2024

# Java Remoted Method Invocation (RMI)

PRÁCTICA 3

CARMEN CHUNYIN FERNÁNDEZ NÚÑEZ

Índice

# 1 ESTRUCTURA DE ARCHIVOS Y COMPILACIÓN

Esta sección sirve únicamente a modo de explicación de cómo están estructuradas las carpetas y cómo se compila esta práctica. En el directorio raíz se encuentran 2 carpetas, la carpeta denominada “Ejemplos” contiene una carpeta por cada ejemplo, y la segunda carpeta denominada “Donaciones”, destinada al ejercicio del servidor de donaciones.

Para ejecutar tanto los ejemplos cómo el ejercicio propuesto, se debe ejecutar primero el **script_server.sh**, una vez ejecutado, en otra terminal se ejecuta el **script_cliente.sh**

# 2 PRIMERA PARTE. EJEMPLOS

En esta primera parte se pide copiar los códigos de prueba que aparecen en el guion de prácticas y comentar cómo funcionan y qué hacen.

Un problema que he tenido es a la hora de usar el script proporcionado, ya que dentro del mismo se lanzan secuencialmente el servidor y los clientes, esto produce que se lancen nada más que el servidor y que los clientes no se lancen. Para ello he separado la parte de la invocación de los clientes en otro script para poder lanzar ambos en dos terminales distintas.

Además, por algún motivo que desconozco, a veces al lanzar el script del servidor me da problemas al ejecutarlo, entonces simplemente, ejecuto los comandos por separado directamente en la terminal.

## 2.1 EJEMPLO 1

### 2.1.1 Descripción

Este ejemplo lo que realiza en el lado del cliente es llamar al método remoto *escribir_mensaje* con el identificador pasado por consola del cliente.

En el lado del servidor al ejecutar el método hay dos opciones, si el identificador es igual a 0 entonces duerme durante 5000 ms (5 segundos), en otro caso simplemente indica que ha recibido la petición y muestra el identificador del cliente.

```
^Calissea@ALISSEA:/mnt/c/Users/carme/OneDrive/Documentos/Universidad/3IngInf/SEGUNDO_CUATRI/DSD/Prácticas/DSD_UGR/P3/Ejemp
los/Ejemplo_1$ java -cp . -Djava.rmi.server.codebase=file:./ -Djava.rmi.server.hostname=localhost -Djava.security.policy=server.policy Ejemplo
Ejemplo bound
Recibida peticion de proceso: 0
Empezamos a dormir
Terminamos de dormir

Hebra 0
Recibida peticion de proceso: 3

Hebra 3
```

```
alissea@ALISSEA:/mnt/c/Users/carme/OneDrive/Documentos/Universidad/3IngInf/SEGUNDO_CUATRI/DSD/Prácticas/DSD_UGR/P3/Ejemp
los/Ejemplo_1$ bash ./script_cliente.sh

Lanzando el primer cliente

Buscando el objeto remoto
Invocando el objeto remoto
------------------------------------------------------------------------


Lanzando el segundo cliente

Buscando el objeto remoto
Invocando el objeto remoto
```

Como se puede observar el primer cliente que se lanza tiene el identificador 0, por lo que se pone a dormir y se indica en el servidor. El siguiente, que tiene identificador 3, simplemente hace que se muestre en el servidor su información, pero no se duerme.

### 2.1.2 Implementación

Para la implementación de este sistema primero es necesario realizar la interfaz remota del servidor que va a exportar los métodos a los clientes. En este caso solo se exporta el método *escribir_mensaje*.

En la parte del servidor se crea una clase que implementa los métodos de la interfaz remota “Ejemplo_I'”

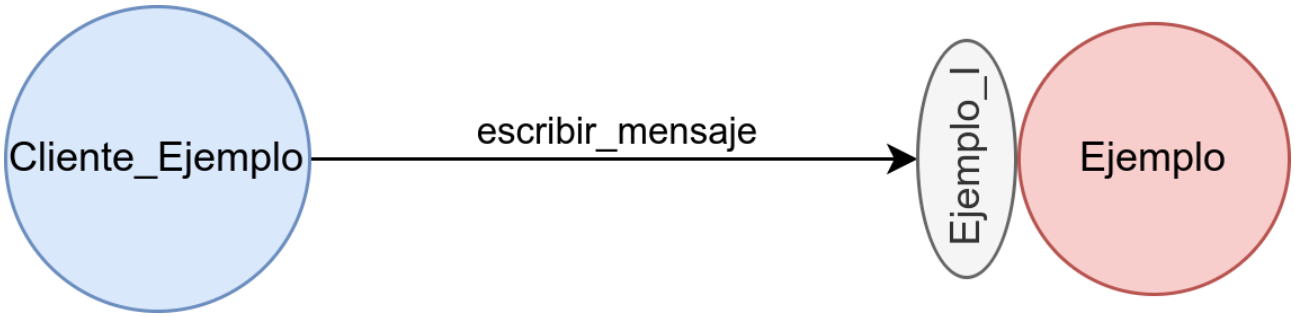
El método imprime la petición del proceso en la pantalla del servidor. Además, comprueba si el número de proceso es 0, si es así, le dice al cliente que llame al método remoto para dormir durante 5 segundos. Después, independientemente de si el proceso es 0 o no, el identificador del cliente se muestra en la pantalla remota.

En el main del servidor se activa el gestor de seguridad, que es exactamente igual para el cliente, crea una instancia de del objeto remoto y luego se exporta y se registra en “rmiregistry” para que los clientes puedan encontrarlo.

En el main del cliente se activa el gestor de seguridad también, obtiene el “rmiregistry” y obtiene a partir del registro el stub del objeto remoto para poder lanzar el método anteriormente mencionado.

### 2.1.3 Funcionamiento

El diagrama del sistema (sin incluir rmiregistry ni stubs, solo los propios objetos y clases) es el siguiente:




Se puede ver que el cliente llama al método escribir_mensaje exportado por la interfaz remota del servidor, que se llama Ejemplo.

## 2.2 EJEMPLO 2

### 2.2.1 Descripción

En este ejemplo se realiza algo similar al anterior, pero esta vez el cliente lanza un número de hebras que se pasan por consola al programa.

**¿Se entrelazan los mensajes?** Sí existe entrelazamiento. Tras varias ejecuciones se puede observar cómo se produce un entrelazamiento de mensajes de salida en el servidor en el que se muestra que entran varias hebras a la vez al método antes de que la primera haya mostrado el mensaje de salida. Puede que esto sea algo no deseado y que siempre se desee que las hebras siguientes esperen a que la hebra que va delante reciba la respuesta del servidor para que siempre haya como mucho una hebra en el método remoto.

```
Entra Hebra Cliente 2

Entra Hebra Cliente 7
Sale Hebra Cliente 2

Entra Hebra Cliente 4
Sale Hebra Cliente 4

Entra Hebra Cliente 10
Empezamos a dormir
Sale Hebra Cliente 7

Entra Hebra Cliente 3

Entra Hebra Cliente 8

Entra Hebra Cliente 9
```

Otro entrelazamiento menor que ocurre es que las hebras no realizan las peticiones de manera ordenada al servidor. Es decir, en la impresión que realiza el servidor por consola se puede observar que no siguen un orden ascendente los identificadores de los clientes. Esto es algo esperable porque realmente las hebras del cliente no llaman al método remoto en el mismo orden en que se crean (las hebras)

**¿Qué hacen las demás hebras?** Las demás hebras no esperan a que la hebra de delante reciba la respuesta del método remoto haciendo que el servidor esté ejecutando el mismo método concurrentemente produciendo así, como se ha dicho anteriormente, entrelazamientos.

**¿Qué ocurre con las hebras cuyo nombre acaba en 0?** Las hebras que acaban en 0, al igual que en el ejemplo 1, se ponen a dormir durante 5000 ms (5 segundos) al llamar al método remoto.

### 2.2.2 Implementación

La implementación de la parte del servidor es exactamente igual que en el ejemplo 1, salvo por algún ligero cambio del mensaje que se muestra por pantalla en el servidor.

En la parte del cliente se realiza prácticamente lo mismo, pero se encapsula en el método *run*, ya que son hebras, luego desde el main se crea un array de hebras y se lanzan con *start*.

### *2.2.3* Uso de *synchronized*

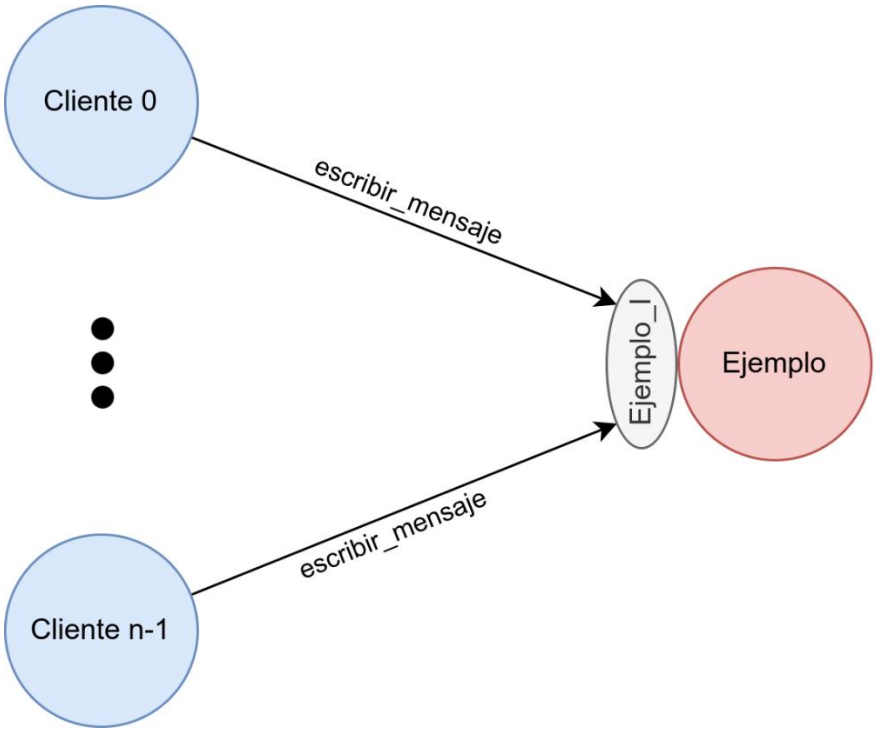
Cuando se añade la palabra clave ***synchronized*** al método remoto del servidor ya no se produce entrelazamientos. Sin embargo, las hebras que acaban en 0 se ponen a dormir y bloquea al servidor para recibir más peticiones a métodos *synchronized* produciendo así posibles esperas innecesarias.

```
Lanzando el servidor
Ejemplo bound

Entra Hebra Cliente 3
Sale Hebra Cliente 3

Entra Hebra Cliente 6
Sale Hebra Cliente 6

Entra Hebra Cliente 1
Sale Hebra Cliente 1

Entra Hebra Cliente 8
Sale Hebra Cliente 8

Entra Hebra Cliente 0
Empezamos a dormir
Terminamos de dormir
Sale Hebra Cliente 0

Entra Hebra Cliente 2
Sale Hebra Cliente 2
```

Como se puede observar en la imagen de la derecha, la hebra 0 empieza a dormir y las demás hebras no pueden ejecutar el método hasta que se cumpla el periodo de dormir. Esto arreglaría el entrelazamiento, pero a costa de introducir esperas, ya que ahora las demás hebras tienen que esperar a la ejecución del método remoto.

Como conclusión se puede decir que la principal diferencia entre no usar *synchronized* y sí usarlo es que en el primero las hebras no necesitan esperar a que salgan las de delante para que su petición pueda ser procesada. Mientras que en el segundo caso solo una hebra puede acceder a métodos *synchronized* y las demás tienen que esperar su finalización para que puedan acceder.

Por último, el diagrama de la relación entre objetos es el siguiente (como antes, obviando stubs y rmiregistry):



## 2.3 EJEMPLO 3

### 2.3.1 Descripción

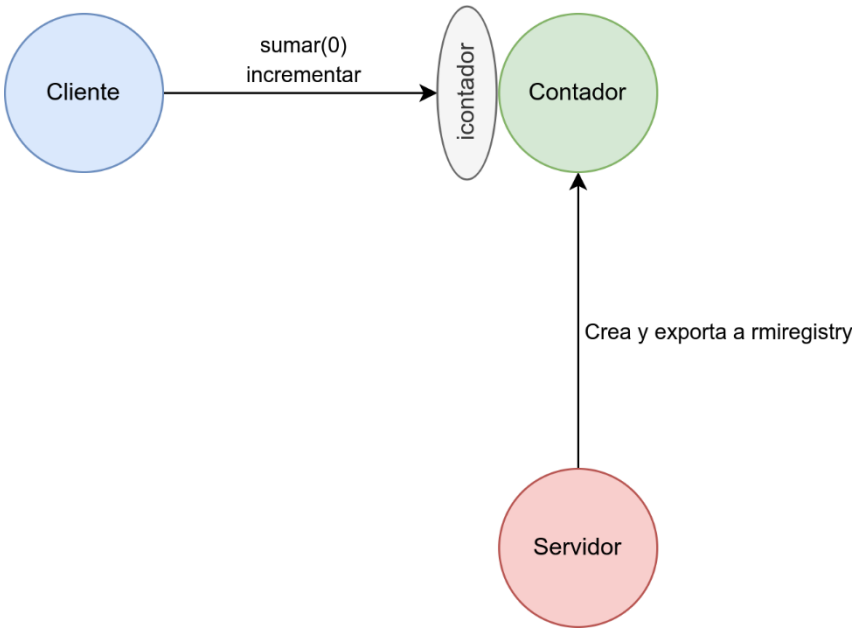
En este ejemplo se separa el propio objeto remoto del servidor. Para ello se crea la interfaz remota como en los ejemplos anteriores, pero esta vez se implementa en una clase aparte sin tener un main asociado.

El main se encuentra en otra clase servidor que se encarga como en los demás ejemplos de crear la instancia del objeto remoto y exportarlo.

Por la parte del cliente realiza la misma tarea que en los ejemplos anteriores, pero esta vez una vez creado el stub pone el contador a 0, ya que es posible que otros procesos hayan modificado el atributo "suma". Una vez realizado esto toma el tiempo de entrada y realiza 1000 llamadas al método remoto *incrementar*, cuando acaba de realizar las llamadas toma el tiempo de salida y muestra por consola el tiempo que ha tardado.

```
Poniendo contador a 0
Incrementando...
Media de las RMI realizadas = 0.088 msegs
RMI realizadas = 1000
------------------------------------------------------------
```

Por último, como en los ejemplos anteriores, un diagrama con las relaciones entre objetos:




## 2.4 DIFERENCIAS ENTRE LOS EJEMPLOS

La primera diferencia es que en los ejemplos 1 y 3 se usan solamente los procesos como tal en los que no existe nada más que 1 hebra, la que ejecuta el main. Mientras que en el ejemplo 2 el cliente lanza varias hebras que acceden concurrentemente a un método remoto.

La segunda diferencia es en la implementación de la interfaz remota, en la que en los ejemplos 1 y 2 se realiza en la propia clase del servidor (donde se encuentra el main) mientras que en el ejemplo 3 se realiza en una clase aparte. Esto permite separar la implementación del objeto remoto del propio código del servidor.

Otra diferencia se encuentra en el uso de rmiregistry, el servidor en los ejemplos 1 y 2 supone que se ha lanzado el rmiregistry en la terminal y solo se encarga en exportar el objeto remoto, mientras que en el ejemplo 3 es el propio servidor el que crea el rmiregistry y le asigna un puerto mediante el método createRegistry. Es más, si se intenta lanzar rmiregistry primero y luego este servidor, va a dar error, por lo que es necesario matar a todos los procesos rmiregistry.

# 3 SEGUNDA PARTE. SERVIDOR DE DONACIONES

## 3.1 DESCRIPCIÓN

El ejercicio consiste en desarrollar en Java RMI un sistema cliente-servidor en el cual los clientes pueden donar una cantidad de dinero a un servidor. Este sistema tiene las siguientes características:

- Existen diferentes clientes que se conectan a una réplica. Pueden crearse infinitos clientes y se les asignará la réplica con menor número de clientes asignados. El cliente dona una cantidad de dinero a la réplica asignada.
- Existen varias réplicas del servidor (en nuestro ejemplo usaremos solo 2, pero se podrían crear más), para ello se instancian N objetos remotos de dos tipos: Donaciones y Replicas. Estos dos objetos implementan, respectivamente, las interfaces Donaciones_I para interacciones Cliente-Servidor y Replicas_I para interacciones Servidor-Servidor. Por lo que cada Replica está compuesta por un objeto de tipo Donacion y un objeto de tipo Replica.
- El cliente utiliza la interfaz Donación, que atenderá las peticiones y llamará a los métodos de la interfaz Replicas. Esta interfaz se comunicará con el resto de réplicas en caso de necesitar datos de éstas.
- Cada objeto Donación tiene una referencia al objeto Replica correspondiente
- Cada réplica almacena:
    - Un Map con el id de cada cliente y la cantidad donada. Con esto se podrá comprobar si un cliente está registrado y además si ha donado algo.
    - Un Map con el id de cada cliente y la contraseña vinculada a la cuenta
    - El identificador de la réplica, un número del 0 al N
    - El número total de réplicas del sistema
    - El subtotal de dinero donado en esa réplica concreta

  Esta interfaz permite operaciones entre servidores, obteniendo por ejemplo el subtotal de donaciones, que sólo pueden conocer las réplicas. Además, es dónde se hace efectivo el registro del cliente y las donaciones, pues Donaciones solo es un intermediario.

Se ha añadido la funcionalidad de poder elegir entre Iniciar Sesión con un cliente ya existente o crear uno nuevo. Además, los clientes tienen una contraseña con la cual acceden a su cuenta. También se ha implementad una función para modificar la contraseña.

Además, he creado el TDA Pair, para poder manejar conjunto de valores de forma más fácil en algunas funciones. Lo he creado en vez de implementarlo, pues me daba algunos errores al poner **implements javaxf.utils.pair.**

Para la asignación del servidor, el cliente por defecto “conecta con el servidor 0”, una vez se registra o inicia sesión, se comprueba en la réplica cuál es la réplica asignada, la cual se devuelve y ya se cambia la variable local para que haga las operaciones sobre la réplica asignada.

```
            System.out.println(x:"Introduzca su ID de usuario: ");
            idUsuario = scanner.nextInt();
            scanner.nextLine();

            System.out.println(x:"Introduzca su contraseña: ");
            pswd = scanner.nextLine();

            Donaciones_I local = (Donaciones_I) registry.lookup("Replica" + 0 + "Donaciones");
            Pair<Integer, Boolean> respuesta;
```

```
@Override
public Pair<Integer,Boolean> registroCliente(int id, String pswd) throws RemoteException, NotBoundException {

    if (estaRegistrado(id)) {
        return new Pair<>(idReplica, second:false);
    }

    int replicaRegistro = idReplica;
    for (int i = 0; i < this.numReplicas; i++) {
        if (i!= this.idReplica) {
            Replicas_I replica = (Replicas_I) registry.lookup("Replica" + i);
            if(replica.estaRegistrado(id))
            {
                return new Pair<>(i, second:false);
            }
            if (replica.getNumRegistrados() < this.getNumRegistrados()) {
                replicaRegistro = i;
            }
        }
    }

    if (replicaRegistro!= idReplica) {
        Replicas_I replica = (Replicas_I) registry.lookup("Replica" + replicaRegistro);
        replica.registrar(id, pswd);
    } else {
        registrar(id, pswd);
    }


    return new Pair<Integer,Boolean>(replicaRegistro, second:true);
}
```

```
            local = (Donaciones_I) registry.lookup("Replica" + replica + "Donaciones");

            System.out.println("ID: " + idUsuario);
            System.out.println("Servidor asignado: " + replica);
```

## 3.2 EJECUCIÓN

En las imágenes se muestra un ejemplo de ejecución en el que se crean dos servidores, se dona diferentes cantidades con cada uno y se muestra por pantalla las diferentes funciones. El proceso para usar el programa, como en las demás prácticas se basa en un menú y en elecciones.

```
alissea@ALISSEA:/mnt/c/Users/carme/OneDrive/Documentos/Universidad/3IngInf/SEGUNDO_CUATRI/DSD/Prácticas/DSD_UGR/P3/Donaciones$ bash ./script_cliente.sh

Lanzando el cliente

=====BIENVENIDO=====
        1. Nuevo Usuario
        2. Iniciar Sesion
        0. Salir
1
Introduzca su ID de usuario:
0
Introduzca su contraseña:
0
ID: 0
Servidor asignado: 0
=====Siguiente Accion=====
        1. Donar
        2. Consultar total donado
        3. Consultar total donado cliente
        4. Consultar listado de donantes
        5. Modificar Contraseña
        0. Salir
1
Introduzca la cantidad que desea donar
10
=====Siguiente Accion=====
        1. Donar
        2. Consultar total donado
        3. Consultar total donado cliente
        4. Consultar listado de donantes
        5. Modificar Contraseña
        0. Salir
1
Introduzca la cantidad que desea donar
5
=====Siguiente Accion=====
        1. Donar
        2. Consultar total donado
        3. Consultar total donado cliente
        4. Consultar listado de donantes
        5. Modificar Contraseña
        0. Salir
3
Recaudaciones individuales: 15.0
```

```
alissea@ALISSEA:/mnt/c/Users/carme/OneDrive/Documentos/Universidad/3IngInf/SEGUNDO_CUATRI/DSD/Practicas/DSD_UGR/P3/Donaciones$ bash ./script_cliente.sh

Lanzando el cliente

=====BIENVENIDO=====
        1. Nuevo Usuario
        2. Iniciar Sesion
        0. Salir
1
Introduzca su ID de usuario:
1
Introduzca su contraseña:
1
ID: 1
Servidor asignado: 1
=====Siguiente Accion=====
        1. Donar
        2. Consultar total donado
        3. Consultar total donado cliente
        4. Consultar listado de donantes
        5. Modificar Contraseña
        0. Salir
1
Introduzca la cantidad que desea donar
10
=====Siguiente Accion=====
        1. Donar
        2. Consultar total donado
        3. Consultar total donado cliente
        4. Consultar listado de donantes
        5. Modificar Contraseña
        0. Salir
2
Recaudaciones totales: 25.0

=====Siguiente Accion=====
        1. Donar
        2. Consultar total donado
        3. Consultar total donado cliente
        4. Consultar listado de donantes
        5. Modificar Contraseña
        0. Salir
4
Listado donantes: [0, 1]
=====Siguiente Accion=====
        1. Donar
```