

Desarrollo de Sistemas Distribuidos

Práctica 2

Programación en RMI

Dpto. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 2015-16

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada

Índice

- Presentación de objetivos
- Comunicaciones entre procesos en sistemas distribuidos
 - Introducción
 - Sockets y RPC
 - RMI
- Java RMI
 - Objetivos
 - Modelo de Objetos Distribuidos
 - Creación de Aplicaciones RMI en Java
 - Compilación de Programas Java RMI
 - Ejecución de Programas Java RMI
- Ejercicios propuestos

Objetivos

- Objetivo *general*:
 - En esta práctica se pretende que el alumno conozca y adquiera experiencia en el manejo de los mecanismos para la implementación de programas distribuidos en Java utilizando la API RMI (*Remote Method Invocation*)
- Objetivos *específicos*:
 - Conocer los mecanismos para definir interfaces de objetos remotos
 - Aprender a implementar interfaces remotas y hacerlas accesibles
 - Implementar programas cliente basados en esta tecnología
 - Compilar, desplegar y ejecutar los programas desarrollados

Objetivos

- Enlaces a información complementaria
 - [Tutorial de Java para RMI](#) (*“trail RMI”*)
 - [Especificación de Java RMI](#)
 - [Remote Method Invocation Home](#)

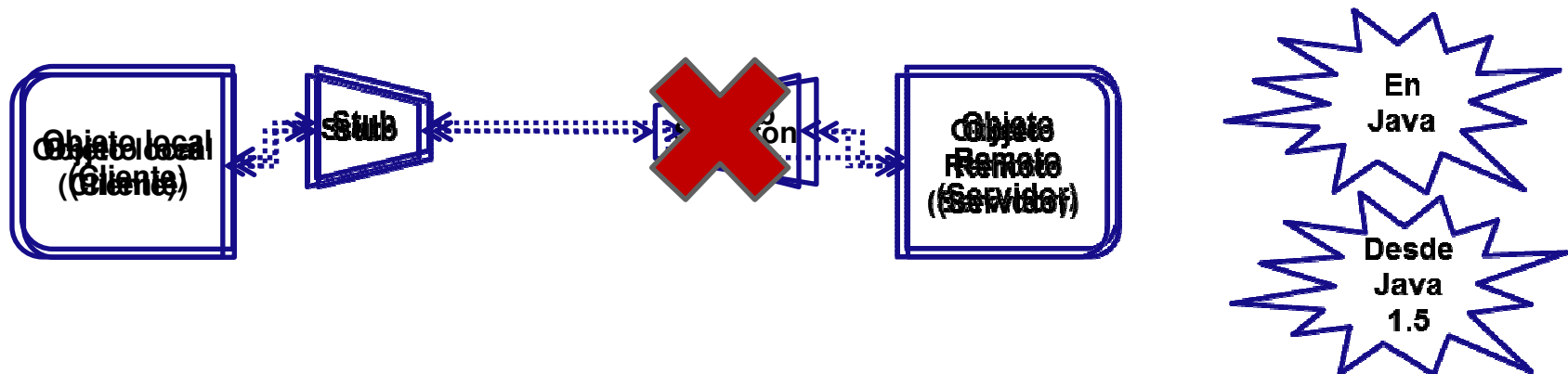
Introducción



- En sistemas distribuidos, existen diferentes tecnologías para comunicar dispositivos, como el uso de *sockets* y RPC
- **Sockets** → requiere el tratamiento de protocolos de **bajo nivel** para la codificación y decodificación de los mensajes intercambiados, lo que dificulta la programación y es más proclive a errores
- **RPC** → resulta adecuado para programas basados en llamadas a procedimientos, pero no encaja bien en **sistemas de objetos distribuidos**, donde se requiere la comunicación entre instancias de clases residentes en espacios de direcciones distintos

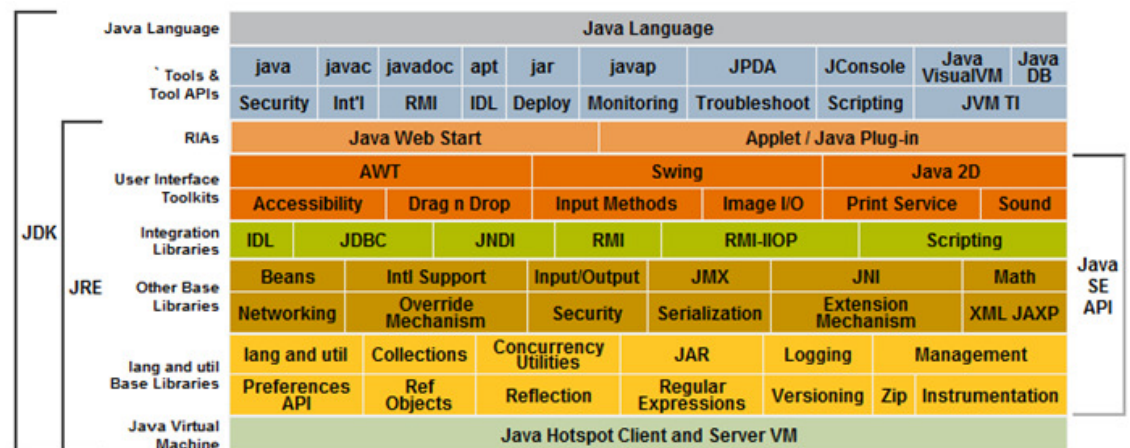
RMI

- Para una correcta adecuación a la semántica de invocación de objetos, los sistemas de objetos requieren *remote method invocation* o *RMI*
- En estos sistemas, un objeto *delegado* o *substituto* (llamado *stub*) gestiona las invocaciones al objeto remoto, gestionando la *serialización* y *deserialización* de parámetros, así como de los valores devueltos



Java RMI

- Java RMI es una tecnología que permite construir objetos distribuidos que operen en el entorno de aplicación Java
- Como entorno de funcionamiento se presupone el de la **Java Virtual Machine** (JVM), que sigue el enfoque “*Write Once, Run **Anywhere***” (“se codifica una vez y se ejecuta en cualquier sitio”). Java RMI se basa y extiende este enfoque para que se ejecute en/por todas partes (“*run **everywhere***”)



Objetivos de Java RMI

- Soportar invocaciones remotas sobre objetos de diferentes máquinas virtuales
- Soportar comunicaciones desde servidores a *applets* (conocidas como *callbacks*)
- Integrar de forma natural el modelo de objetos distribuidos en el lenguaje de programación Java, preservando la mayor parte de la semántica de objetos en este lenguaje
- Hacer visibles las diferencias entre el modelo de objetos distribuidos y el modelo de objetos de la plataforma Java
- Preservar seguro el entorno de la plataforma Java proporcionado por gestores de seguridad y cargadores de clases
- ...

Como requisito subyacente a los objetivos anteriores aparece la simplicidad de uso y la homogeneidad/naturalidad del modelo Java RMI, de forma que las construcciones de dicho modelo puedan utilizarse de forma similar al resto de los elementos del lenguaje Java

Modelo de Objetos Distribuidos

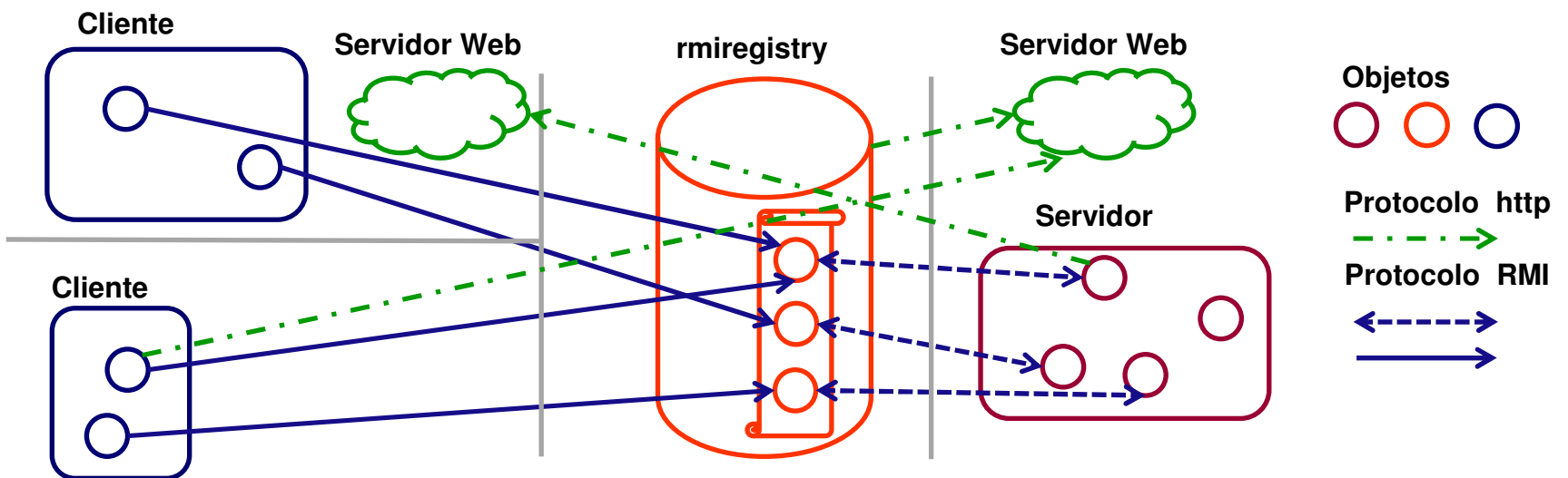
- En el modelo de objetos distribuidos de la plataforma Java, un **objeto remoto** es aquel cuyos métodos pueden invocarse desde otra JVM, posiblemente en otra computadora
- Un objeto de este tipo **se describe** mediante una o más **interfaces remotas**, que son **interfaces** Java donde se declaran los métodos del objeto remoto. Las **clases** implementan los métodos declarados en las **interfaces** y eventualmente, métodos adicionales
 - **Remote method invocation** (RMI) es la acción de invocar un método en una **interfaz remota** de un objeto remoto. La invocación de un método en una **interfaz remota** sigue la misma sintaxis que en un objeto local

Modelo de Objetos Distribuidos

- Normalmente, las aplicaciones RMI constan de dos aplicaciones separadas: **servidor** y **cliente**
- **Servidor:**
 1. Crea objetos remotos
 2. Hace accesibles las referencias a dichos objetos remotos
 3. Aguada a la invocación de métodos sobre dichos objetos remotos por parte de los clientes
- **Cliente:**
 1. Obtiene una referencia remota a uno o más objetos remotos en el servidor
 2. Invoca métodos sobre los objetos remotos
- RMI proporciona los mecanismos para que servidor y cliente se comuniquen e intercambien información. Esta aplicación normalmente se denomina **aplicación de objetos distribuidos**

Aplicaciones de Objetos Distribuidos

- Las aplicaciones de objetos distribuidos requieren:
 - Localizar objetos remotos
 - Para ello, las aplicaciones pueden registrar sus objetos remotos utilizando el servicio `rmiregistry`, o puede enviar y devolver referencias a objetos remotos como argumentos y resultados de procedimientos.
 - Comunicarse con objetos remotos
 - Cargar bytecodes de objetos que se pasan como parámetros o valores de retorno



Modelos Distribuidos y no-Distribuidos

- **Similitudes:**

- Una referencia a un objeto remoto puede ser pasada como argumento o devuelta como resultado en cualquier invocación de un método (local o remoto)
- Un objeto remoto puede convertirse ("*cast*") al tipo de cualquiera de las interfaces remotas soportadas en la implementación utilizando la sintaxis para castings de Java
- Se puede utilizar el operador `instanceof` para comprobar las interfaces remotas soportadas por un objeto remoto

- **Diferencias:**

- Los clientes de objetos remotos interactúan con interfaces remotas, nunca con las clases que implementan dichas interfaces
- Los argumentos no-remotos y los resultados de una invocación remota se pasan por copia (en lugar de por referencia; las referencias sólo tienen sentido dentro de la misma JVM)
- Los objetos remotos se pasan por referencia
- La semántica de algunos métodos de la clase `java.lang.Object` se especializan para objetos remotos
- Es necesario tratar con tipos especiales de excepciones por tratarse de un sistema distribuido

Creación de Aplicaciones RMI en Java

- La utilización de Java RMI para aplicaciones distribuidas sigue las siguientes etapas:
 1. Diseñar e implementar los componentes de la aplicación distribuida**
 2. Compilar el código fuente
 3. Hacer las clases accesibles a través de la red
 4. Iniciar la aplicación
- **Diseñar e implementar los componentes de la aplicación distribuida:
Esta etapa incluye:
 1. Definir las interfaces remotas: Los métodos que pueden ser invocados remotamente por los clientes (sólo los de las interfaces, no los de las clases que los implementan)
 2. Implementar los objetos remotos: Deben implementar una o más interfaces remotas (también pueden implementar otras que sólo sean locales, pero no estarán disponibles para los clientes)
 3. Implementar los clientes

Creación de Aplicaciones RMI en Java

- En lo que sigue, vamos a ver algunos ejemplos de **creación**, **compilación** y **ejecución** de aplicaciones RMI en Java.
- Comenzamos viendo un sencillo ejemplo de interfaz, y objeto remoto que la implementa, consistentes en la impresión por pantalla de un mensaje de forma remota en función del identificador del proceso que lo invoca y que se pasa como parámetro: si el identificador es 0, suspende el proceso durante 5 segundos.
- La interfaz tendrá por nombre “Ejemplo_I” (“*ejemplo de interfaz*”, no “ejemplo 1”). Por claridad, en lo que sigue a todos los nombres de interfaces le añadiremos el sufijo “_I” (“subrayado *i* máyúscula”).
- En este caso, sólo es necesario definir un método: “escribir_mensaje”

```
//Diseño de la interfaz. Fichero Ejemplo_I.java del directorio "simple"
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Ejemplo_I extends Remote {
    public void escribir_mensaje (int id_proceso) throws RemoteException;
}
```

Creación de Aplicaciones RMI en Java

- Implementación del objeto remoto

```
//Implementación de la interfaz remota. Fichero Ejemplo.java del directorio
//"simple"
public class Ejemplo implements Ejemplo_I {

    public void escribir_mensaje (int id_proceso) {
        System.out.println("Recibida petición de proceso: "+id_proceso);
        if (id_proceso == 0) {
            try{
                System.out.println("Empezamos a dormir");
                Thread.sleep(5000);
                System.out.println("Terminamos de dormir");
            }
            catch (Exception e) {
                System.err.println("Ejemplo exception:");
                e.printStackTrace();
            }
        }
        System.out.println("\nHebra "+id_proceso);
    }
    ...
}
```

Declaración e
implementación de la interfaz

Creación de Aplicaciones RMI en Java

- Una vez implementado el objeto remoto, necesitamos crear el objeto remoto inicial y exportarlo al entorno RMI, para habilitar la recepción de invocaciones remotas.
- Pasos a seguir:
 - Crear e instalar un **gestor de seguridad** (“**security manager**”)
 - Crear y exportar uno más objetos remotos
 - Registrar al menos un objeto remoto en el registro RMI
- Este proceso puede realizarse dentro de algún método de inicialización de la propia clase o llevarse a cabo en cualquier otro método de otra clase. En el ejemplo, será la propia clase que implemente el objeto remoto quien lo realice.

Creación de Aplicaciones RMI en Java

- Implementación del objeto remoto

```
//Continuación del código anterior... Fichero Ejemplo.java del directorio  
//"simple"
```

```
public static void main(String[] args) {  
    if (System.getSecurityManager() == null) {  
        System.setSecurityManager(new SecurityManager());  
    }
```

Instalación del gestor de seguridad

```
}  
try {
```

Creamos una instancia

```
    Ejemplo_I prueba = new Ejemplo();
```

```
    Ejemplo_I stub =
```

Puerto anónimo para atender peticiones

```
        (Ejemplo_I) UnicastRemoteObject.exportObject(prueba, 0);
```

```
    Registry registry = LocateRegistry.getRegistry();
```

```
    String nombre_objeto_remoto = "un_nombre_para_obj_remoto";
```

```
    registry.rebind(nombre_objeto_remoto, stub);
```

```
    System.out.println("Ejemplo bound");
```

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

```
    System.err.println("Ejemplo exception:");
```

```
    e.printStackTrace();
```

```
}
```

La exportamos y le damos un nombre con el que identificarla en el RMI registry

Mientras exista una referencia al objeto Ejemplo, local o remota, no se cerrará ni liberará su espacio de memoria por el recolector de basura

Creación de Aplicaciones RMI en Java

- Implementación del cliente

```
//Implementación del cliente que accede a la interfaz remota Ejemplo_I.  
//Fichero Cliente_Ejemplo.java del directorio "simple"  
import java.rmi.registry.LocateRegistry;  
import java.rmi.registry.Registry;  
  
public class Cliente_Ejemplo {  
    public static void main(String args[]) {  
        if (System.getSecurityManager() == null) {  
            System.setSecurityManager(new SecurityManager() );  
        }  
        try {  
            Registry registry = LocateRegistry.getRegistry(args[0]);  
            System.out.println("Buscando el objeto remoto");  
            String nombre_objeto_remoto = "un_nombre_para_obj_remoto";  
            Ejemplo_I instancia_local = (Ejemplo_I)  
registry.lookup(nombre_objeto_remoto);  
            System.out.println("Invocando el objeto remoto");  
            instancia_local.escribir_mensaje(Integer.parseInt(args[1]));  
        } catch (Exception e) {  
            System.err.println("Ejemplo_I exception:");  
            e.printStackTrace();  
        }  
    }  
}
```

Creación de Aplicaciones RMI en Java

- Al igual que el servidor `Ejemplo`, el cliente comienza instalando un servidor de seguridad, necesario para que el stub local pueda descargar la definición de una clase desde el servidor.
- A continuación, el cliente almacena en una variable el nombre con el que buscar en el `registry` el objeto remoto. Utilizamos el mismo nombre que la instancia de la clase `Ejemplo` para ligar el objeto remoto.
- Asimismo, el cliente invoca `LocateRegistry.getRegistry` para obtener una referencia al RMI `registry` en la máquina del servidor, cuyo nombre se pasará como primer argumento al programa cliente (`args[0]`, p.ej.: `localhost`, `ei142168.ugr.es`, etc).
- Entonces, el cliente invoca el método `lookup` en el `registry` para buscar el objeto remoto por su nombre.

Compilación de Programas Java RMI

- En este caso, dado que no se trata de un programa muy complejo y que no hemos creado un estructura de paquetes, podemos compilar todos los archivos con la orden `javac *.java`, estableciendo previamente como directorio de trabajo actual aquel donde se encuentren los ficheros fuente (en este caso sería el directorio “*simple*”).
- En un entorno de trabajo real, donde los ficheros de servidor y cliente pueden encontrarse en máquinas distintas, hay que velar por que, una vez compiladas, las clases estén públicamente accesibles a través de la red, lo que se puede hacer por medio de un servidor Web sencillo.

Ejecución de Programas Java RMI

- Como ya hemos visto, servidor y cliente (en este caso, clases **Ejemplo** y **Cliente_Ejemplo**, respectivamente) se ejecutan instalando un gestor de seguridad (*security manager*).
- Por esta razón, para ejecutar cualquiera de ellos es necesario especificar un *fichero de políticas de seguridad* tal que conceda al código los permisos de seguridad que necesita para ejecutarse.
- A continuación mostramos un ejemplo de fichero de políticas de seguridad que utilizaremos para lanzar servidor y cliente.

Da igual el sistema operativo: las barras deben ir siempre hacia la derecha

```
//Fichero server.policy del directorio "simple"  
grant codeBase "file:/C:/Users/froxendo/p4_RMI/simple/" {  
    permission java.security.AllPermission;  
};
```

Debe ser una URL

Este path habrá que adaptarlo a cada usuario

- En este ejemplo, se conceden todos los permisos a las clases en el **classpath** del programa local, de manera que no se conceden permisos para el código descargado desde otras ubicaciones.
- El fichero se denomina "server.policy" y lo utilizaremos al lanzar, tanto el cliente, como el servidor.

Ejecución del Servidor

- Antes de ejecutar el programa servidor “Ejemplo”, debemos lanzar el RMI registry.
 - Esto podemos realizarlo ejecutando la orden `rmiregistry` (en Linux podemos lanzarlo en segundo plano con `&`)
 - Por defecto, el `registry` se ejecuta en el puerto 1099, y este es también el número de puerto que se utiliza en todas las operaciones con el registry (como `getRegistry`). Si queremos utilizar otro puerto, debemos pasarlo como argumento (p. ej. `rmiregistry 2011 &`), y por tanto, también en todas las operaciones con el `registry`.
- Una vez que se ha iniciado el `registry`, se puede lanzar el servidor.

Adaptar según
usuario

```
#java -cp . -Djava.rmi.server.codebase=file:/C:/Users/froxendo/p4_RMI/simple/  
-Djava.rmi.server.hostname=localhost  
-Djava.security.policy=server.policy Ejemplo
```

Ejecución del Servidor

```
#java -cp . -Djava.rmi.server.codebase=file:/C:/Users/froxendo/p4_RMI/simple/  
-Djava.rmi.server.hostname=localhost  
-Djava.security.policy=server.policy Ejemplo
```

- La orden anterior especifica distintas propiedades para el entorno Java:
 - `java.rmi.server.codebase` especifica la ubicación desde la que poder descargar definiciones de clases desde el servidor.
 - `java.rmi.server.hostname` especifica el nombre de la máquina donde colocar los stubs para los objetos remotos.
 - `java.security.policy` especifica el fichero de políticas de seguridad que se pretenden seguir o conceder.

Ejecución del Cliente

- Una vez que el RMI `registry` y el programa servidor están ejecutándose, podemos lanzar los clientes. Para ello, debemos especificar:
 - La ubicación desde la que el cliente puede proporcionar la definición de sus clases mediante la propiedad `java.rmi.server.codebase`. En este caso, dado que el cliente (`Cliente_Ejemplo`) no envía como parámetro ninguno que sea de un tipo o clase desconocidos para el servidor, no hace falta especificarlo.
 - La propiedad `java.security.policy` con los permisos que queremos conceder al programa.
 - El nombre de la máquina (host) donde se encuentra el servidor, así como el identificador del cliente. Ambos parámetros se pasan como argumentos de la línea de comandos (`args[0]` y `args[1]`).

Ejecución del Cliente

- Ejemplos de órdenes para lanzar clientes

```
#java -cp . -Djava.security.policy=server.policy Cliente_Ejemplo localhost 0
```

Cliente 0



```
#java -cp . -Djava.rmi.server.codebase=file:///C:/Users/froxendo/p4_RMI/  
-Djava.security.policy=server.policy Cliente_Ejemplo localhost 1
```

Cliente 1 especificando la propiedad
java.rmi.server.codebase



Un Ejemplo Multihebrado

- En lugar de lanzar varios clientes, creamos varias hebras que realizan la misma tarea de imprimir un mensaje remoto accediendo al stub de un objeto remoto.
- El objetivo es ver también la gestión de la concurrencia en RMI.
- En esta ocasión, en vez de pasar al objeto remoto un número entero, pasamos una cadena **String**.

```
//Diseño de la interfaz. Fichero Ejemplo_I.java del directorio "multi_hebra"  
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface Ejemplo_I extends Remote {  
    public void escribir_mensaje (String mensaje) throws RemoteException;  
}
```

Un Ejemplo Multihebrado

- Implementación del objeto remoto: programa servidor

//Diseño de la interfaz. Fichero Ejemplo.java del directorio "multi_hebra"

...

```
public class Ejemplo implements Ejemplo_I {

    public void escribir_mensaje (String mensaje) {
        System.out.println("\\nEntra Hebra "+mensaje);

        //Buscamos los procesos 0, 10, 20,...
        if (mensaje.endsWith("0")) {
            try{
                System.out.println("Empezamos a dormir");
                Thread.sleep(5000);
                System.out.println("Terminamos de dormir");
            }
            catch (Exception e) {
                System.err.println("Ejemplo exception:");
                e.printStackTrace();
            }
        }
        System.out.println("Sale Hebra "+mensaje);
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String nombre_objeto_remoto = "un_nombre_para_obj_remoto";
            ...//Igual que el anterior
        }
    }
}
```

Un Ejemplo Multihebrado

- Programa cliente

```
//Diseño de la interfaz. Fichero Cliente_Ejemplo_Multi_Threaded.java en directorio "multi_hebra"
...
public class Cliente_Ejemplo_Multi_Threaded
    implements Runnable {

    public String nombre_objeto_remoto = "un_nombre_para_obj_remoto";
    public String server;

    public Cliente_Ejemplo_Multi_Threaded (String server) {
        //Almacenamos el nombre del host servidor
        this.server = server;
    }

    public void run() {
        System.out.println("Buscando el objeto remoto");
        try {
            Registry registry = LocateRegistry.getRegistry(server);
            Ejemplo_I instancia_local = (Ejemplo_I) registry.lookup(nombre_objeto_remoto);
            System.out.println("Invocando el objeto remoto");
            instancia_local.escribir_mensaje(Thread.currentThread().getName());
        } catch (Exception e) {
            System.err.println("Ejemplo_I exception:");
            e.printStackTrace();
        }
    }
    ...
}
```

Un Ejemplo Multihebrado

- Programa cliente (II, viene de la transparencia anterior)

```
//Diseño de la interfaz. Fichero Cliente_Ejemplo_Multi_Threaded.java en directorio "multi_hebra"
...
public static void main(String args[]) {

    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    int n_hebras = Integer.parseInt(args[1]);
    //Seguimos el mismo esquema de creación de vectores de hebras visto en asignaturas
    anteriores
    Cliente_Ejemplo_Multi_Threaded[] v_clientes = new Cliente_Ejemplo_Multi_Threaded[n_hebras];
    Thread[] v_hebras = new Thread[n_hebras];
    for (int i=0;i<n_hebras;i++) {
        //A cada hebra le pasamos el nombre el servidor
        v_clientes[i] = new Cliente_Ejemplo_Multi_Threaded(args[0]);
        v_hebras[i] = new Thread(v_clientes[i], "Cliente "+i);
        v_hebras[i].start();
    }
}
```

Ejecución de Servidor y Cliente Multihebrado

- Suponemos lanzado el *rmiregistry*

Servidor

```
#java -cp . -Djava.rmi.server.codebase=file:/C:/Users/froxendo/p4_RMI/multi_hebra/  
-Djava.rmi.server.hostname=localhost  
-Djava.security.policy=server.policy Ejemplo
```

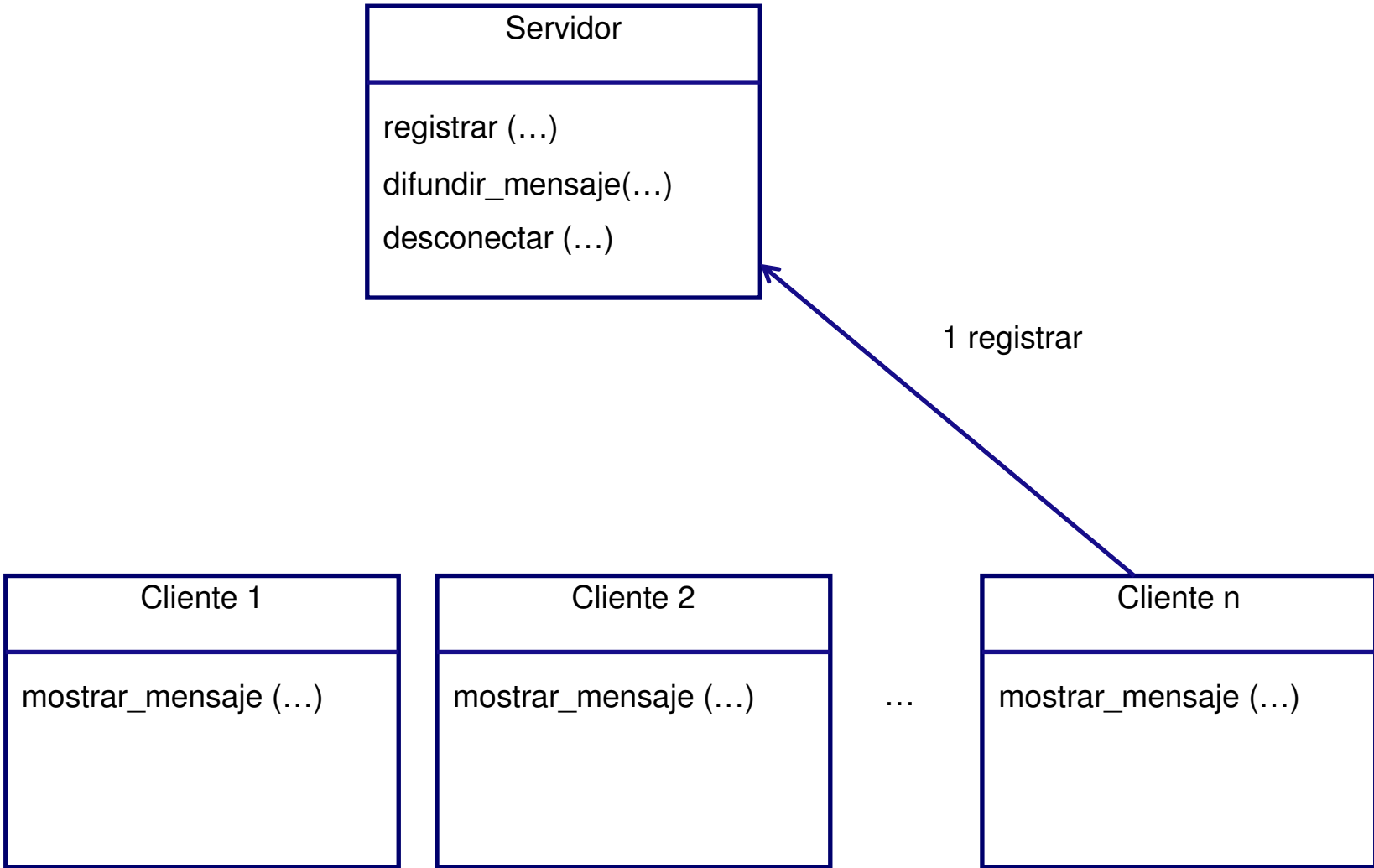
```
#java -cp . -Djava.rmi.server.codebase=file:/C:/Users/froxendo/p4_RMI/multi_hebra/  
-Djava.security.policy=server.policy  
Cliente_Ejemplo_Multi_Threaded localhost 11
```

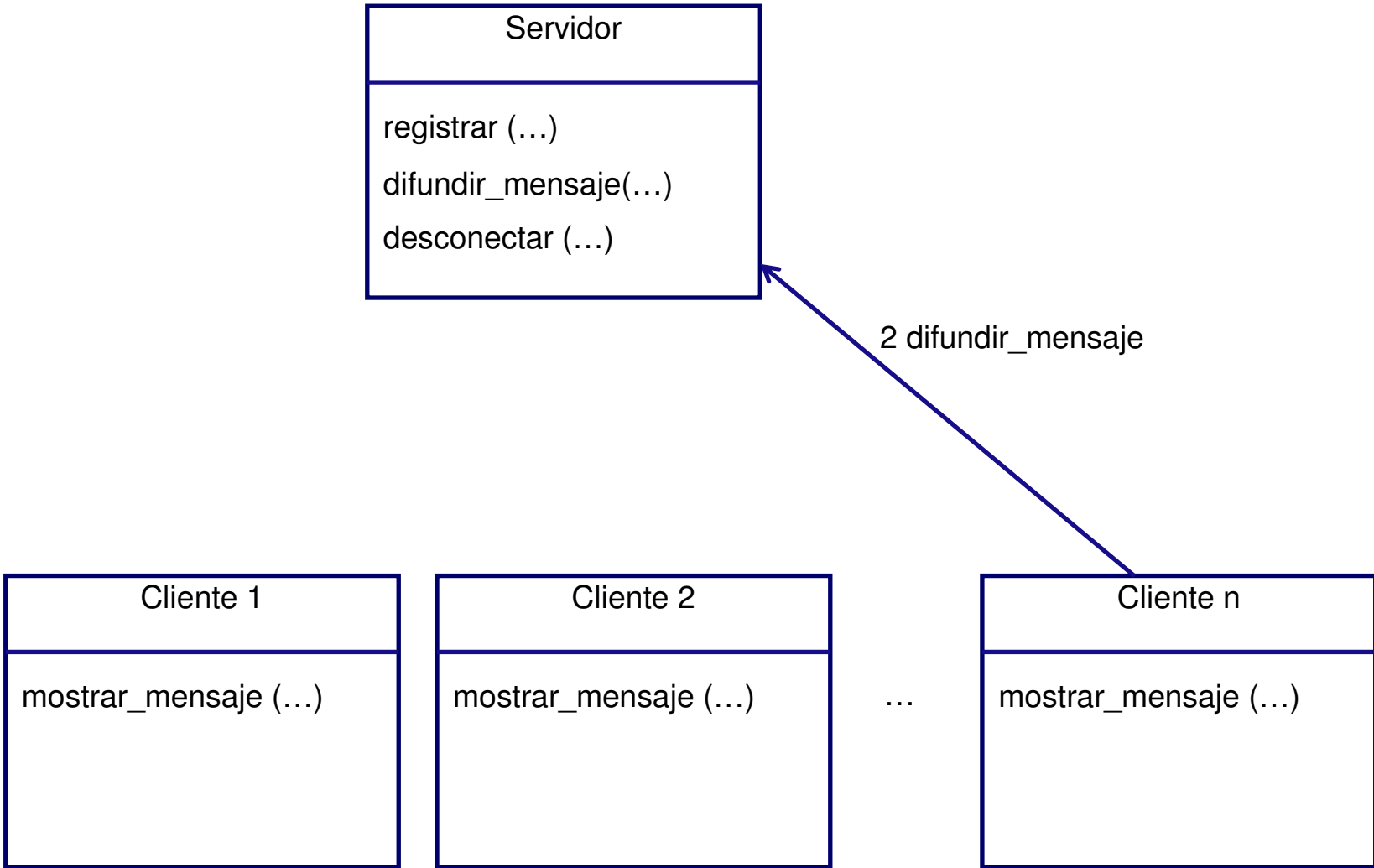
Cliente: Esta vez el segundo argumento especifica el número de hebras a crear

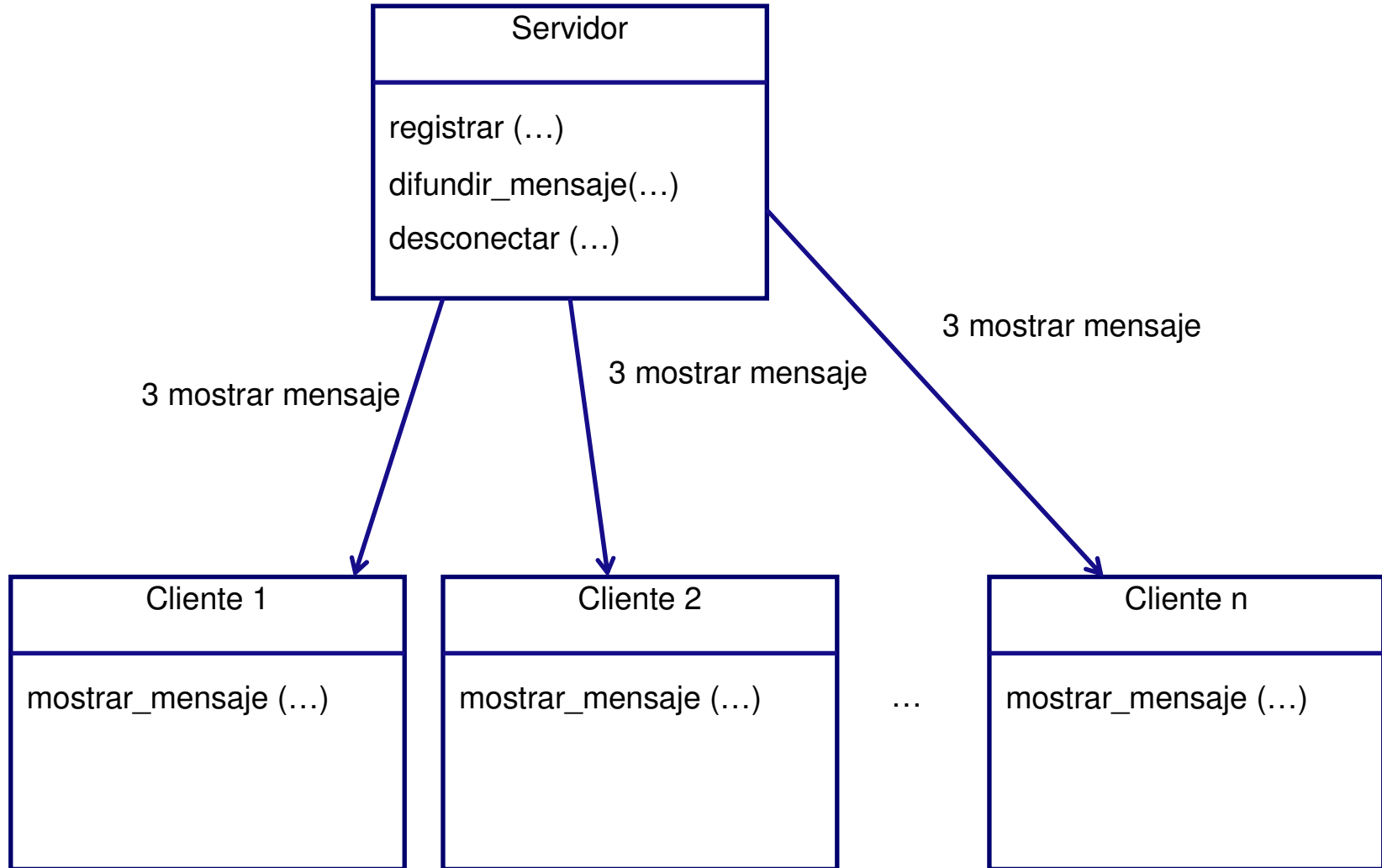
Ejercicio propuesto 1:

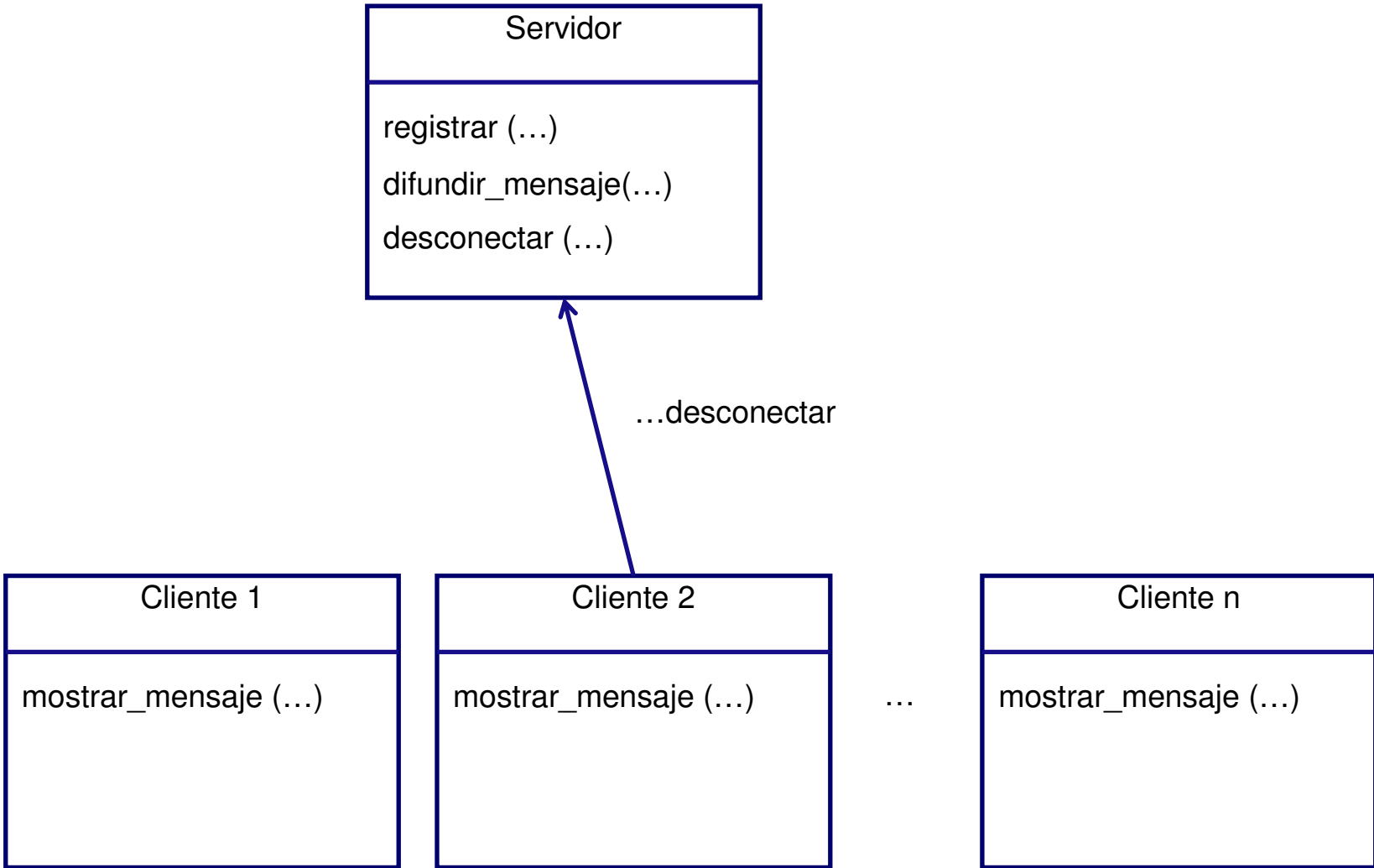
Chat centralizado

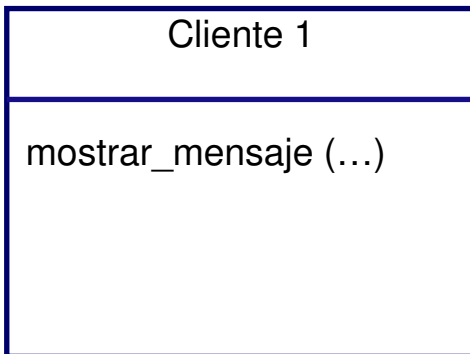
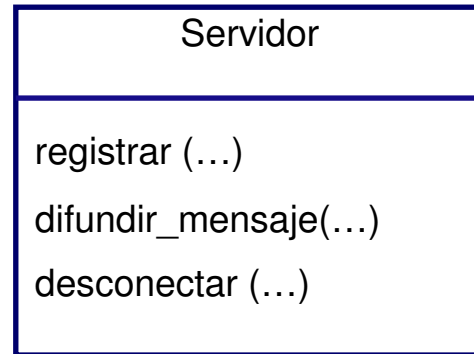
- Utilizando la tecnología Java RMI, se pide desarrollar un chat con una interfaz gráfica sencilla consistente en, al menos, los siguientes elementos:
 - Un campo de texto en el que mostrar los mensajes enviados por los usuarios, un campo de texto para escribir los mensajes que se deseen enviar y un botón “Enviar”
- Existirá una entidad a modo servidor centralizado que mantendrá un registro de usuarios conectados
- Cada vez que un usuario inicie la aplicación de chat, notificará al servidor para que éste lo dé de alta en el listado de usuarios conectados
- Los mensajes en el chat se envían desde un usuario al servidor y éste se encarga de difundirlo entre todos los usuarios conectados para que lo muestren en el campo de texto correspondiente
- Debe gestionarse adecuadamente la conexión y desconexión de usuarios, ya que el recolector de basura de la JVM mantendrá en memoria todos los objetos activos mientras exista alguna referencia a cada uno de ellos



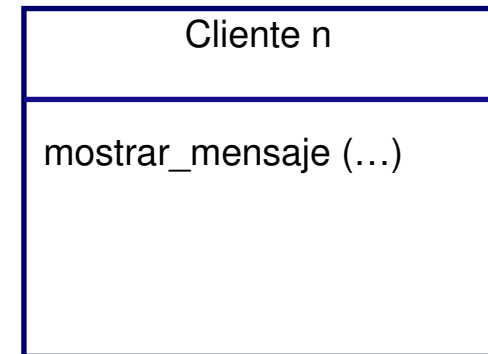








...



Ejercicio propuesto 2:

Chat peer-to-peer

- Partiendo del ejercicio propuesto 1, se pide desarrollar un chat *peer-to-peer* en el que cada usuario es capaz de comunicarse directamente con otro sin necesidad de que los mensajes se envíen a través de la entidad servidor, que sólo mantendrá un registro de los usuarios activos en cada momento