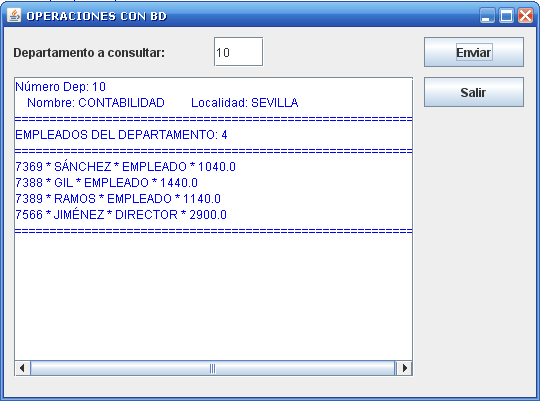
3.8.3. Consulta de base de datos.

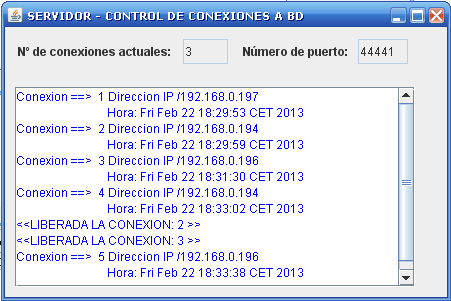
Una aplicación cliente-servidor típica es la consulta a bases de datos donde el cliente solicita una información y el servidor se la envía. En el siguiente ejemplo partimos de una base de datos **Db4o** que almacena objetos *Empleados* y *Departamentos*. Desde el programa cliente se introduce por medio de su pantalla el número de departamento que se desea consultar y se pulsa el botón *Enviar* para enviar la solicitud al servidor. El servidor le devolverá los datos solicitados para que los muestre en pantalla, véase Figura 3.18.



**Figura 3.18.** Cliente de BD.

El servidor, es muy similar al servidor de chat TCP creado anteriormente, espera las conexiones de los clientes y cuando le llega uno crea un hilo para satisfacer sus necesidades. El hilo devolverá al cliente la información solicitada. En este caso le devolverá un objeto *Departamentos*, ya que el cliente lo que solicita son los datos de un departamento.

El servidor en su pantalla muestra todas las conexiones que se van realizando, Figura 3.19. A cada cliente se le asignará un número de conexión. Cuando un cliente abandona la conexión también se mostrará un mensaje en la pantalla, además se muestra la IP, la hora y la fecha de la conexión. Esto es útil para llevar un registro de la actividad del servidor.



**Figura 3.19.** Servidor de BD.

Para manejar los datos de la base de datos disponemos de las clases **Departamentos** y **Empleados**. La clase **Departamentos** contiene los atributos *deptNo, dnombre, loc* y *empleadoses*, este último representa la colección de empleados del departamento. También se definen varios constructores y los métodos getter y setter correspondientes:

import java.util.HashSet;

import java.util.Set;

**public class Departamentos implements java.io.Serializable** {

private static final long serialVersionUID = 1L;

private int deptNo;

private String dnombre;

private String loc;

**private Set<Empleados> empleadoses** = new HashSet<Empleados>(0);

//constructores

public Departamentos() {}

public Departamentos(int deptNo) {this.deptNo = deptNo;}

public Departamentos(int deptNo, String dnombre, String loc,

Set<Empleados> empleadoses) {

this.deptNo = deptNo;

this.dnombre = dnombre;

this.loc = loc;

this.empleadoses = empleadoses;

}

public int getDeptNo() {return this.deptNo;}

public void setDeptNo(int i) {this.deptNo = i;}

public String getDnombre() {return this.dnombre;}

public void setDnombre(String dnombre) {this.dnombre = dnombre;}

public String getLoc() {return this.loc;}

public void setLoc(String loc) {this.loc = loc;}

public Set<Empleados> getEmpleadoses() {return this.empleadoses;}

public void setEmpleadoses(Set<Empleados> empleadoses) {

this.empleadoses = empleadoses;

}

}//..Departamentos

La clase **Empleados** define los atributos *empNo,apellido, oficio, dir, fechaAlt, salario, comision* y *departamentos*, este último es un objeto **Departamentos**. También se definen varios constructores y los métodos getter y setter correspondientes:

import java.util.Date;

**public class Empleados implements java.io.Serializable** {

private static final long serialVersionUID = 1L;

private int empNo;

**private Departamentos** departamentos;

private String apellido;

private String oficio;

private int dir;

private Date fechaAlt;

private Float salario;

private Float comision;

//constructores

public Empleados() {}

public Empleados(int empNo, Departamentos departamentos) {

this.empNo = empNo;

this.departamentos = departamentos;

}

public Empleados(int empNo, Departamentos departamentos,

String apellido, String oficio, int dir, Date fechaAlt,

Float salario, Float comision) {

this.empNo = empNo;

this.departamentos = departamentos;

this.apellido = apellido;

this.oficio = oficio;

this.dir = dir;

this.fechaAlt = fechaAlt;

this.salario = salario;

this.comision = comision;

}

public int getEmpNo() {return this.empNo;}

public void setEmpNo(int empNo) {this.empNo = empNo;}

public Departamentos getDepartamentos()

{return this.departamentos;}

public void setDepartamentos(Departamentos departamentos)

{this.departamentos = departamentos;}

public String getApellido() {return this.apellido;}

public void setApellido(String apellido)

{this.apellido = apellido;}

public String getOficio() {return this.oficio;}

public void setOficio(String oficio) {this.oficio = oficio;}

public int getDir() {return this.dir;}

public void setDir(int dir) {this.dir = dir;}

public Date getFechaAlt() {return this.fechaAlt;}

public void setFechaAlt(Date fechaAlt)

{this.fechaAlt = fechaAlt;}

public Float getSalario() {return this.salario;}

public void setSalario(Float salario) {this.salario = salario;}

public Float getComision() {return this.comision;}

public void setComision(Float comision)

{this.comision = comision;}

}//..

Utilizaremos la clase **Conexion** para obtener un objeto de base de datos (patrón *singleton*). El método *getDBConexion()* devuelve el objeto creado. La base de datos se llama **EMPLEDEP.YAP**.

import com.db4o.Db4oEmbedded;

import com.db4o.ObjectContainer;

**public class Conexion {**

final static String *BDPer* = "EMPLEDEP.YAP";

static ObjectContainer *db*;

static {

*db* = Db4oEmbedded.*openFile*(Db4oEmbedded.*newConfiguration*(), *BDPer*);

}

public static ObjectContainer **getDBConexion()** {

return *db*;

}

} // Fin Conexion

El **programa servidor** es similar al servidor TCP de chat. Primero se definen las variables y campos de pantalla:

import java.io.\*;

import java.net.\*;

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

**public class Servidor extends JFrame** {

private static final long serialVersionUID = 1L;

**static Integer PUERTO = 44441;**

public static Integer conexiones = 0;

**static ServerSocket servidor;**

static java.util.Date hora;

// campos de la pantalla

static public JTextField numconex = new JTextField();

static JLabel numconexLabel = new JLabel();

static JTextField puerto = new JTextField();

static JLabel puertoLabel = new JLabel();

static public JTextArea area = new JTextArea();

static JScrollPane scroll = new JScrollPane(area);

Desde el constructor se prepara la pantalla:

**// Constructor**

**public Servidor()** {

super("SERVIDOR - CONTROL DE CONEXIONES A BD");

Container c = getContentPane();

numconexLabel.setText("Nºde conexiones actuales:");

puertoLabel.setText("Número de puerto:");

numconexLabel.setBounds(new Rectangle(10, 10, 160, 25));

numconex.setBounds(new Rectangle(175, 10, 45, 25));

puertoLabel.setBounds(new Rectangle(235, 10, 200, 25));

puerto.setBounds(new Rectangle(350, 10, 50, 25));

area.setBounds(new Rectangle(10, 60, 390, 200));

scroll.setBounds(new Rectangle(10, 60, 400, 200));

area.setEditable(false);c.add(scroll, null);

c.add(numconexLabel);c.add(numconex);

numconex.setEditable(false);c.add(puertoLabel);

c.add(puerto);puerto.setEditable(false);

c.setLayout(null);

area.setForeground(Color.blue);area.setText("");

setSize(450, 300); //COLOCACIÓN DE LA PANTALLA

Dimension dim = getToolkit().getScreenSize();

setLocation(dim.width / 2 - getWidth() / 2 + 200,

(dim.height / 2 - getHeight() / 2) + 200);

setDefaultCloseOperation(JFrame.DISPOSE\_ON\_CLOSE);

También se define la acción a realizar cuando se cierra la ventana (en este ejemplo no hay un botón para cerrar el servidor). Al cerrar la ventana se cierra el servidor y la conexión a la base de datos:

**//CERRAMOS VENTANA**

**addWindowListener(new WindowListener()** {

public void windowClosing(WindowEvent we) {

try { // CERRAMOS EL SERVERSOCKET

**servidor.close();**

System.out.println("Servidor cerrado .....");

**Conexion.db.close()**;// cerrar BD

System.exit(0);

} catch (IOException e) {

System.err.println("NO SE PUEDE CERRAR servidor."

+ e.getMessage());

System.exit(0);

}

}

//resto de métodos no se implementan

public void windowOpened(WindowEvent we) {; }

public void windowClosed(WindowEvent we) {;}

public void windowIconified(WindowEvent we) {;}

public void windowDeiconified(WindowEvent we) {;}

public void windowActivated(WindowEvent we) {;}

public void windowDeactivated(WindowEvent we) {;}

});

}//.. fin del constructor

Desde *main()* se inicia el servidor y las variables, se visualiza el número de puerto por el que escucha y el número de conexiones actuales, y se prepara la pantalla (Figura 3.20):

//MAIN

**public static void main(String[] args) throws IOException** {

int idCliente = 0;//cada cliente tendra un id

**servidor = new ServerSocket(PUERTO)**;

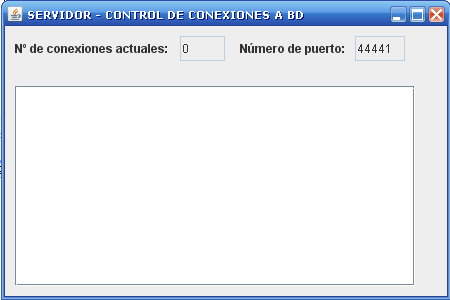
System.out.println("Servidor Iniciado .....");

Servidor pantalla = new Servidor();

pantalla.setVisible(true);

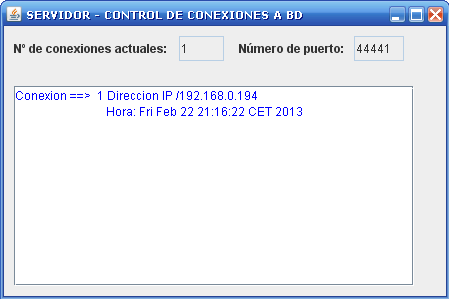
puerto.setText(PUERTO.toString());

numconex.setText(conexiones.toString());



**Figura 3.20.** Pantalla inicial del servidor.

Se hace un bucle para controlar las conexiones de los clientes. Dentro del bucle el servidor espera la conexión del cliente, cuando uno se conecta se calcula la hora, se incrementa el número de conexiones, se calcula el identificador para el cliente (empieza en 0 y se va incrementando de 1 en 1), se averigua su dirección IP, se muestra toda esta información en pantalla, véase Figura 3.21, y se crea un hilo de ejecución para ese cliente:



**Figura 3.21.** Pantalla del servidor cuando se conecta un cliente.

**while (true)** {

try {

**Socket cliente=servidor.accept(); //esperando al cliente**

hora = new java.util.Date(System.currentTimeMillis());

conexiones++;

idCliente++;

numconex.setText(conexiones.toString());

area.append("Conexion ==> " + idCliente);

**InetAddress direccion = cliente.getInetAddress();**

area.append(" Direccion IP " + direccion.toString()

+ "\n\t Hora: " + hora + "\n");

**HiloServidor hilo = new HiloServidor(cliente, idCliente);//** **hilo.start();** // Ejecutamos el hilo

} catch (IOException e) {

//ocurre cuando cerramos la ventana

//porque el servidor esta cerrado

System.out.println(e.getMessage());

System.exit(0);

}

}//while

} //main

} //..fin SERVIDOR

El hilo **HiloServidor** se encarga de recibir el número de departamento que el cliente quiere consultar, y enviar al cliente un objeto **Departamentos** con los datos solicitados. Necesita declarar un stream para la entrada **DataInputStream** y para la salida creará un **ObjectOutputStream** ya que devuelve un objeto al cliente. Desde el constructor se crean los flujos de entrada y salida del cliente:

import java.io.\*;

import java.net.\*;

import javax.swing.\*;

import java.awt.\*;

**class HiloServidor extends Thread** {

Socket socket;

int identificador;

static JLabel *texto* = new JLabel();

**ObjectOutputStream outObjeto;**

**DataInputStream entrada;**

//Constructor

**public HiloServidor(Socket s, int idCliente)** throws IOException {

socket = s;

identificador=idCliente;

**entrada = new DataInputStream(socket.getInputStream()); outObjeto = new ObjectOutputStream(socket.getOutputStream());**

}

//

En el método *run()* se lee del stream el departamento que el cliente desea consultar y se envía un objeto **Departamentos** con los datos solicitados. Para hacer esto se crea un objeto de la clase *AccesoDatos()*. Con este objeto llamamos al método *procesarCadena()* que recibe el departamento que escribió el cliente en su pantalla y devuelve un objeto **Departamentos** con los datos del departamento solicitado:

**public void run()** {

try {

**AccesoDatos adat = new AccesoDatos()**;

while (true) {

**String depar = entrada.readUTF();//leer stream**

**Departamentos dep = adat.procesarCadena(depar.trim())**; // Se envía el objeto al cliente

**outObjeto.writeObject(dep)**;

}

} catch (IOException e) {

**// cuando un cliente Cierra la conexión**

Servidor.conexiones--;

Servidor.numconex.setText(Servidor.conexiones.toString());

texto.setText("<<LIBERADA LA CONEXION: "+

identificador+" >>\n");

texto.setForeground(Color.red);

Servidor.area.append(texto.getText());

try {

entrada.close();

outObjeto.close();

socket.close();

} catch (IOException ee) {

ee.printStackTrace();

}

}//catch

} // de run()

}//..fin HiloServidor

Cuando el cliente cierra la conexión (pulsa el botón *Salir* de su pantalla) se produce *IOException*, entonces se decrementa el número de conexiones y se muestra en la pantalla del servidor un mensaje indicando que se ha liberado la conexión del cliente. Se cierran los flujos de entrada y salida y el socket.

La clase **AccesoDatos** es la que comunica con la base de datos. En el constructor se crea un objeto a la base de datos:

import com.db4o.ObjectContainer;

import com.db4o.ObjectSet;

//ACCESO A BD db4o

**public class AccesoDatos** {

static ObjectContainer db;

// Constructor

**public AccesoDatos()** {

db = Conexion.getDBConexion();

}

Dispone del método *procesarCadena()* que recibe en un String el departamento a consultar. Lo pasa a entero. Usa el método *queryByExample()* para obtener el departamento deseado, si lo encuentra devolverá un objeto **Departamentos** y si no existe devolverá *null*:

// Se procesa la cadena que manda el hilo con el dep a localizar

**synchronized Departamentos procesarCadena(String str)** {

int i;

Departamentos d = null;

try {

i = Integer.parseInt(str);

} catch (NumberFormatException n) {

System.out.println("<<DEPARTAMENTO: " + str + "

INCORRECTO>> ");

return d;

}

**Departamentos dep = new Departamentos(i, null, null, null);**

**ObjectSet<Departamentos> result = db.queryByExample(dep);**

if (result.size() == 0)

System.out.println("<<DEPARTAMENTO: " + i + " NO EXISTE>> ");

else {

d = result.next();

}

return d;// devuelve un objeto Departamentos

}//procarCadena

}//..fin AccesoDatos

El método *procesarCadena()* se define **synchronized** para evitar que dos hilos accedan simultáneamente al método. Sólo uno lo podrá utilizar, el otro tendrá que esperar a que el primero termine.

Por último nos queda ver el código del **programa cliente**. En este caso la clase **ClienteBD** implementa **Runnable**. Se definen las variables, campos de la pantalla y los streams de entrada y de salida, en este caso el stream de entrada es un **ObjectInputStream** porque el cliente recibe un objeto; y el stream de salida es un **DataOutputStream** por el que el cliente envía un String:

import javax.swing.\*;

import java.awt.event.\*;

import java.net.\*;

import java.util.\*;

import java.io.\*;

import java.awt.\*;

**public class ClienteBD extends JFrame implements**

**ActionListener, Runnable** {

private static final long serialVersionUID = 1L;

static JTextField depconsultar = new JTextField(2);

static JLabel etiqueta = new JLabel("Departamento a consultar:");

private JScrollPane scrollpane1;

static JTextArea textarea1;

JButton boton = new JButton("Enviar");

JButton desconectar = new JButton("Salir");

boolean repetir = true;

static Socket socket;

**//streams**

**ObjectInputStream inObjeto;**

**DataOutputStream salida;**

En el constructor se recibe el socket creado, se crean los flujos de entrada y de salida y se prepara la pantalla:

**// constructor**

**public ClienteBD(Socket s)** {

super("OPERACIONES CON BD");

socket = s;

try {

// flujo de salida – para enviar cadena

**salida = new DataOutputStream(socket.getOutputStream())**;

// flujo de entrada – para recibir objeto

**inObjeto = new ObjectInputStream(socket.getInputStream())**;

} catch (IOException e) {

e.printStackTrace();

System.exit(0);

}

setLayout(null);

etiqueta.setBounds(10, 10, 200, 30);add(etiqueta);

depconsultar.setBounds(210, 10, 50, 30);add(depconsultar);

textarea1 = new JTextArea();

scrollpane1 = new JScrollPane(textarea1);

scrollpane1.setBounds(10, 50, 400, 300); add(scrollpane1);

boton.setBounds(420, 10, 100, 30); add(boton);

desconectar.setBounds(420, 50, 100, 30); add(desconectar);

textarea1.setEditable(false);

boton.addActionListener(this);

desconectar.addActionListener(this);

setDefaultCloseOperation(JFrame.DO\_NOTHING\_ON\_CLOSE);

}// constructor

Cuando se pulsa el botón *Enviar* se envía eldepartamento tecleado por el stream de salida al hilo servidor:

**// acción cuando pulsamos botones**

**public void actionPerformed(ActionEvent e)** {

if (e.getSource() == boton) { // ENVIAR DEP

try {

**salida.writeUTF(depconsultar.getText());**

} catch (IOException e1) {

e1.printStackTrace();

}

}//if

Cuando se pulsa el botón *Salir* se cierra el socket y finaliza la ejecución del cliente

if (e.getSource() == desconectar) { // SALIR try {

**socket.close()**;

} catch (IOException e1) {

e1.printStackTrace();

}

System.exit(0);

}//if

}// actionPerformed

En el método *run()* se realiza un proceso repetitivo donde el cliente recibe del hilo el objeto **Departamentos** con los datos del departamento solicitado. Si es *null* se visualiza un mensaje indicando que no existe el departamento, véase Figura 3.22, si no lo es se visualizarán los datos. Primero se pintan los datos del departamento (número, nombre y localidad) y después se llama al método *PintarEmpleados()* para visualizar los datos de los empleados del departamento:

**// proceso repetitivo**

**public void run() {**

String texto = "";

**while (repetir) {**

try {

Departamentos d = null;

**d = (Departamentos) inObjeto.readObject(**);//recibo un objeto

textarea1.setText("");

textarea1.setForeground(Color.blue);

**if (d == null)** {

textarea1.setForeground(Color.red);

PintaMensaje(" <<EL DEPARTAMENTO NO EXISTE>>");

} else {

**//datos del departamento**

texto = "Número Dep: " + d.getDeptNo() + "\n "

+ " Nombre: " + d.getDnombre() + "\tLocalidad: "

+ d.getLoc();

textarea1.append(texto);

**PintarEmpleados(d)**;**// visualizar empleados**

}//else

} catch (SocketException s){

**repetir=false;**//se produce al cerrar socket en botón salir

} catch (IOException e) {

e.printStackTrace();

**repetir = false;**

} catch (ClassNotFoundException e) {

e.printStackTrace();

**repetir = false;**

}

**}//fin while**

try {

**socket.close();** // CERRAR SOCKET

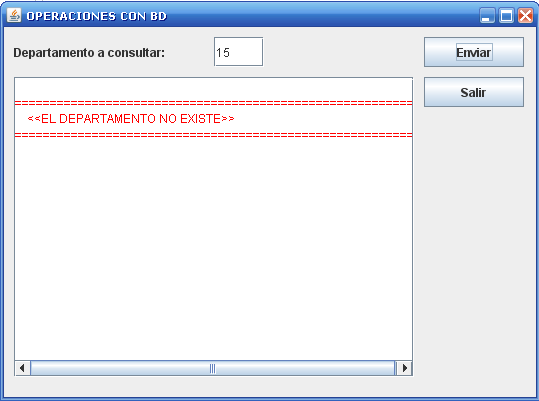
System.exit(0);

} catch (IOException e) {

e.printStackTrace();

}

}// fin run



**Figura 3.22.** Pantalla del cliente, no existe el departamento.

El proceso repetitivo finaliza cuando la variable *repetir* sea *false*, que puede ser cuando ocurra alguna excepción. Al final se cierra el socket.

El método *PintarEmpleados()* pinta los empleados del objeto **Departamentos** que recibe. Primero recupera la colección de empleados en un objeto Set y después los recorre mediante un iterador. Según va recuperando empleados visualiza su datos en el textarea:

**// PINTA LOS EMPLEADOS EN EL AREA**

**private void PintarEmpleados(Departamentos d)** {

**Set<Empleados> listaemple = d.getEmpleadoses();//** obtenemos

// empleados

textarea1.setForeground(Color.blue);

if (listaemple == null) {

**PintaMensaje**("EL DEPARTAMENTO NO TIENE EMPLEADOS");

} else {

**PintaMensaje**("EMPLEADOS DEL DEPARTAMENTO: " +

listaemple.size());

**Iterator<Empleados> it = listaemple.iterator();**

**while (it.hasNext())** {

Empleados emple = new Empleados();

emple = it.next();

textarea1.append("\n" + emple.getEmpNo() + " \* "

+ emple.getApellido() + " \* " + emple.getOficio()

+ " \* " + emple.getSalario());

}//while

textarea1.append("\n=====================================");

}//else

}//PintarEmpleados

El método *PintarMensaje()* visualiza en el textarea el mensaje que recibe:

**// PINTA CABECERAS**

**void PintaMensaje(String mensaje)** {

textarea1.append("\n==========================================");

textarea1.append("\n" + mensaje);

textarea1.append("\n==========================================");

}//PintaMensaje

Por último desde el método *main()* se realiza la conexión al servidor, se prepara la pantalla y se lanza el hilo cliente:

**// MAIN**

**public static void main(String args[]) throws UnknownHostException,**

**IOException** {

int puerto = 44441;

Socket s = new Socket("localhost", puerto);//máquina local

**ClienteBD hiloC = new ClienteBD(s);**

hiloC.setBounds(0, 0, 540, 400);

hiloC.setVisible(true);

**new Thread(hiloC).start();**

}// fin main

}// Fin del CLIENTE

Para poder ejecutar el servidor necesitamos incluir en el CLASSPATH la librería para poder usar la base de datos **Db4o**, en este ejemplo se ha usado *db4o-8.0.224.15975-core-java5.jar*. En la máquina donde instalemos el servidor necesitamos las clases: *Servidor, HiloServidor, Empleados, Departamentos, Conexion* y *AccesoDatos*; además del fichero JAR. En la máquina donde instalemos el cliente necesitamos las clases *ClienteBD, Empleados* y *Departamentos.*

Igual que en el ejemplo del chat TCP, cuando los programas cliente y el servidor estén en máquinas diferentes, al crear el socket en el programa cliente indicaremos la dirección IP donde está la máquina servidora, ejemplo:

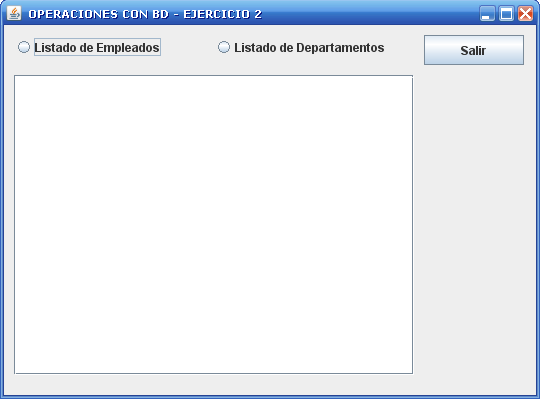
Socket s = new Socket("192.168.0.194", puerto);

ACTIVIDAD 3.7

Prueba los programas cliente y servidor desde diferentes máquinas. Las clases: *Servidor, HiloServidor, Empleados, Departamentos, Conexion*, *AccesoDatos* y el fichero JAR tienen que estar en la máquina servidora. Las clases *ClienteBD, Empleados* y *Departamentos* en las máquinas cliente.

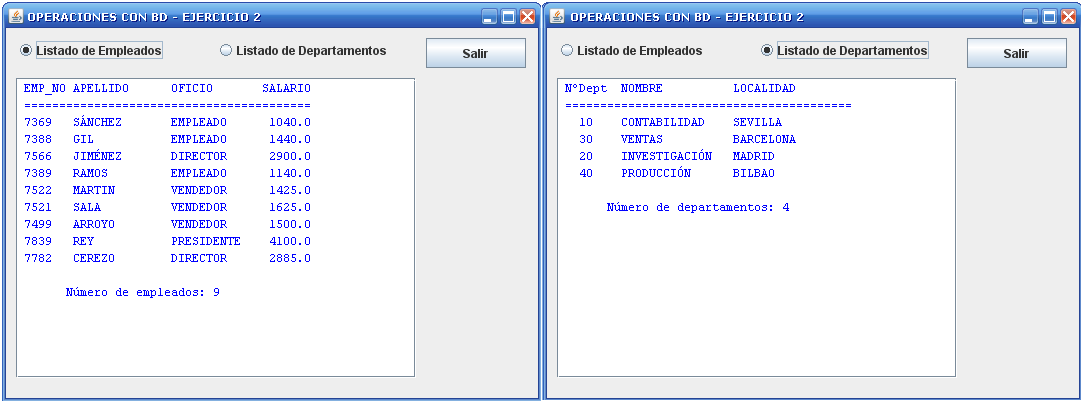
COMPRUEBA TU APRENDIZAJE

2º) Partimos del ejercicio que utiliza la base de datos **Db4o** EMPLEDEP.YAP. Realiza los cambios necesarios en el programa cliente, en el hilo servidor y en la clase de acceso a la base de datos. La pantalla inicial del programa cliente se muestra en la Figura 3.25.



**Figura 3.25.** Pantalla inicial del cliente Ejercicio 2.

* Desde esta pantalla el cliente puede seleccionar un listado de empleados o un listado de departamentos. Al pulsar en la primera opción (*Listado de Empleados*) se debe enviar al hilo servidor una cadena que indique que se desea un listado de empleados (por ejemplo la cadena EMP), si el listado seleccionado es el de departamentos la cadena será diferente (por ejemplo DEP).
* La cadena recibida en el hilo servidor será enviada a un método de la clase **AccesoDatos** (que tienes que crear) que nos devolverá en otra cadena los datos solicitados (el listado de empleados o de departamentos).
* Desde el hilo servidor se envía la cadena con los datos de consulta solicitados al programa cliente.
* El programa cliente muestra en el textarea los datos que le envía el hilo servidor, véase Figura 3.26.

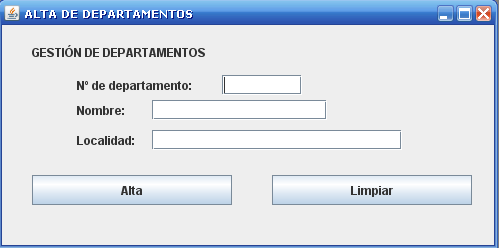
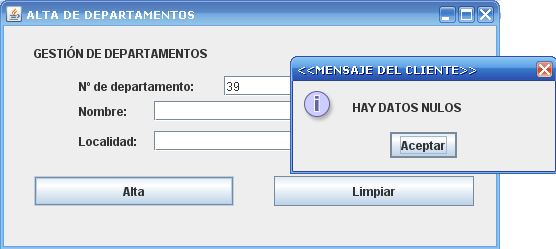


**Figura 3.26.** Consulta de empleados y departamentos Ejercicio 2.

* El botón *Salir* finaliza el programa cliente.

ACTIVIDADES DE AMPLIACIÓN

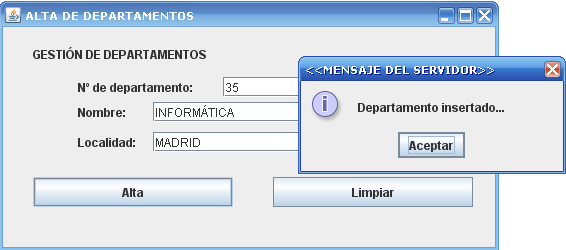
1º) Partiendo del ejemplo resuelto de consulta a la base de datos **Db4o** realiza un programa cliente para dar de alta departamentos. Al ejecutar el cliente se debe mostrar la pantalla de alta, Figura 3.29.

**Figura 3.29.** Pantalla de entrada de departamentos. **Figura 3.30.** Pantalla de error en la entrada de datos.

Cuando se pulse el botón de *Alta* se debe comprobar si el departamento es numérico y si los campos nombre y localidad tienen datos. Si ocurre algún error en la comprobación se debe visualizar una ventanita indicando el error, Figura 3.30.

Si todo está correcto el cliente envía un objeto **Departamentos** con los datos tecleados al hilo servidor para que realice la inserción del departamento en la base de datos. El hilo servidor debe mandar como respuesta al cliente los mensajes *Departamento insertado*, si se ha insertado el departamento o *Departamento existente*, si ya existe*.* El cliente recibe el mensaje y lo visualiza en una ventanita, Figura 3.31.



**Figura 3.31.** Pantalla de entrada de departamentos.

El botón *Limpiar*, limpia la pantalla. Al cerrar la ventana finaliza el cliente y se cierra el socket.

2º) Partiendo del ejemplo resuelto de consulta a la base de datos **Db4o** realiza un programa cliente que nos permita consultar los datos de un empleado. Se enviará al servidor el número de empleado a consultar, el hilo servidor devolverá el objeto empleado solicitado si existe o null si no existe.