**Generación de servicios en red**

**4**

# Ejercicios

### implementa una aplicación cliente-servidor que sirva documentos de texto. el servidor deberá almacenar documentos de texto, en ficheros *txt*. cuando un cliente se conecte al servidor, especificará el número de fichero que desea obtener y el servidor le enviará su contenido, carácter a carácter. usa para ello *sockets stream*.

**solución**

En la solución presentada, el servidor accede a ficheros ubicados en el directorio donde ha sido arrancado. Los ficheros deben estar nombrados siguiendo el patrón *id\_de\_fichero.txt*. El programa cliente recibe como argumento el identificador del fichero que se desea descargar, lo recibe e imprime su contenido por la salida estándar.

Código del servidor:

**import** java.io.BufferedReader;

**import** java.io.FileReader;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.InputStreamReader;

**import** java.io.OutputStream;

**import** java.net.InetSocketAddress;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**public class** Ej1Servidor {

**public static void procesarCliente**(Socket socket) {

### try {

InputStream is = socket.getInputStream(); OutputStream os = socket.getOutputStream();

BufferedReader br\_socket = **new** BufferedReader(**new** InputStreamReader(is)); String id\_fichero = br\_socket.readLine();

BufferedReader br\_fichero = **new** BufferedReader(**new** FileReader(id\_fichero+”.txt”));

**while** (br\_fichero.ready()) {

**char** b = (**char**) br\_fichero.read();

os.write(b);

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

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

### try {

ServerSocket serverSocket = **new** ServerSocket();

InetSocketAddress addr = **new** InetSocketAddress(“localhost”, 5555); serverSocket.bind(addr);

**while** (**true**) {

Socket newSocket = serverSocket.accept();

*procesarCliente*(newSocket);

newSocket.close();

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

Código del cliente:

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.InputStreamReader;

**import** java.io.OutputStream;

**import** java.io.PrintWriter;

**import** java.net.InetSocketAddress;

**import** java.net.Socket;

**public class** Ej1Cliente {

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

**if**(args.length != 1) {

System.out.println(“argumentos: <id del fichero>”);

System.*exit*(1);

}

### try {

String id\_fichero = args[0];

Socket clientSocket = **new** Socket();

InetSocketAddress addr = **new** InetSocketAddress(“localhost”, 5555);

clientSocket.connect(addr);

InputStream is = clientSocket.getInputStream();

OutputStream os = clientSocket.getOutputStream();

PrintWriter pw = **new** PrintWriter(os, **true**);

System.out.println(“Descargando fichero “+id\_fichero);

pw.println(id\_fichero);

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));

**char** buffer[] = **new char**[1];

br.read(buffer);

**while** (br.read(buffer) != -1) {

System.out.print(buffer[0]);

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

### implementa un servicio de identificación de direcciones iP usando *sockets datagram*. los clientes enviarán peticiones de identificación al servidor, que contestará con un mensaje que contenga la dirección iP del cliente como una cadena de texto.

**solución**

Código del programa servidor:

**import** java.io.IOException;

**import** java.net.DatagramPacket;

**import** java.net.DatagramSocket;

**import** java.net.InetSocketAddress;

**public class** Ej2Servidor {

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

### try {

InetSocketAddress addr = **new** InetSocketAddress(“localhost”, 5555); DatagramSocket datagramSocket = **new** DatagramSocket(addr);

**while** (**true**) {

**byte**[] buffer = **new byte**[25];

DatagramPacket datagrama1 = **new** DatagramPacket(buffer, buffer.length); datagramSocket.receive(datagrama1);

String respuesta = datagrama1.getAddress().toString(); DatagramPacket datagrama2 = **new** DatagramPacket(respuesta.getBytes(),

respuesta.getBytes().length, datagrama1.getAddress(), datagrama1.getPort()); datagramSocket.send(datagrama2);

}

} **catch** (IOException e) { e.printStackTrace();

}

}

}

Código del programa cliente:

**import** java.io.IOException; **import** java.net.DatagramPacket; **import** java.net.DatagramSocket; **import** java.net.InetAddress;

**public class** Ej2Cliente {

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

### try {

DatagramSocket datagramSocket = **new** DatagramSocket(); String token = “id”;

InetAddress addr1 = InetAddress.*getByName*(“localhost”); DatagramPacket datagrama1 = **new** DatagramPacket(token.getBytes(), token.getBytes().length, addr1, 5555); datagramSocket.send(datagrama1);

**byte**[] buffer = **new byte**[128];

DatagramPacket datagrama2 = **new** DatagramPacket(buffer, buffer.length); datagramSocket.receive(datagrama2);

String respuesta = **new** String(buffer); System.out.println(“Mi dirección es “+respuesta);

} **catch** (IOException e) { e.printStackTrace();

}

}

}

### se pretende crear una aplicación cliente-servidor que almacene una agenda de contactos. los clientes pueden conectarse al servidor y subir su información de contacto (nombre, dirección y teléfono), o modificarla si ya existe. especifica las funciones del servidor, escoge la tecnología de comunicaciones adecuada y diseña (de forma conceptual) el protocolo de nivel de aplicación de este servicio.

**solución**

Funciones del servidor:

n Almacenar una colección de contactos. Los clientes pueden conectarse para subir un contacto o descargar un contacto.

n Un contacto está formado por un nombre, una dirección y un teléfono.

n No es necesario mantener conexiones abiertas con el cliente, ya que la interacción con este es sencilla: envía un mensaje de petición y (a veces) recibe uno de respuesta.

n El servidor debe ser capaz de atender múltiples clientes simultáneamente.

Tecnología de comunicaciones:

n Este servicio se puede implementar usando *sockets datagram*, ya que solo requiere interacciones sencillas y no tiene requisitos de fiabilidad. Como alternativa avanzada, se podría implementar como un servicio web.

Protocolo de nivel de aplicación:

n Mensaje 1. Subida de un contacto. Se trata de la petición de subida de un contacto por parte de un cliente.

* **¿Quién lo envía?** El cliente.
* **¿cuándo se envía?** En cualquier momento, un cliente puede enviar este mensaje al servidor para realizar una petición.
* **¿Qué contiene?** Una cadena de texto con el nombre del contacto, otra con su dirección y otra con su teléfono.

n Mensaje 2. Consulta de un contacto. Se trata de la petición de consulta de un contacto por parte de un cliente.

* **¿Quién lo envía?** El cliente.
* **¿cuándo se envía?** En cualquier momento, un cliente puede enviar este mensaje al servidor para realizar una petición.
* **¿Qué contiene?** Una cadena de texto con el nombre del contacto.

n Mensaje 3. Respuesta de consulta. Se trata de la respuesta que envía el servidor al cliente, con los datos del contacto.

* **¿Quién lo envía?** El servidor.
* **¿cuándo se envía?** Como respuesta a un cliente que ha realizado una petición de consulta.
* **¿Qué contiene?** Una cadena de texto con la dirección y otra con el teléfono del contacto solicitado.

### implementa un servicio de publicación de mensajes. los clientes enviarán mensajes de texto al servidor, que los presentará por pantalla. utiliza para ello invocación de métodos remotos.

**solución**

Código de la interfaz del servidor:

**import** java.rmi.Remote;

**import** java.rmi.RemoteException;

**public interface** Ej4Interfaz **extends** Remote {

**void publicarMensaje**(String mensaje) **throws** RemoteException;

}

Código del programa servidor:

**import** java.rmi.RemoteException;

**import** java.rmi.registry.LocateRegistry;

**import** java.rmi.registry.Registry;

**import** java.rmi.server.UnicastRemoteObject;

**public class** Ej4Servidor **implements** Ej4Interfaz {

@Override

**public void publicarMensaje**(String mensaje) **throws** RemoteException { System.out.println(mensaje);

}

**public static void main**(String[] args) { Registry reg = **null**;

### try {

reg = LocateRegistry.*createRegistry*(5555);

} **catch** (Exception e) {

System.out.println(“ERROR: No se ha podido crear el registro”); e.printStackTrace();

}

Ej4Servidor serverObject = **new** Ej4Servidor();

0));

**try** {

reg.rebind(“Ej4”, (Ej4Interfaz) UnicastRemoteObject.*exportObject*(serverObject,

} **catch** (Exception e) {

System.out.println(“ERROR: No se ha podido inscribir el objeto servidor.”); e.printStackTrace();

}

}

}

Código del programa cliente:

**import** java.rmi.RemoteException;

**import** java.rmi.registry.LocateRegistry;

**import** java.rmi.registry.Registry;

**public class** Ej4Cliente {

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

**if**(args.length == 0) { System.out.println(“argumentos: <mensaje>”); System.*exit*(1);

}

String mensaje = args[0];

Ej4Interfaz serv = **null**; **try** {

Registry registry = LocateRegistry.*getRegistry*(“localhost”, 5555); serv = (Ej4Interfaz) registry.lookup(“Ej4”);

} **catch** (Exception e) { e.printStackTrace();

}

**if** (serv != **null**) {

### try {

serv.publicarMensaje(mensaje);

} **catch** (RemoteException e) { e.printStackTrace();

}

}

}

}

### implementa un servicio de control de estado de procesos remotos mediante *heartbeat*. la función de este servicio es controlar que sus clientes están activos, y detectar cuándo se han desactivado. Para ello, los clientes deben registrarse en el servidor cuando arrancan. una vez se han registrado, deben enviar un mensaje especial (latido del corazón o *heartbeat*) al servidor a intervalos de tiempo regulares (cada 10 segundos). si al cabo de 20 segundos el servidor no ha recibido ningún mensaje por parte de un cliente, lo considerará muerto y avisará por pantalla. implementa este servicio usando *sockets datagram*.

**solución**

En este ejercicio, el servidor debe almacenar la información de los clientes, contando el tiempo que ha pasado desde su último *heartbeat*. Para ello utiliza una tabla *hash*, representada por un objeto de clase *HashMap*. Además, el servidor contiene dos hilos de ejecución independientes. El hilo principal se encarga de escuchar por el *socket* y recibir los “latidos” de los clientes. El hilo secundario monitoriza la tabla de clientes, detectando cuando uno de estos ha muerto. Dado que ambos hilos acceden a un recurso compartido (la tabla de clientes), el programa implementa exclusión mutua en los accesos a esta.

Código del programa servidor:

**import** java.io.IOException; **import** java.net.DatagramPacket; **import** java.net.DatagramSocket;

**import** java.net.InetSocketAddress;

**import** java.util.Date;

**import** java.util.HashMap;

**public class** Ej5Servidor **extends** Thread {

**private** HashMap<String, Integer> tabla\_clientes = **new** HashMap<String, Integer>();

**private int** tiempo\_max\_espera = 20;

**private int** tiempo\_paso\_espera = 1;

**public synchronized void agregarCliente**(String id\_cliente) {

System.out.println((**new** Date()) + “ Heartbeat recibido de “ + id\_cliente); tabla\_clientes.put(id\_cliente, 0);

}

**public void run**() {

**while** (**true**) {

### try {

*sleep*(tiempo\_paso\_espera \* 1000);

### synchronized (this) {

String[] ids = tabla\_clientes.keySet().toArray(**new** String[0]);

**for** (String id\_cliente : ids) {

**int** tiempo\_cliente = tabla\_clientes.get(id\_cliente); tiempo\_cliente = tiempo\_cliente + tiempo\_paso\_espera;

**if** (tiempo\_cliente > tiempo\_max\_espera) {

System.out.println((**new** Date()) + “ El cliente “ + id\_cliente + “ ha muerto”); tabla\_clientes.remove(id\_cliente);

} **else** {

tabla\_clientes.put(id\_cliente, tiempo\_cliente);

}

}

}

} **catch** (InterruptedException e) { e.printStackTrace();

}

}

}

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

### try {

InetSocketAddress addr = **new** InetSocketAddress(“localhost”, 5555); DatagramSocket datagramSocket = **new** DatagramSocket(addr);

**byte**[] buffer = **new byte**[128];

DatagramPacket datagrama1 = **new** DatagramPacket(buffer, buffer.length);

Ej5Servidor serv = **new** Ej5Servidor(); serv.start();

**while** (**true**) { datagramSocket.receive(datagrama1);

serv.agregarCliente(**new** String(datagrama1.getData()));

}

} **catch** (IOException e) { e.printStackTrace();

}

}

}

Ejemplo de programa cliente:

**import** java.net.DatagramPacket; **import** java.net.DatagramSocket; **import** java.net.InetAddress;

**public class** Ej5Cliente {

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

**if**(args.length != 1) { System.out.println(“argumentos: <id del cliente>”); System.*exit*(1);

}

String id\_cliente = args[0];

### try {

DatagramSocket datagramSocket = **new** DatagramSocket(); InetAddress addr1 = InetAddress.*getByName*(“localhost”);

DatagramPacket datagrama1 = **new** DatagramPacket(id\_cliente.getBytes(), id\_ cliente.getBytes().length, addr1, 5555);

**while** (**true**) { datagramSocket.send(datagrama1); Thread.*sleep*(10\*1000);

}

} **catch** (Exception e) { e.printStackTrace();

}

}

}

### repite el ejercicio anterior, introduciendo las siguientes modificaciones:

n **deben emplearse las llamadas a métodos remotos como tecnología de comunicaciones.**

n **la duración del intervalo de tiempo del *heartbeat* se indicará como parámetro cuando el cliente se registre. de esta forma, el intervalo de un cliente puede ser de 10 s, otro de 5 s, otro de 20 s, etc.**

n **el tiempo que debe pasar antes de que el servidor declare muerto a un cliente será el doble del intervalo de *heartbeat* especificado.**

n **el servidor debe poder detectar “resurrecciones”, esto es, si un cliente que ha sido dado por muerto vuelve a enviar un mensaje de *heartbeat*, el servidor debe darse cuenta y notificarlo por su salida estándar.**

**solución**

Para esta versión, la solución incluye la definición de una clase privada *InfoCliente*, que almacena los datos de cada cliente (tiempo desde su último *heartbeat* y tiempo de actualización). El resto de la lógica del programa es similar al del ejercicio anterior, incluyendo la tabla de clientes y el hilo dedicado a su monitorización.

Código de la interfaz del servidor:

**import** java.rmi.Remote;

**import** java.rmi.RemoteException;

**public interface** Ej6Interfaz **extends** Remote {

**public void registro**(String id\_cliente, **int** intervalo\_heartbeat) **throws**

RemoteException;

**public void heartbeat**(String id\_cliente) **throws** RemoteException;

}

Código del programa servidor:

**import** java.rmi.RemoteException;

**import** java.rmi.registry.LocateRegistry;

**import** java.rmi.registry.Registry;

**import** java.rmi.server.UnicastRemoteObject;

**import** java.util.Date;

**import** java.util.HashMap;

**public class** Ej6Servidor **extends** Thread **implements** Ej6Interfaz {

**private class** InfoCliente {

**private int** tiempo\_cliente = 0; **private int** tiempo\_heartbeat = 0; **private boolean** muerto = **false**;

**public InfoCliente**(**int** tiempo\_cliente, **int** tiempo\_heartbeat) {

**this**.tiempo\_cliente = tiempo\_cliente;

**this**.tiempo\_heartbeat = tiempo\_heartbeat;

}

### public int getTiempoCliente() {

**return** tiempo\_cliente;

}

**public void setTiempoCliente**(**int** tiempo\_cliente) {

**this**.tiempo\_cliente = tiempo\_cliente;

}

**public void incrementarTiempoCliente**(**int** incremento) {

**this**.tiempo\_cliente += incremento;

}

### public int getTiempoHeartbeat() {

**return** tiempo\_heartbeat;

}

### public boolean estaMuerto() {

**return** muerto;

}

**public void setMuerto**(**boolean** muerto) {

**this**.muerto = muerto;

}

}

**private** HashMap<String, InfoCliente> tabla\_clientes = **new** HashMap<String, InfoCliente>();

**private int** tiempo\_paso\_espera = 1;

@Override

**public synchronized void registro**(String id\_cliente, **int** intervalo\_heartbeat)

**throws** RemoteException {

InfoCliente info = tabla\_clientes.get(id\_cliente);

**if** (info == **null**) {

System.out.println((**new** Date()) + “ Cliente “ + id\_cliente + “ registrado”); tabla\_clientes.put(id\_cliente, **new** InfoCliente(0, intervalo\_heartbeat));

} **else if** (info.estaMuerto()) {

System.out.println((**new** Date()) + “ El cliente “ + id\_cliente + “ ha resucitado”);

info.setTiempoCliente(0); info.setMuerto(**false**);

}

}

@Override

**public synchronized void heartbeat**(String id\_cliente) **throws** RemoteException { InfoCliente info = tabla\_clientes.get(id\_cliente);

**if** (info != **null**) {

**if** (info.estaMuerto()) {

System.out.println((**new** Date()) + “ El cliente “ + id\_cliente + “ ha resucitado”);

info.setMuerto(**false**);

} **else** {

System.out.println((**new** Date()) + “ Heartbeat recibido de “ + id\_cliente);

}

info.setTiempoCliente(0);

}

}

**public void run**() {

**while** (**true**) {

### try {

*sleep*(tiempo\_paso\_espera \* 1000);

### synchronized (this) {

String[] ids = tabla\_clientes.keySet().toArray(**new** String[0]);

**for** (String id\_cliente : ids) {

InfoCliente info\_cliente = tabla\_clientes.get(id\_cliente);

**if**(!info\_cliente.estaMuerto()) { info\_cliente.incrementarTiempoCliente(tiempo\_paso\_espera);

\* 2)) {

**if** (info\_cliente.getTiempoCliente() > (info\_cliente.getTiempoHeartbeat()

ha muerto”);

}

}

}

System.out.println((**new** Date()) + “ El cliente “ + id\_cliente + “

info\_cliente.setMuerto(**true**);

}

} **catch** (InterruptedException e) { e.printStackTrace();

}

}

}

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

Registry reg = **null**; **try** {

reg = LocateRegistry.*createRegistry*(5555);

} **catch** (Exception e) {

System.out.println(“ERROR: No se ha podido crear el registro”); e.printStackTrace();

}

Ej6Servidor serverObject = **new** Ej6Servidor();

### try {

reg.rebind(“Ej6”, (Ej6Interfaz) UnicastRemoteObject.*exportObject*(serverObject, 0));

} **catch** (Exception e) {

System.out.println(“ERROR: No se ha podido inscribir el objeto servidor.”); e.printStackTrace();

}

serverObject.start();

}

}

Ejemplo de programa cliente:

**import** java.rmi.registry.LocateRegistry;

**import** java.rmi.registry.Registry;

**public class** Ej6Cliente {

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

**if** (args.length != 2) {

System.out.println(“argumentos: <id cliente> <tiempo heartbeat>”); System.*exit*(1);

}

String id\_cliente = args[0];

**int** tiempo\_heartbeat = Integer.*parseInt*(args[1]);

Ej6Interfaz serv = **null**; **try** {

Registry registry = LocateRegistry.*getRegistry*(“localhost”, 5555); serv = (Ej6Interfaz) registry.lookup(“Ej6”);

} **catch** (Exception e) { e.printStackTrace();

}

**if** (serv != **null**) {

### try {

serv.registro(id\_cliente, tiempo\_heartbeat);

**while** (**true**) { serv.heartbeat(id\_cliente); Thread.*sleep*(tiempo\_heartbeat\*1000);

}

} **catch** (Exception e) { e.printStackTrace();

}

}

}

}

### implementa un servicio de “operador telefónico”. un “operador telefónico” es un servicio que mantiene comunicación con un grupo de clientes, y que ayuda a que los clientes puedan comunicarse entre sí sin necesidad de conocer su dirección iP. Para ello, cuando un cliente se conecta al servidor indica su identificador personal. después, cuando envía un mensaje, indica además el identificador personal del cliente al que va destinado. el servidor lo recoge y lo hace llegar al destinatario correcto. implementa este servicio usando *sockets stream*.

**solución**

En la solución de este ejercicio se utilizan dos elementos fundamentales. El primero es un hilo dedicado para cada cliente, que escucha por un *socket* específico conectado a un cliente concreto. El segundo es una tabla de clientes, que almacena el *socket* de cada cliente. Cuando el hilo de un cliente recibe un mensaje, busca el *socket* del destinatario en la tabla y reenvía el mensaje a su destino. Para probar el servicio se han implementado además tres clientes de prueba.

Código del programa servidor:

**import** java.io.BufferedReader; **import** java.io.IOException; **import** java.io.InputStreamReader; **import** java.io.PrintWriter;

**import** java.net.InetSocketAddress; **import** java.net.ServerSocket; **import** java.net.Socket;

**import** java.util.HashMap;

**public class** Ej7Servidor **extends** Thread {

HashMap<String, Socket> tabla\_clientes; Socket socket\_cliente;

**public Ej7Servidor**(HashMap<String, Socket> tabla\_clientes, Socket socket\_cliente)

{

**this**.tabla\_clientes = tabla\_clientes;

**this**.socket\_cliente = socket\_cliente;

}

**public void run**() {

### try {

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(socket\_cliente. getInputStream()));

String id\_cliente = br.readLine();

**synchronized**(tabla\_clientes) {

System.out.println(“Conectando con un nuevo cliente: “+id\_cliente); tabla\_clientes.put(id\_cliente, socket\_cliente);

}

String id\_destino = br.readLine();

**while** (id\_destino != **null**) { String mensaje = br.readLine(); **synchronized**(tabla\_clientes) {

Socket socket\_destino = tabla\_clientes.get(id\_destino);

**if** (socket\_destino != **null**) {

System.out.println(“Retransmitiendo mensaje de “+id\_cliente+” a “+id\_

destino);

}

}

PrintWriter pw = **new** PrintWriter(socket\_destino.getOutputStream(), **true**); pw.println(id\_cliente+”: “+mensaje);

id\_destino = br.readLine();

}

**synchronized**(tabla\_clientes) {

System.out.println(“El cliente “+id\_cliente+” se ha desconectado”); tabla\_clientes.remove(id\_cliente);

}

} **catch** (IOException e) { e.printStackTrace();

}

}

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

### try {

ServerSocket serverSocket = **new** ServerSocket();

InetSocketAddress addr = **new** InetSocketAddress(“localhost”, 5555); serverSocket.bind(addr);

HashMap<String, Socket> tabla\_clientes = **new** HashMap<String, Socket>();

**while** (**true**) {

Socket newSocket = serverSocket.accept();

Ej7Servidor nuevo\_thread = **new** Ej7Servidor(tabla\_clientes, newSocket); nuevo\_thread.start();

}

} **catch** (IOException e) { e.printStackTrace();

}

}

}

Ejemplo de programa cliente 1:

**import** java.io.BufferedReader; **import** java.io.IOException; **import** java.io.InputStreamReader; **import** java.io.PrintWriter;

**import** java.net.InetSocketAddress;

**import** java.net.Socket;

**public class** Ej7Cliente1 {

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

### try {

Socket clientSocket = **new** Socket();

InetSocketAddress addr = **new** InetSocketAddress(“localhost”, 5555); clientSocket.connect(addr);

PrintWriter pw = **new** PrintWriter(clientSocket.getOutputStream(), **true**); BufferedReader br = **new** BufferedReader(**new** InputStreamReader(clientSocket.

getInputStream()));

// Enviando su identificador pw.println(“cliente1”);

**while** (clientSocket.isConnected()) System.out.println(br.readLine());

clientSocket.close();

} **catch** (IOException e) { e.printStackTrace();

}

}

}

Ejemplo de programa cliente 2:

**import** java.io.BufferedReader; **import** java.io.IOException; **import** java.io.InputStreamReader; **import** java.io.PrintWriter;

**import** java.net.InetSocketAddress;

**import** java.net.Socket;

**public class** Ej7Cliente2 {

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

### try {

Socket clientSocket = **new** Socket();

InetSocketAddress addr = **new** InetSocketAddress(“localhost”, 5555); clientSocket.connect(addr);

PrintWriter pw = **new** PrintWriter(clientSocket.getOutputStream(), **true**); BufferedReader br = **new** BufferedReader(**new** InputStreamReader(clientSocket.

getInputStream()));

// Enviando su identificador pw.println(“cliente2”);

// Enviando un mensaje para el cliente 1 pw.println(“cliente1”); pw.println(“mensaje de prueba 1”);

**while** (clientSocket.isConnected()) System.out.println(br.readLine());

clientSocket.close();

} **catch** (IOException e) { e.printStackTrace();

}

}

}

Ejemplo de programa cliente 3:

**import** java.io.IOException;

**import** java.io.PrintWriter;

**import** java.net.InetSocketAddress;

**import** java.net.Socket;

**public class** Ej7Cliente3 {

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

### try {

Socket clientSocket = **new** Socket();

InetSocketAddress addr = **new** InetSocketAddress(“localhost”, 5555); clientSocket.connect(addr);

PrintWriter pw = **new** PrintWriter(clientSocket.getOutputStream(), **true**);

// Enviando su identificador pw.println(“cliente3”);

// Enviando un mensaje para el cliente 2 pw.println(“cliente2”); pw.println(“mensaje de prueba 2”);

// Enviando un mensaje para el cliente 1 pw.println(“cliente1”); pw.println(“mensaje de prueba 3”);

clientSocket.close();

} **catch** (IOException e) { e.printStackTrace();

}

}

}

# solución del test de conocimientos

## **Tabla 4.2** Soluciones del test de conocimientos del capítulo 4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1: | b | 6: | b |  |
| 2: c | | 7: b | |
| 3: d | | 8: c | |
| 4: c | | 9: d | |
| 5: d | | 10: c | |

**solución del caso Práctico**

### enunciado

Se desea programar una aplicación distribuida de buzón, siguiendo el modelo cliente-servidor. La idea consiste en que los clientes se conecten a un servidor, indicando su nombre. Una vez indicado su nombre, el cliente podrá consultar si tiene mensajes para él/ella, o dejar mensajes para otros usuarios. El servidor almacenará los mensajes entregados hasta que el usuario al que van dirigidos los consulte. Se pide:

n Especificar las funciones del servidor.

n Seleccionar la tecnología de comunicación adecuada (*sockets stream*, *sockets datagram*, llamada a métodos remotos, etc.).

n Diseñar el protocolo de nivel de aplicación.

n Implementar en java el servidor.

n Implementar en java un programa cliente que sirva para que diferentes usuarios consulten sus mensajes y dejen mensajes nuevos en el buzón.

Para simplificar el problema, se podrá asumir que los usuarios se identifican solo usando su nombre. No es necesario implementar gestión de contraseñas u otros mecanismos de seguridad.

### solución

*Funciones del servidor*:

n La función básica del servidor será almacenar mensajes de los diferentes usuarios. Cuando un cliente se conecte, deberá indicar su identificador (nombre de usuario) y la operación que desea realizar.

n Las operaciones que puede realizar un cliente son dos: enviar mensajes o consultar mensajes. Para enviar mensajes se debe proporcionar el mensaje. Al consultar los mensajes, el servidor devolverá al cliente todos los mensajes que tenga almacenados, destinados a este.

n Un mensaje está formado por un identificador de remitente, uno de destinatario y una cadena de texto con el mensaje propiamente dicho.

n No es necesario que el servidor mantenga una conexión abierta con cada cliente. Cada operación puede realizarse en una conexión independiente.

n La interacción con el cliente es sencilla: petición → respuesta.

n El servidor debe ser capaz de atender a múltiples clientes a la vez.

*Tecnología de comunicaciones*:

La tecnología elegida es la llamada a métodos remotos. Esto facilita la programación y permite representar de manera sencilla la interacción entre servidor y clientes.

*Protocolo de nivel de aplicación*:

n Mensaje 1. Envío de mensaje. Se trata de la petición de envío de mensaje por parte de un cliente.

* **¿Quién lo envía?** El cliente.
* **¿cuándo se envía?** En cualquier momento, un cliente puede enviar este mensaje al servidor para realizar una petición.
* **¿Qué contiene?** Un mensaje, formado por una cadena de texto con el remitente (el cliente), otra con el destinatario y otra con el texto del mensaje.

n Mensaje 2. Petición de lista de mensajes. Se trata de la petición de mensajes almacenados por parte de un cliente.

* **¿Quién lo envía?** El cliente.
* **¿cuándo se envía?** En cualquier momento, un cliente puede enviar este mensaje al servidor para realizar una petición.
* **¿Qué contiene?** Una cadena de texto con el nombre del cliente.

n Mensaje 3. Respuesta de lista de mensajes. Se trata de la respuesta que envía el servidor al cliente, con sus mensajes almacenados.

* **¿Quién lo envía?** El servidor.
* **¿cuándo se envía?** Como respuesta a un cliente que ha realizado una petición de lista de mensajes.
* **¿Qué contiene?** Una lista de mensajes, cada uno formado por las tres cadenas de texto mencionadas antes.

*Implementación del servidor*:

El servidor se divide en varios ficheros. Por un lado está la clase pública *Mensaje*, que representa un mensaje. Contiene información sobre remitente, destinatario, texto del mensaje y fecha de recepción (esta última no figura en el enunciado pero se ha añadido para enriquecer la solución). Esta clase debe implementar la interfaz *Serializable* para que pueda ser enviada como parámetro de los mensajes RMI. Por otro lado está la interfaz RMI que implementa el servidor. Por último, el programa servidor propiamente dicho.

Clase *Mensaje*:

**import** java.io.Serializable;

**import** java.util.Date;

**public class** Mensaje **implements** Serializable { **private static final long** serialVersionUID = 1L; **private** String remitente;

**private** String destinatario; **private** String mensaje; **private** Date fecha;

**public Mensaje**(String remitente, String destinatario, String mensaje) {

**this**.remitente = remitente; **this**.destinatario = destinatario; **this**.mensaje = mensaje;

}

**public** String **getRemitente**() {

**return** remitente;

}

**public void setRemitente**(String remitente) {

**this**.remitente = remitente;

}

**public** String **getDestinatario**() {

**return** destinatario;

}

**public void setDestinatario**(String destinatario) {

**this**.destinatario = destinatario;

}

**public** String **getMensaje**() {

**return** mensaje;

}

**public void setMensaje**(String mensaje) {

**this**.mensaje = mensaje;

}

**public** Date **getFecha**() {

**return** fecha;

}

**public void setFecha**(Date fecha) {

**this**.fecha = fecha;

}

@Override

**public** String **toString**() {

**return** (“De: “+remitente+”\n”+ “Para: “+destinatario+”\n”+ “Fecha: “+fecha+”\n”+ “Mensaje:\n”+mensaje);

}

}

Interfaz del servidor:

**import** java.rmi.Remote;

**import** java.rmi.RemoteException;

**public interface** BuzonInterfaz **extends** Remote {

**public void publicar**(String nombre\_usuario, Mensaje mensaje) **throws** RemoteException;

**public** Mensaje[] **obtenerMensajes**(String nombre\_usuario) **throws** RemoteException;

}

Programa servidor:

**import** java.rmi.RemoteException;

**import** java.rmi.registry.LocateRegistry;

**import** java.rmi.registry.Registry;

**import** java.rmi.server.UnicastRemoteObject;

**import** java.util.Date; **import** java.util.ArrayList; **import** java.util.HashMap;

**public class** BuzonServidor **implements** BuzonInterfaz { HashMap<String, ArrayList<Mensaje>> tabla\_mensajes;

### public BuzonServidor() {

tabla\_mensajes = **new** HashMap<String, ArrayList<Mensaje>>();

}

@Override

**public synchronized void publicar**(String nombre\_usuario, Mensaje mensaje) **throws**

RemoteException {

ArrayList<Mensaje> mensajes = tabla\_mensajes.get(mensaje.getDestinatario());

**if** (mensajes == **null**) {

mensajes = **new** ArrayList<Mensaje>(); tabla\_mensajes.put(mensaje.getDestinatario(), mensajes);

}

mensaje.setFecha(**new** Date()); mensajes.add(mensaje);

}

@Override

**public synchronized** Mensaje[] **obtenerMensajes**(String nombre\_usuario) **throws**

RemoteException {

Mensaje[] resultado = **null**;

ArrayList<Mensaje> mensajes = tabla\_mensajes.get(nombre\_usuario);

**if** (mensajes != **null**) {

resultado = mensajes.toArray(**new** Mensaje[0]);

}

**return** resultado;

}

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

Registry reg = **null**; **try** {

reg = LocateRegistry.*createRegistry*(5555);

} **catch** (Exception e) {

System.out.println(“ERROR: No se ha podido crear el registro”);

e.printStackTrace();

}

BuzonServidor serverObject = **new** BuzonServidor();

### try {

reg.rebind(“Buzón”, (BuzonInterfaz) UnicastRemoteObject.*exportObject*(serverObject, 0));

} **catch** (Exception e) {

System.out.println(“ERROR: No se ha podido inscribir el objeto servidor.”); e.printStackTrace();

}

}

}

*Implementación del cliente*:

El cliente es un programa pensado para ser ejecutado por línea de mandatos. Cuando esto sucede, se deben proporcionar una serie de parámetros de entrada. Si se ejecuta sin ellos, muestra la siguiente salida:

argumentos: <nombre usuario> <operación> [<destinataro> <texto mensaje>] El campo ‘operación’ puede tomar los valores ‘mensaje’ o ‘consulta’. Si se especifica un ‘mensaje’ se debe indicar el destinatario y el texto del mensaje. La operación ‘consulta’ muestra los mensajes que el servidor almacena para el usuario.

Dependiendo de los parámetros con los que sea invocado, el programa cliente realizará el envío de un mensaje o la consulta de los mensajes almacenados.

Código del programa cliente:

**import** java.rmi.registry.LocateRegistry;

**import** java.rmi.registry.Registry;

**public class** BuzonCliente {

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

**if** (args.length < 2) {

System.out.println(“argumentos: <nombre usuario> <operacion> [<destinataro>

<texto mensaje>]”);

System.out.println(“ El campo \”operacion\” puede tomar los valores ‘mensaje’ o”); System.out.println(“ ‘consulta’. Si se especifica un ‘mensaje’ se debe indicar el”); System.out.println(“ destinatario y el texto del mensaje. La operación ‘consulta’”); System.out.println(“ muestra los mensajes que el servidor almacena para el usuario.”); System.*exit*(1);

}

String nombre\_usuario = args[0]; String operacion = args[1];

BuzonInterfaz serv = **null**; **try** {

Registry registry = LocateRegistry.*getRegistry*(“localhost”, 5555); serv = (BuzonInterfaz) registry.lookup(“Buzon”);

} **catch** (Exception e) { e.printStackTrace();

}

**if** (serv != **null**) {

### try {

**if** (operacion.equals(“mensaje”)) {

**if** (args.length < 4) {

System.out.println(“ERROR: Debe especificar el destinatario y el

mensaje.”);

}

System.*exit*(1);

String destinatario = args[2]; String texto\_mensaje = args[3];

Mensaje mensaje = **new** Mensaje(nombre\_usuario, destinatario, texto\_mensaje); serv.publicar(nombre\_usuario, mensaje);

} **else if** (operacion.equals(“consulta”)) {

Mensaje[] mensajes = serv.obtenerMensajes(nombre\_usuario);

**if** (mensajes != **null**) {

**for** (Mensaje mensaje : mensajes) { System.out.println(“------------------”); System.out.println(mensaje);

}

}

} **else** {

System.out.println(“ERROR: Operacion desconocida”);

}

} **catch** (Exception e) { e.printStackTrace();

}

}

}

}