSQLDic        = null;
    ResultSet  ResultadoDic       = null;
    String     VetorTabela[ ];
    int        NumeroTabela       = 1;
    int        PosicaoPalavra;
    int        PosicaoEspaco;
    int        PosicaoVirgula;
```

```
int        IndiceAuxiliar;
int        NesteBanco = 9999;
String     Select;
String     Insert;
String     Update;
String     Delete;
String     TabelaApelido      = null;
String     ComplementoSelect  = "1 = 1 ";
String     Onde               = null;
boolean    AchouOnde          = false;
boolean    InsertMaisBanco    = false;
boolean    TabelaSemPermissao = false;

PosicaoPalavra = Comando.indexOf("@");
if (PosicaoPalavra != -1)
{
  Onde = Comando.substring(PosicaoPalavra+1).trim().toUpperCase();
  Comando = Comando.substring(0,PosicaoPalavra).trim();
}

if (Comando.toUpperCase().startsWith("SELECT"))
{
  Select = Comando.toUpperCase();
  PosicaoPalavra = Select.indexOf(" FROM ");
  Select = Select.substring(PosicaoPalavra+6).trim();
  PosicaoPalavra = Select.indexOf(" WHERE ");
  if (PosicaoPalavra != -1)
  {
    Select = Select.substring(0, PosicaoPalavra).trim();
  }
  PosicaoPalavra = Select.indexOf(" ORDER BY ");
  if (PosicaoPalavra != -1)
  {
    Select = Select.substring(0, PosicaoPalavra).trim();
  }
  TabelaApelido = Select;
}
else if (Comando.toUpperCase().startsWith("INSERT"))
{
  Insert = Comando.toUpperCase();
  PosicaoPalavra = Insert.indexOf(" INTO ");
  Insert = Insert.substring(PosicaoPalavra+6).trim();
  PosicaoPalavra = Insert.indexOf("(");
  if (PosicaoPalavra != -1)
  {
    Insert = Insert.substring(0, PosicaoPalavra).trim();
  }
  PosicaoPalavra = Insert.indexOf("VALUES");
  if (PosicaoPalavra != -1)
  {
    Insert = Insert.substring(0, PosicaoPalavra).trim();
  }
  TabelaApelido = Insert;
}
else if (Comando.toUpperCase().startsWith("DELETE"))
{
  Delete = Comando.toUpperCase();
  PosicaoPalavra = Delete.indexOf(" FROM ");
  Delete = Delete.substring(PosicaoPalavra+6).trim();
```

```
      PosicaoPalavra = Delete.indexOf(" WHERE ");
      if (PosicaoPalavra != -1)
      {
        Delete = Delete.substring(0, PosicaoPalavra).trim();
      }
      TabelaApelido = Delete;
    }
    else if (Comando.toUpperCase().startsWith("UPDATE"))
    {
      Update = Comando.toUpperCase();
      PosicaoPalavra = Update.indexOf(" ");
      Update = Update.substring(PosicaoPalavra+1).trim();
      PosicaoPalavra = Update.indexOf(" SET ");
      if (PosicaoPalavra != -1)
      {
        Update = Update.substring(0, PosicaoPalavra).trim();
      }
      TabelaApelido = Update;
    }
    VetorTabela = new String[20];
    PosicaoEspaco = TabelaApelido.indexOf(" ");
    PosicaoVirgula = TabelaApelido.indexOf(",");
    if (PosicaoEspaco != -1 || PosicaoVirgula != -1)
    {
      while (PosicaoEspaco != -1 || PosicaoVirgula != -1)
      {
        PosicaoEspaco = TabelaApelido.indexOf(" ");
        if (PosicaoEspaco != -1)
        {
          VetorTabela[NumeroTabela] = TabelaApelido.substring(0,
PosicaoEspaco).trim();
          TabelaApelido = TabelaApelido.substring(PosicaoEspaco).trim();
        }
        else
        {
          PosicaoVirgula = TabelaApelido.indexOf(",");
          if (PosicaoVirgula != -1)
          {
            VetorTabela[NumeroTabela] = TabelaApelido.substring(0,
PosicaoVirgula).trim();
            TabelaApelido = TabelaApelido.substring(PosicaoVirgula).trim();
          }
          else
          {
            VetorTabela[NumeroTabela] = TabelaApelido;
          }
        }
        PosicaoVirgula = TabelaApelido.indexOf(",");
        if (PosicaoVirgula != -1)
        {
          TabelaApelido = TabelaApelido.substring(PosicaoVirgula+1).trim();
        }
        ++NumeroTabela;
      }
    }
    else
    {
      VetorTabela[NumeroTabela] = TabelaApelido;
    }
```

```
    try
    {
      BancoDic = DriverManager.getConnection (StringConexaoDic,
UsuarioConexaoDic, SenhaConexaoDic);
      EnviaSQLDic = BancoDic.createStatement();
      if (Comando.toUpperCase().startsWith("SELECT"))
      {
        ComplementoSelect = "ind_select = 'S' ";
      }
      else if (Comando.toUpperCase().startsWith("INSERT"))
      {
        ComplementoSelect = "ind_insert = 'S' ";
      }
      else if (Comando.toUpperCase().startsWith("DELETE"))
      {
        ComplementoSelect = "ind_delete = 'S' ";
      }
      else if (Comando.toUpperCase().startsWith("UPDATE"))
      {
        ComplementoSelect = "ind_update = 'S' ";
      }

      NumeroTabela = 1;
      while ((VetorTabela[NumeroTabela] != null) && (!TabelaSemPermissao))
      {
        if (Onde == null)
        {
          IndiceAuxiliar = 1;
          while (VetorCod_Banco[IndiceAuxiliar] != null)
          {
            ResultadoDic = EnviaSQLDic.executeQuery("select '1' "+
                                                    "from tcc_tabela tab,
tcc_permissao per "+
                                                    "where tab.nom_tabela =
'"+VetorTabela[NumeroTabela]+"' "+
                                                      "and tab.cod_tabela =
per.cod_tabela "+
                                                      "and
"+ComplementoSelect+
                                                      "and per.cod_banco =
"+VetorCod_Banco[IndiceAuxiliar]+" "+
                                                      "and per.cod_usuario
= "+Cod_Usuario);
            if (!ResultadoDic.next())
            {
              TabelaSemPermissao = true;
            }
            if (Comando.toUpperCase().startsWith("INSERT"))
            {
              InsertMaisBanco = true;
            }
            ++IndiceAuxiliar;
          }
        }
        else if (Onde != null)
        {
          IndiceAuxiliar = 1;
          while ((VetorDsc_Alias[IndiceAuxiliar] != null) && (!AchouOnde))
```

```
          {
            if (VetorDsc_Alias[IndiceAuxiliar].toUpperCase().equals(Onde))
            {
              NesteBanco = IndiceAuxiliar;
              AchouOnde = true;
            }
            else
            {
              ++IndiceAuxiliar;
            }
          }
          if (AchouOnde)
          {
            ResultadoDic = EnviaSQLDic.executeQuery("select '1' "+
                                                    "from tcc_tabela tab,
tcc_permissao per "+
                                                    "where tab.nom_tabela =
'"+VetorTabela[NumeroTabela]+"' "+
                                                      "and tab.cod_tabela =
per.cod_tabela "+
                                                      "and
"+ComplementoSelect+
                                                      "and per.cod_banco =
"+VetorCod_Banco[NesteBanco]+" "+
                                                      "and per.cod_usuario
= "+Cod_Usuario);
            if (!ResultadoDic.next())
            {
              TabelaSemPermissao = true;
            }
          }
        }
        ++NumeroTabela;
      }
      BancoDic.close();
    }
    catch (Exception Erro)
    {
      FluxoSaida.println("Erro ao conectar no dicionario para verificar
Permissoes: "+Erro.getMessage());
    }

    if ((Onde == null) && (!TabelaSemPermissao) && (!InsertMaisBanco))
    {
      ExecutaComando(NesteBanco);
    }
    else if ((AchouOnde) && (!TabelaSemPermissao))
    {
      ExecutaComando(NesteBanco);
    }
    else if (InsertMaisBanco)
    {
      FluxoSaida.println("Insert nao pode ser executado em mais de um
Banco.");
      FluxoSaida.println("FIM");
      FluxoSaida.flush();
    }
    else if ((!AchouOnde) && (Onde != null))
    {
```

```
      FluxoSaida.println("Local designado para o Comando nao existe");
      FluxoSaida.println("FIM");
      FluxoSaida.flush();
    }
    else if (TabelaSemPermissao)
    {
      FluxoSaida.println("Usuario possui Restricao em alguma Tabela");
      FluxoSaida.println("FIM");
      FluxoSaida.flush();
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void ExecutaComando(int NesteBanco)
  {
    Connection Banco          = null;
    Statement  EnviaSQL       = null;
    int        IndiceAuxiliar = 1;
    boolean    AcabouExecutar = false;


    while (!AcabouExecutar)
    {
      if (NesteBanco != 9999)
      {
        IndiceAuxiliar = NesteBanco;
      }
      Banco = VetorBanco[IndiceAuxiliar];
      try
      {
        EnviaSQL = Banco.createStatement();
      }
      catch (Exception Erro)
      {
        FluxoSaida.println("Erro ao criar o Envia SQL.
"+Erro.getMessage());
      }
      try
      {
        if (Comando.toUpperCase().trim().startsWith("SELECT"))
        {
          EnviaResultadoSelect(EnviaSQL.executeQuery(Comando),
IndiceAuxiliar);
        }
        else
        {
          EnviaResultadoOutros(EnviaSQL.executeUpdate(Comando));
          if (Comando.toUpperCase().startsWith("INSERT"))
          {
            IndiceAuxiliar = QuantidadeConexoes;
          }
        }
      }
      catch (Exception Erro)
      {
        FluxoSaida.println("Erro ao executar o Comando.
"+Erro.getMessage());
        IndiceAuxiliar = QuantidadeConexoes;
      }
```

```
      if (NesteBanco != 9999)
      {
        IndiceAuxiliar = 9999;
      }
      else
      {
        ++IndiceAuxiliar;
      }
      if (IndiceAuxiliar > QuantidadeConexoes)
      {
        AcabouExecutar = true;
      }
    }
    FluxoSaida.println("FIM");
    FluxoSaida.flush();
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void EnviaResultadoSelect(ResultSet Resultado, int EnviarCabecalho)
  {
    try
    {
      ResultSetMetaData MetaDados;
      int               QuantidadeColunas;
      int               IndiceAuxiliar;
      int               TamanhoColuna;
      int               TamanhoResultadoColuna;
      String            LinhaResultado = null;
      String            ResultadoPreenchido;


      MetaDados = Resultado.getMetaData();
      QuantidadeColunas = MetaDados.getColumnCount();
      if (EnviarCabecalho == 1)
      {
        for (IndiceAuxiliar = 1; IndiceAuxiliar <= QuantidadeColunas;
IndiceAuxiliar++)
        {
          if (LinhaResultado != null)
          {
            LinhaResultado = LinhaResultado + " - ";
          }
          ResultadoPreenchido = MetaDados.getColumnLabel(IndiceAuxiliar);
          TamanhoColuna = MetaDados.getColumnDisplaySize(IndiceAuxiliar);
          if (ResultadoPreenchido.length() > TamanhoColuna)
          {
            ResultadoPreenchido =
ResultadoPreenchido.substring(0,TamanhoColuna);
          }
          while (ResultadoPreenchido.length() < TamanhoColuna)
          {
            ResultadoPreenchido = ResultadoPreenchido + " ";
          }
          if (LinhaResultado == null)
          {
            LinhaResultado = ResultadoPreenchido;
          }
          else
          {
```

```
            LinhaResultado = LinhaResultado + ResultadoPreenchido;
          }
        }
        FluxoSaida.println(LinhaResultado);
        FluxoSaida.flush();
      }
      while (Resultado.next())
      {
        LinhaResultado = null;
        for (IndiceAuxiliar = 1; IndiceAuxiliar <= QuantidadeColunas;
IndiceAuxiliar++)
        {
          if (LinhaResultado != null)
          {
            LinhaResultado = LinhaResultado + " - ";
          }
          ResultadoPreenchido = Resultado.getString(IndiceAuxiliar);
          TamanhoColuna = MetaDados.getColumnDisplaySize(IndiceAuxiliar);
          if (ResultadoPreenchido == null)
          {
            ResultadoPreenchido = " ";
          }
          while (ResultadoPreenchido.length() < TamanhoColuna)
          {
            ResultadoPreenchido = ResultadoPreenchido + " ";
          }
          if (LinhaResultado == null)
          {
            LinhaResultado = ResultadoPreenchido;
          }
          else
          {
            LinhaResultado = LinhaResultado + ResultadoPreenchido;
          }
        }
        FluxoSaida.println(LinhaResultado);
        FluxoSaida.flush();
      }
    }
    catch (Exception Erro)
    {
      System.err.println("Erro ao enviar o Resultado. "+Erro.getMessage());
    }

  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void EnviaResultadoOutros(int QuantidadeLinhasAfetadas)
  {
    try
    {
      FluxoSaida.println(QuantidadeLinhasAfetadas+" linhas foram
afetadas");
      FluxoSaida.flush();
    }
    catch (Exception Erro)
    {
      System.err.println("Erro ao encerrar a Conexao. "+Erro.getMessage());
    }
```

```
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  void EncerraConexao()
  {
    try
    {
      FluxoEntrada = new DataInputStream(Conexao.getInputStream());
      FluxoSaida = new PrintStream(Conexao.getOutputStream());
    }
    catch (Exception Erro)
    {
      System.err.println("Erro ao encerrar a Conexao. "+Erro.getMessage());
    }
  }

//*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
  public void run()
  {
    String Usuario;
    int    QuantidadeValidacao = 1;


    try
    {
      ConexaoCliente();
      while (Cod_Usuario == null && QuantidadeValidacao <= 3)
      {
        Usuario = FluxoEntrada.readLine();
        ValidaUsuario(Usuario);
        QuantidadeValidacao = QuantidadeValidacao + 1;
      }
      CarregaBanco();
      RecebeExecutaEnvia();
      EncerraConexao();
    }
    catch (Exception Erro)
    {
      FluxoSaida.println("Erro na funcao main: "+Erro.getMessage());
    }
  }
}
```

REFERÊNCIAS BIBLIOGRÁFICAS

[BOC1995] BOCKENSKI, Bárbara. **Implementando sistemas cliente/servidor de qualidade**. São Paulo : Makron Books, 1995.

[CAM1996] CAMPIONE, Mary; WLARATH, Kate. **The Java tutorial: oriented object programming for the internet**. São Paulo : Érica, 1996.

[CAS1985] CASANOVA, Marco Antônio. **Princípios de gerência de bancos de dados distribuídos**. Rio de Janeiro : Campus, 1985.

[CER1995] CERÍCOLA, Vincent O. **Banco de dados relacional e distribuído**. São Paulo : McGraw-Hill, 1995.

[COU1991] COUCEIRO, Luiz Antônio C. da Cunha. **Sistemas de gerência de banco de dados distribuídos**. Brasília : Livros Técnicos e Científicos, 1991.

[DAT1985] DATE, C. J. . **Banco de dados: fundamentos**. Rio de Janeiro : Campus 1985.

[DAT1988] DATE, C. J.. **Banco de dados: tópicos avançados**. Rio de Janeiro: Campus, 1988.

[DAT1989] DATE, C. J.. **Guia para o padrão SQL**. Rio de Janeiro : Campus, 1989.

[DAT1991] DATE, Christian J. **Introdução a sistemas de banco de dados**. Rio de Janeiro : Editora Campus, 1991.

[FUR1998] FURLAN, Jose Davi. **Modelagem de dados através da UML – Unified modeling language**. São Paulo : Makron Books, 1998.

[GAS1993] GASPARINI, L. Anteu e BARRELA, E. Francisco. **TCP/IP Solução para Conectividade**. São Paulo : Erica, 1993.

[GOS1996] GOSLING, James; JOY, Bill; STEELE, Guy. **The Java language specification**. Califórnia : Addison Wesley, 1996.

[HAC1993] HACKATHORN, Richard D.. **Conectividade de bancos de dados empresariais**. Rio de Janeiro : Infobook, 1993.

[HAM1998] HAMILTON, Graham; Catell, Rick; Mydene Fisher. **JDBC database acces with Java. A tutorial and annotated reference**. Massachusetts : Addisson-Wessley, 1998.

[HEL1997] HELLER, Philip; ROBERTS, Simon. **Java 1.1 developer's handbook**. Alameda, USA : SYBEX Inc., 1997.

[HUR1990] HURSCH, Carolyn J. **SQL linguagem de Consulta Estruturada**. Rio de Janeiro : LTC-Livros Técnicos e Cientifícos, 1990.

[JEP1997] JEPSON, Brian. **Programando banco de dados em Java**. São Paulo : Makron Books, 1997.

[KOR1995] KORT, Henry F. **Sistemas de banco de dados**. São Paulo : Makron Books, 1995.

[LEM1996] LEMARY, Laura. **Teach yourself Java in 21 days**. São Paulo : McGraw-Hill, 1996.

[MAN1999] MANNING, Michelle M. **Borland Jbuilder in 21 days**. Indianapolis : Sams Net, 1999.

[NAU1996] NAUGHTON, Patrick. **Dominando o Java. Guia autorizado da Sun Microsystems**. São Paulo : Makron Books, 1996.

[PER1996] PERRY, Paul J. **Creating cool web applets with Java**. Foster-City : IDG Books, 1996.

[RAM2000] RAMON, Fábio. **JDBC 2 – Acesso a banco de dados usando a linguagem Java**. Rio de Janeiro : Novatec, 2000.

[SYB1999] SYBASE. **http://www.sybase.com/products/entcon/dircon.html**. 1999.

[TAN1994] TANENBAUM, Andrew S. **Redes de Computadores**. Rio de Janeiro : Campus, 1994.

[THO1997] THOMAS, Robert M. **Introdução às redes locais**. São Paulo : Makron Books, 1997.