

## Índice

Introducción.....	2
Arquitectura.....	2
Pruebas de funcionamiento.....	5
Estructuras de las clases.....	7
Código fuente.....	8
Paquete Cliente.....	8
ProgramaCliente (clase):.....	8
Cliente (interfaz):.....	10
ClienteRMI (clase):.....	10
ClienteRPC (clase):.....	11
ClienteSocket (clase):.....	12
Paquete Común.....	14
OperacionesRMI_Interface (interfaz):.....	14
Paquete Servidor.....	14
ProgramaServidor (clase):.....	14
BaseDatos (clase):.....	15
ServidorSocket (clase):.....	17
Manejador (clase):.....	18
ServidorRPC (clase):.....	19
OperacionesRPC (clase):.....	19
ServidorRMI (clase):.....	20
OperacionesRMI (clase):.....	20

## Introducción

En el presente documento se explica cómo es que se estructuró nuestro proyecto, así como la arquitectura con la que cuenta y finalmente el código fuente de este. En términos generales se tiene un Sistema Distribuido basado en la arquitectura cliente-servidor. Tres servidores (Socket, RMI y RPC) atienden a múltiples clientes y es este último el que decide con qué tipo de servidor desea interactuar. A continuación, se presenta una explicación más a fondo.

## Arquitectura

La arquitectura general del sistema desarrollado es Cliente-Servidor. Múltiples clientes son atendidos por 3 servidores uno implementado con Socket, otro con RPC y un tercero con RMI. Además, se cuenta con una componente del lado del servidor llamada *BaseDatos*, la cual se encarga de cargar todos los datos que nos interesan para que los servidores puedan comunicarse con esta. Se sigue una arquitectura **basada en capas**, de acuerdo con la figura 1.0:

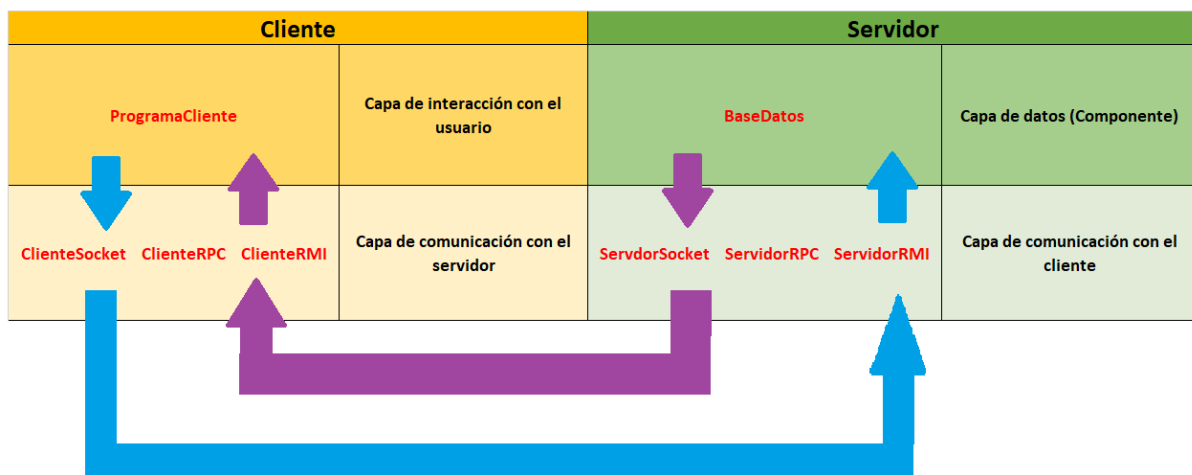


Figura 1.0. Diagrama que representa la arquitectura de nuestro proyecto implementado.

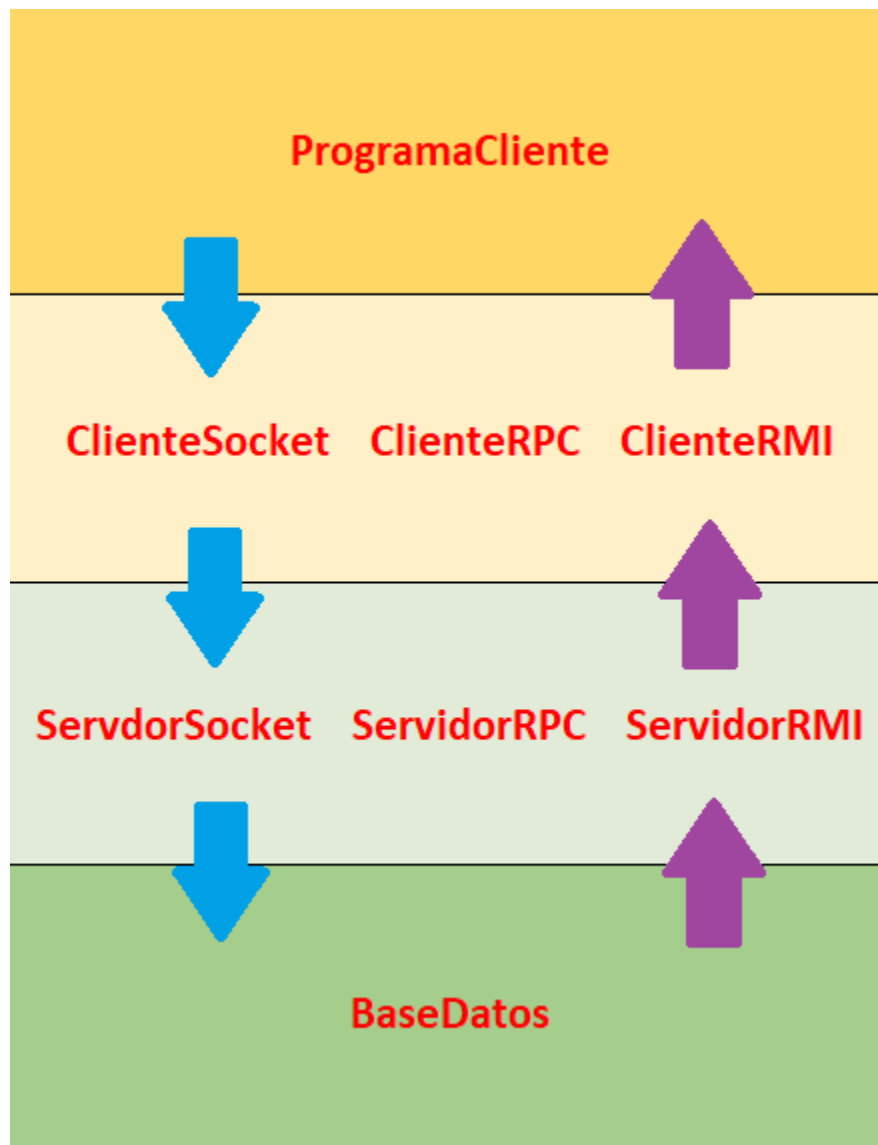


Figura 1.1. Otro punto de vista de nuestra arquitectura basada en capas

Explicando un poco más a fondo la arquitectura, el texto en rojo representa las clases implementadas que tienen que ver con la arquitectura del sistema (tanto en el diagrama, como en el texto a continuación). Para que la comunicación pueda llevarse a cabo, es necesario primero encender los servidores, esto se hace mediante la clase **ProgramaServidor**, que no se muestra en el diagrama debido a que precisamente solo tiene la función de iniciar los 3 servidores; no se contempló

el uso de un cuarto servidor para esta tarea debido a que resultaba mas factible tener en todo momento los servidores encendidos.

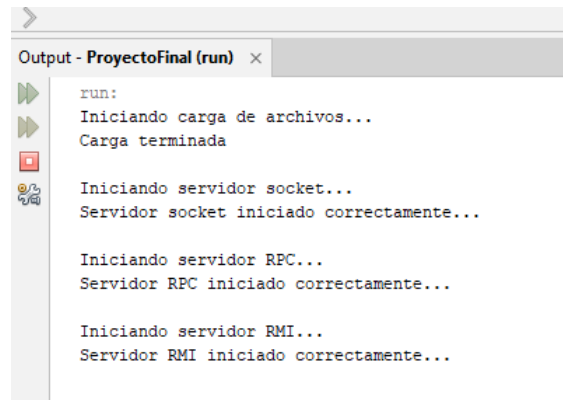
**ProgramaServidor** instancia 4 clases: **BaseDatos**, **ServidorSocket**, **ServidorRPC** y **ServidorRMI**. El constructor de la componente **BaseDatos** se encarga de hacer el barrido de las palabras en los archivos y guardar la relación de en qué archivos aparece cada palabra y cuantas veces. Los constructores de las clases **ServidorSocket**, **ServidorRPC** y **ServidorRMI** toman el objeto instanciado de la base de datos e inician su ejecución haciendo uso del método **start()**.

Una vez que los servidores están prendidos ya podemos hablar de una comunicación Cliente-Servidor.

- **Flecha azul (figura 1.0):** La comunicación inicia con la clase **ProgramaCliente**, la cual es la interfaz que ve el cliente; permite elegir al usuario el tipo de servidor al cual desea conectarse y posteriormente elegir la palabra a buscar. La palabra es entonces mandada a buscar al servidor elegido, el cual hace uso de **BaseDatos** para realizar la consulta.
- **Flecha morada (figura 1.0):** El resultado de las coincidencias (o un mensaje indicando la nula aparición de la palabra en los archivos) es mandado desde **BaseDato** hacia el servidor elegido, el cual a su vez se la envía al cliente de su mismo tipo para finalmente ser mandado para ser mostrado a **ProgramaCliente**, completando así el recorrido en capas.

## Pruebas de funcionamiento

Prender servidores:



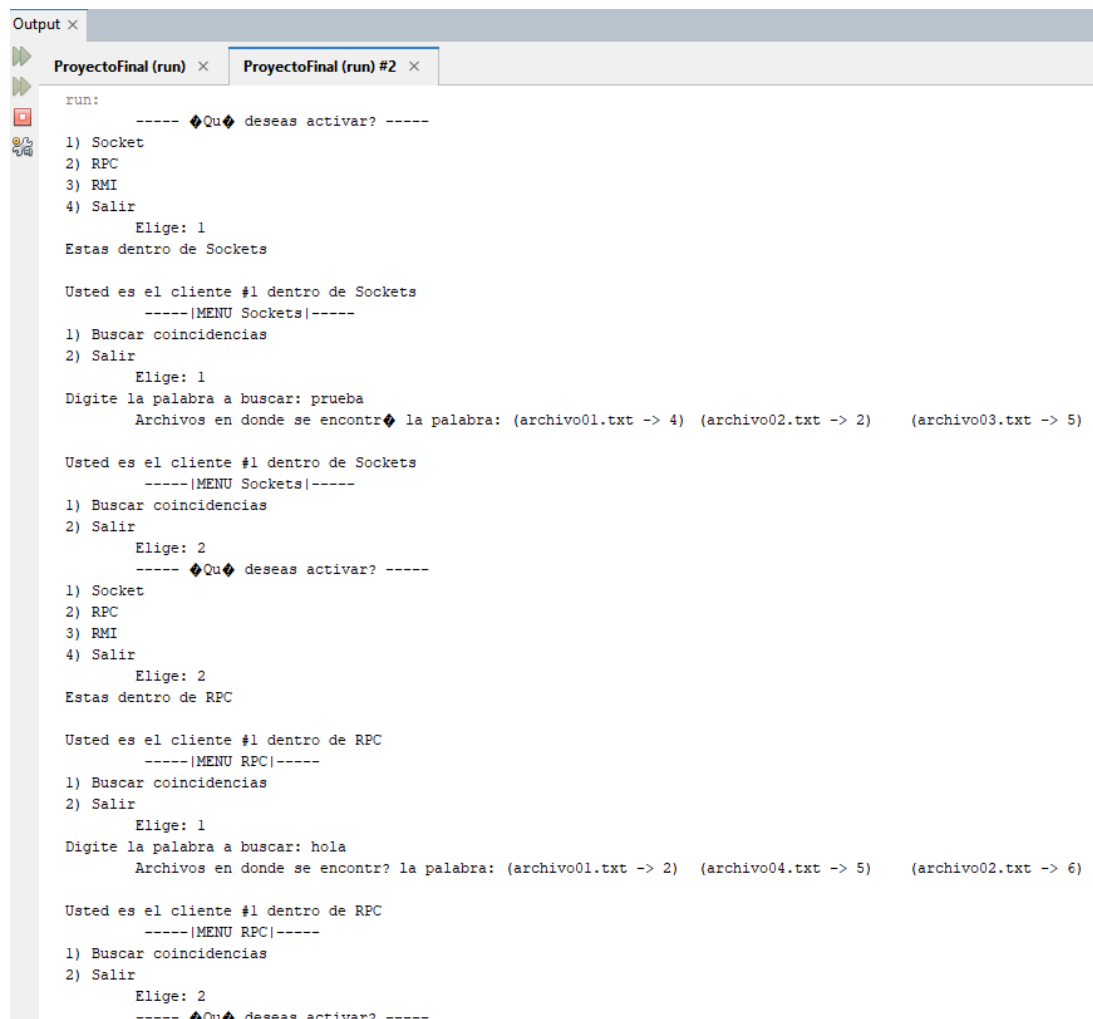
```
run:
Iniciando carga de archivos...
Carga terminada

Iniciando servidor socket...
Servidor socket iniciado correctamente...

Iniciando servidor RPC...
Servidor RPC iniciado correctamente...

Iniciando servidor RMI...
Servidor RMI iniciado correctamente...
```

Cliente #1:



```
Output x
ProyectoFinal (run) x ProyectoFinal (run) #2 x

run:
----- ◆Qu◆ deseas activar? -----
1) Socket
2) RPC
3) RMI
4) Salir
   Elige: 1
Estas dentro de Sockets

Usted es el cliente #1 dentro de Sockets
-----|MENU Sockets|-----
1) Buscar coincidencias
2) Salir
   Elige: 1
Digite la palabra a buscar: prueba
Archivos en donde se encontr◆ la palabra: (archivo01.txt -> 4) (archivo02.txt -> 2) (archivo03.txt -> 5)

Usted es el cliente #1 dentro de Sockets
-----|MENU Sockets|-----
1) Buscar coincidencias
2) Salir
   Elige: 2
----- ◆Qu◆ deseas activar? -----
1) Socket
2) RPC
3) RMI
4) Salir
   Elige: 2
Estas dentro de RPC

Usted es el cliente #1 dentro de RPC
-----|MENU RPC|-----
1) Buscar coincidencias
2) Salir
   Elige: 1
Digite la palabra a buscar: hola
Archivos en donde se encontr? la palabra: (archivo01.txt -> 2) (archivo04.txt -> 5) (archivo02.txt -> 6)

Usted es el cliente #1 dentro de RPC
-----|MENU RPC|-----
1) Buscar coincidencias
2) Salir
   Elige: 2
----- ◆Qu◆ deseas activar? -----
```

```

----- ♦Qu♦ deseas activar? -----
1) Socket
2) RPC
3) RMI
4) Salir
    Elige: 3
Estas dentro de RMI

Usted es el cliente #1 dentro de RMI
-----|MENU RMI|-----
1) Buscar coincidencias
2) Salir
    Elige: 1
Digite la palabra a buscar: como
Archivos en donde se encontr♦ la palabra: (archivo01.txt -> 1) (archivo02.txt -> 1) (archivo03.txt -> 1)

Usted es el cliente #1 dentro de RMI
-----|MENU RMI|-----
1) Buscar coincidencias
2) Salir
    Elige:

```

## Ciente #2:

```

Output x
ProyectoFinal (run) x ProyectoFinal (run) #2 x ProyectoFinal (run) #3 x
run:
----- ♦Qu♦ deseas activar? -----
1) Socket
2) RPC
3) RMI
4) Salir
    Elige: 1
Estas dentro de Sockets

Usted es el cliente #2 dentro de Sockets
-----|MENU Sockets|-----
1) Buscar coincidencias
2) Salir
    Elige: 1
Digite la palabra a buscar: hola
Archivos en donde se encontr♦ la palabra: (archivo01.txt -> 2) (archivo04.txt -> 5) (archivo02.txt -> 6)

Usted es el cliente #2 dentro de Sockets
-----|MENU Sockets|-----
1) Buscar coincidencias
2) Salir
    Elige: 2
----- ♦Qu♦ deseas activar? -----
1) Socket
2) RPC
3) RMI
4) Salir
    Elige: 2
Estas dentro de RPC

Usted es el cliente #2 dentro de RPC
-----|MENU RPC|-----
1) Buscar coincidencias
2) Salir
    Elige: 1
Digite la palabra a buscar: prueba
Archivos en donde se encontr? la palabra: (archivo01.txt -> 4) (archivo02.txt -> 2) (archivo03.txt -> 5)

Usted es el cliente #2 dentro de RPC
-----|MENU RPC|-----
1) Buscar coincidencias
2) Salir
    Elige: 2
----- ♦Qu♦ deseas activar? -----

```

```

----- ♦Qu♦ deseas activar? -----
1) Socket
2) RPC
3) RMI
4) Salir
    Elige: 3
Estas dentro de RMI

Usted es el cliente #2 dentro de RMI
-----|MENU RMI|-----
1) Buscar coincidencias
2) Salir
    Elige: 1
Digite la palabra a buscar: como
Archivos en donde se encontr♦ la palabra: (archivo01.txt -> 1) (archivo02.txt -> 1) (archivo03.txt -> 1)

Usted es el cliente #2 dentro de RMI
-----|MENU RMI|-----
1) Buscar coincidencias
2) Salir

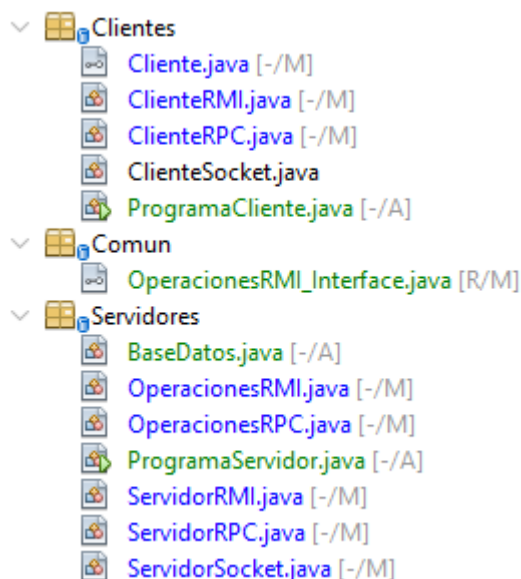
```

## Estructuras de las clases

El paquete Cliente contiene la Interfaz Cliente, la cual es implementada por las clases ClienteRMI, ClienteRPC, ClienteSocket y también contiene el programa principal ProgramaCliente.

El paquete Servidor contiene las clases ServidorRMI, ServidorRPC, ServidorSocket y BaseDatos (que son instanciadas desde ProgramaServidor), mientras que las clases OperacionesRMI y OperacionesRPC son en donde se encuentran definidas las acciones a realizar por parte de ServidorRMI y ServidorRPC respectivamente. Existe una tercera clase llamada Manejador, la cual se encuentra dentro de la clase ServidorSocket.

El paquete Común contiene las clases en común, en este caso solamente la que tiene en común ClienteRMI con ServidorRMI.



## Código fuente

### Paquete Cliente

#### ProgramaCliente (clase):

```

/***** ProgramaCliente *****/
* Este es el programa principal que se ejecuta de parte del cliente.

* El primer menú da la opción al cliente de elegir el servidor al cual
desea conectarse. El objeto "c" se instancia en este punto haciendo uso del
Polimorfismo para saber si se trata de un cliente de tipo Socket, RPC o
RMI.

* Si se eligió una opción válida, se manda a llamar a c.inicia(). Cada
clase inicia de una forma diferente de acuerdo con el tipo de Cliente, pero
en todos los casos retorna el número de cliente en caso de éxito.

* El segundo menú da la opción al cliente de buscar una palabra en los
archivos. Una vez más haciendo uso del Polimorfismo se manda a llamar a
c.buscaPalabra(palabra). Cada clase pasa la palabra al servidor para
realizar la búsqueda de una forma diferente de acuerdo con el tipo de
Cliente.

* Al salir del segundo menú se manda a llamar a c.termina(), por lo que
si un mismo cliente desea volver a conectarse con un servidor al que ya se
había conectado se le asignará un número de Cliente diferente.

* Al salir del primer menú se termina con la sesión del cliente.
*****/

package Clientes;

import Clientes.ClienteSocket;
import Clientes.Cliente;
import Clientes.ClienteRMI;
import Clientes.ClienteRPC;
import java.util.Scanner;

public class ProgramaCliente {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        while(true){
            // Primer Menu
            System.out.println("\t----- ¿Qué deseas activar? -----");
            System.out.println("1) Socket");
            System.out.println("2) RPC");
            System.out.println("3) RMI");
            System.out.println("4) Salir");
            System.out.print("\tElige: ");
            int opcion = entrada.nextInt(); entrada.nextLine(); // ->
            Quito el salto de línea
            int opcion2 = 1;

```



```

        Cliente c = null;    // Se instanciará hasta que elija el tipo
de Servidor

        String tipoConexion = "";
        if(opcion == 1){
            System.out.println("Estas dentro de Sockets\n");
            c = new ClienteSocket();
            tipoConexion = "Sockets";
        } else if(opcion == 2){
            System.out.println("Estas dentro de RPC\n");
            c = new ClienteRPC();
            tipoConexion = "RPC";
        } else if(opcion == 3){
            System.out.println("Estas dentro de RMI\n");
            c = new ClienteRMI();
            tipoConexion = "RMI";
        } else if(opcion == 4){
            break;
        } else{
            System.out.println("Elige una opción válida, vuelve a
intentarlo\n");
            continue;
        }

        int cliente = c.inicia(); // retorno el numero de cliente
        while(true){
            System.out.println("Usted es el cliente #" + cliente + "
dentro de " + tipoConexion);
            // Segundo menu
            System.out.println("\t ----|MENU " + tipoConexion +
"|----");

            System.out.println("1) Buscar coincidencias");
            System.out.println("2) Salir");
            System.out.print("\tElige: ");

            opcion2 = entrada.nextInt(); entrada.nextLine(); // quito
el salto de línea

            if(opcion2 == 1){
                System.out.print("Digite la palabra a buscar: ");
                String palabra = entrada.nextLine().toUpperCase();

                // c buscará de acuerdo al tipo de Cliente que sea
                System.out.println(c.buscaPalabra(palabra) + "\n");
            } else if(opcion2 == 2){
                c.termina();
                break;
            } else{
                System.out.println("Elige una opción válida, vuelve a
intentarlo\n");
            }
        }
    }
}
}
}
}

```

### Cliente (interfaz):

```
/* ***** Interfaz Cliente *****
 * Todos los tipos de Cliente (Socket, RPC, RMI) implementan esta clase.
 * En inicia instancian los elementos necesarios para operar y devuelven
el número de cliente que son.
 * En buscaPalabra tomar la palabra a buscar y establecen la
comunicación con su tipo de Servidor correspondiente para hacer la
búsqueda, para posteriormente devolver las coincidencias.
 * En termina en realidad la única que lo implementa es ClienteSocket
para cerrar la conexión, así como el buffer de entrada y salida de datos.
***** */
```

```
package Clientes;

public interface Cliente {
    int inicia();    // Devolverá el número de cliente en caso de éxito, o
-1 en caso de fracaso
    String buscaPalabra(String palabra);
    void termina();    // Se implementa para aquellos clientes que definan
como cerrar el buffer de datos
}
```

### ClienteRMI (clase):

```
package Clientes;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import Comun OperacionesRMI_Interface;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ClienteRMI implements Cliente{
    Registry registry;
    OperacionesRMI_Interface op;

    @Override
    public int inicia() {
        try {
            registry = LocateRegistry.getRegistry("localhost", 1237);
            op = (OperacionesRMI_Interface)
registry.lookup("rmi://localhost:1237/ServicioBuscaPalabra");

            return op.noCliente();
        } catch (RemoteException | NotBoundException ex) {
            Logger.getLogger(ClienteRMI.class.getName()).log(Level.SEVERE,
null, ex);
        }
        return -1;
    }
}
```

```

    }

    @Override
    public String buscaPalabra(String palabra) {
        try {
            return op.buscaPalabra(palabra);
        } catch (RemoteException ex) {
            Logger.getLogger(ClienteRMI.class.getName()).log(Level.SEVERE,
null, ex);
        }
        return "Fallo inesperado";
    }

    @Override
    public void termina() {
        // NO definido
    }
}

```

### ClienteRPC (clase):

```

package Clientes;

import java.io.IOException;
import java.util.Scanner;
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.apache.xmlrpc.XmlRpcClient;
import org.apache.xmlrpc.XmlRpcException;

public class ClienteRPC implements Cliente{
    public XmlRpcClient cliente;
    public Scanner entrada;

    public Vector<Integer> params;
    public Object respuestaNoCliente;

    public Vector<String> palabras;
    public Object respuestaPalabra;

    @Override
    public int inicia() {
        try{
            cliente = new XmlRpcClient("http://localhost:1232");
            entrada = new Scanner(System.in);

            params = new Vector<Integer>();

            respuestaNoCliente = cliente.execute("miServidor.noCliente",
params);
            int noCliente = ((Integer) respuestaNoCliente).intValue();
            return noCliente;

        } catch(Exception exception){
            System.err.println("JavaClient: " + exception);
        }
    }
}

```

```

        return -1;
    }
}

@Override
public String buscaPalabra(String palabra) {
    try {
        Vector<String> palabras = new Vector<String>();
        palabras.add(palabra);
        respuestaPalabra = cliente.execute("miServidor.buscaPalabra",
palabras);
        String resultado = ((String) respuestaPalabra);
        return resultado;
    } catch (XmlRpcException ex) {
        Logger.getLogger(ClienteRPC.class.getName()).log(Level.SEVERE,
null, ex);
    } catch (IOException ex) {
        Logger.getLogger(ClienteRPC.class.getName()).log(Level.SEVERE,
null, ex);
    }
    return "Fallo inesperado";
}

@Override
public void termina() {
    // No definido
}
}

```

### ClienteSocket (clase):

```

package Clientes;

import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ClienteSocket implements Cliente{
    public Socket conexion;
    public PrintWriter salida;
    private Scanner entrada;

    @Override
    public int inicia() {
        try {
            conexion = new Socket("localhost", 1235);
            salida = new PrintWriter(conexion.getOutputStream(), true);
            entrada = new Scanner(conexion.getInputStream());
            // Scanner teclado = new Scanner(System.in);
        } catch (IOException ex) {

```

```

Logger.getLogger(ClienteSocket.class.getName()).log(Level.SEVERE, null,
ex);
        return -1;
    }
    int contador = entrada.nextInt();
    entrada.nextLine(); // Me salto el salto de linea
    return contador; // Devuelvo el número de usuario
}

@Override
public String buscaPalabra(String palabra) {
    salida.println("Activo"); // Le informo al Manejador que sigo
activo
    salida.println(palabra);
    return entrada.nextLine();
}

@Override
public void termina() {
    salida.println("Inactivo"); // Le informo al Manejador antes de
morirme
    try {
        // Cierre de flujos y conexión al finalizar
        entrada.close();
        salida.close();
        conexion.close();
    } catch (IOException ex) {
        Logger.getLogger(ClienteSocket.class.getName()).log(Level.SEVERE, null,
ex);
    }
}
}
}

```

## Paquete Común

### OperacionesRMI\_Interface (interfaz):

```
package Comun;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface OperacionesRMI_Interface extends Remote{
    public String buscaPalabra(String palabra) throws RemoteException;
    public int noCliente() throws RemoteException;
}
```

## Paquete Servidor

### ProgramaServidor (clase):

```
/* ***** ProgramaServidor *****
 * Este es el programa principal que se ejecuta para iniciar los
 * Servidores.
 *
 * Se instancia un objeto de la clase "BaseDatos" llamado bd, el cual se
 * pasará por referencia a cada uno de los servidores (Socket, RPC, RMI)
 *
 * Posteriormente se instancian cada uno de los 3 tipos de servidores y
 * se encienden haciendo uso de la función start()
 *
 * Cada uno de los 3 servidores atiende a múltiples clientes de manera
 * concurrente.
 * ***** */

package Servidores;

import Servidores.ServidorRMI;
import Servidores.ServidorRPC;
import Servidores.ServidorSocket;

public class ProgramaServidor {
    public static void main(String[] args) {
        try{
            System.out.println("Iniciando carga de archivos...");
            BaseDatos bd = new BaseDatos();
            System.out.println("Carga terminada \n");

            System.out.println("Iniciando servidor socket...");
            ServidorSocket ss = new ServidorSocket(bd);
            ss.start();
            System.out.println("Servidor socket iniciado correctamente...
\n");
        }
    }
}
```

```

        System.out.println("Iniciando servidor RPC...");
        ServidorRPC sr = new ServidorRPC(bd);
        sr.start();
        System.out.println("Servidor RPC iniciado correctamente...
\n");

        System.out.println("Iniciando servidor RMI...");
        ServidorRMI srmi = new ServidorRMI(bd);
        srmi.start();
        System.out.println("Servidor RMI iniciado correctamente...
\n");
    } catch (Exception exception){
        System.err.println("Server: " + exception);
    }
}
}

```

### **BaseDatos (clase):**

```

/***** BaseDatos *****/
* Esta clase es la encargada de leer el contenido de los archivos y
guardar la información que nos interesa.

* Su constructor es el encargado de ir a la carpeta especificada,
verificar el contenido de los archivos e ir guardando la relación de las
palabras, con los nombres de los archivos en donde aparece, así como el
número de veces que aparece dicha palabra en el mismo archivo.

* Se tiene un HashMap<String, HashMap<String, Integer>> mapa. El primer
String corresponde con las palabras contenidas en el archivo, el segundo
HashMap<String, Integer> tiene como String el nombre del archivo y como
Integer la cantidad de veces que aparece esa palabra en el archivo.

* Se hace uso de una regex para que palabras dentro del archivo del
tipo ";Hola!" sea igual a "Hola", es decir, despreciando los signos de
puntuación.

* La regex solo se contruye una vez haciendo uso de:
Pattern regex = Pattern.compile("[.,;!\\?\\(\\)\\{\\}\\[\\]\\&]");

* La clase implementa el método buscaPalabra(), que toma como parámetro
la palabra a buscar y devuelve una cadena con todas las coincidencias,
dicha cadena se podría formatear para ser entregada al cliente en formato
XML o JSON. Este método será usado por todos los Servidores.
*****/

```

```

package Servidores;

import java.io.*;
import java.util.*;
import java.util.regex.*;

public class BaseDatos {

```

```

private HashMap<String, HashMap<String, Integer>> mapa = new
HashMap<>();

public BaseDatos() throws IOException{
    // Especifica la ruta de la carpeta que deseas leer
    String carpeta = "archivos";

    // Crea un objeto File para representar la carpeta
    File directorio = new File(carpeta);

    // Verifica si la ruta especificada es una carpeta
    if (directorio.isDirectory()) {
        // Obtiene una lista de todos los archivos en la carpeta
        File[] archivos = directorio.listFiles();
        Pattern regex =
Pattern.compile("[.,;!\\?\\(\\)\\{\\}\\[\\]\\;]");
        // Itera a través de la lista de archivos e imprime sus nombres
        for (File archivo : archivos) {
            if (archivo.isFile()) {
                String nombreArchivo = archivo.getName();
                // System.out.println(nombreArchivo);
                String ruta = carpeta + "\\\" + nombreArchivo;
                try (BufferedReader br = new BufferedReader(new
FileReader(ruta))) {
                    String linea;
                    while ((linea = br.readLine()) != null) {
                        linea = linea.toUpperCase();
                        linea = regex.matcher(linea).replaceAll(""); //
Quito los que no me interesan

                        String[] palabras = linea.split(" ");
                        for (String palabra : palabras) {
                            if(mapa.containsKey(palabra)){

                                if(mapa.get(palabra).containsKey(nombreArchivo)){
                                    int valor = 1 +
mapa.get(palabra).get(nombreArchivo);

                                mapa.get(palabra).put(nombreArchivo, valor);
                                } else{

                                mapa.get(palabra).put(nombreArchivo, 1);
                                }

                                } else{
                                    HashMap<String, Integer> nuevo = new
HashMap<>();

                                    nuevo.put(nombreArchivo, 1);

                                    mapa.put(palabra, nuevo);
                                }
                            }
                        // System.out.println(linea); // Imprimir cada
línea del archivo
                    }
                } catch (IOException e) {

```



```

        System.err.println("Error al leer el archivo: " +
e.getMessage());
    }

    }
} else {
    System.out.println("La ruta especificada no es una carpeta.");
}
}

public String buscaPalabra(String palabra){
    String coincidencias = "";
    if(this.mapa.containsKey(palabra)){
        coincidencias = "\tArchivos en donde se encontró la palabra: ";
        Iterator<HashMap.Entry<String, Integer>> iterator =
mapa.get(palabra).entrySet().iterator();
        while(iterator.hasNext()){
            HashMap.Entry<String, Integer> actual = iterator.next();
            coincidencias += "(" + actual.getKey() + " -> " +
actual.getValue() + ")\t";
        }
    } else{
        coincidencias = "Palabra no encontrada en ningun archivo";
    }
    return coincidencias;
}
}

```

### **ServidorSocket (clase):**

```

package Servidores;

import java.net.*;
import java.io.*;
import java.util.*;
import java.util.logging.*;

public class ServidorSocket {
    private BaseDatos bd;

    public ServidorSocket(BaseDatos bd){
        this.bd = bd;
    }

    // Se hizo uso de una lambda, para que el servidor de Socket tenga su
    // propio hilo
    // Con esto lo que se consigue es que se pueda instanciar en cualquier
    // momento en ProgramaServidor
    public void start() {
        (new Thread(() -> {
            try {
                int numeroHilo = 1;
                ServerSocket servidor = new ServerSocket(1235);
                while(true){

```

```

        Socket entrante = servidor.accept();
        // System.out.println("Generando Hilo " + numeroHilo +
        ".");

        Runnable r = new Manejador(entrante, numeroHilo,
        this.bd);

        Thread t = new Thread(r);
        t.start();
        ++numeroHilo;
    }
} catch (Exception ex) {

Logger.getLogger(ServidorSocket.class.getName()).log(Level.SEVERE, null,
ex);
    }
    })).start();
}
}

```

### Manejador (clase):

```

class Manejador implements Runnable {
    private BaseDatos bd;
    private Socket cliente;
    private int contador;

    public Manejador(Socket cliente, int contador, BaseDatos bd){
        this.cliente = cliente;
        this.contador = contador;
        this.bd = bd;
    }

    @Override
    public void run(){
        try{
            try{
                Scanner entrada = new Scanner(cliente.getInputStream());
                PrintWriter salida = new
                PrintWriter(cliente.getOutputStream(), true);
                salida.println(contador); // Le informo que numero de
                cliente es

                while(true){
                    String verifica = entrada.nextLine();
                    if(verifica.equals("Activo")){
                        String palabra = entrada.nextLine();
                        salida.println(this.bd.buscaPalabra(palabra));
                    } else if(verifica.equals("Inactivo")){
                        break;
                    }
                }
            } finally {
                cliente.close();
                // System.out.println("Cliente " + contador + "
                finalizado.");
            }
        }
    }
}

```

```

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

```

### **ServidorRPC (clase):**

```

package Servidores;

import org.apache.xmlrpc.WebServer;

public class ServidorRPC {
    private BaseDatos bd;

    public ServidorRPC(BaseDatos bd) {
        this.bd = bd;
    }

    public void start() throws Exception {
        WebServer server = new WebServer(1232);
        OperacionesRPC op = new OperacionesRPC(this.bd); // OperacionesRPC
op = new OperacionesRPC(this.bd);
        server.addHandler("miServidor", op);
        server.start();
    }
}

```

### **OperacionesRPC (clase):**

```

package Servidores;

public class OperacionesRPC {
    private BaseDatos bd;
    private int noCliente = 1;

    public OperacionesRPC(BaseDatos bd) {
        this.bd = bd;
    }

    public String buscaPalabra(String palabra) {
        return this.bd.buscaPalabra(palabra);
    }

    public int noCliente() {
        return noCliente++;
    }
}

```

### **ServidorRMI (clase):**

```

package Servidores;

```

```

import Comun OperacionesRMI_Interface;
import java.rmi.Remote;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ServidorRMI {
    private BaseDatos bd;

    public ServidorRMI(BaseDatos bd) {
        this.bd = bd;
    }

    public void start() throws Exception {
        OperacionesRMI_Interface op = new OperacionesRMI(bd);

        Registry reg = LocateRegistry.createRegistry(1237);

        Remote stub = UnicastRemoteObject.exportObject(op, 0);
        reg.bind("rmi://localhost:1237/ServicioBuscaPalabra", stub);
    }
}

```

### OperacionesRMI (clase):

```

package Servidores;

import Comun OperacionesRMI_Interface;

public class OperacionesRMI implements OperacionesRMI_Interface{
    private BaseDatos bd;
    private int noCliente = 1;

    public OperacionesRMI(BaseDatos bd) {
        this.bd = bd;
    }

    @Override
    public String buscaPalabra(String palabra){
        return this.bd.buscaPalabra(palabra);
    }

    @Override
    public int noCliente(){
        return noCliente++;
    }
}

```