
Práctica 3: Java RMI - Marina Muñoz Cano



Índice

Índice	2
Pruebas ejemplos del guión	3
Servidor Replicado	3
Programa Principal Cliente	4
Programa Principal Servidor	5
Interfaz Cliente-Servidor	6
Interfaz Servidor-Servidor	6
Implementación Exclusión Mutua en Anillo	8
Implementación N réplicas	12
Scripts Ejecución	12

Pruebas ejemplos del guión

Ejemplo 1:

- Cliente: Activa el gestor de seguridad, busca el objeto remoto y lo invoca.
- Servidor: si el cliente invoca el proceso 0, lo duerme 5 segundos, en caso contrario, no lo duerme. En ambos casos escribe en pantalla el proceso que se ha llamado y el número de hebra, aunque en realidad solo hay una hebra por cliente y se lanza varias veces el programa Cliente. También imprime cuando empieza y termina de dormir al proceso 0.

Ejemplo 2:

- Cliente: Lanza tantas hebras como se le pase como argumento e invoca con cada hebra el proceso escribir mensaje del servidor.
- Servidor: Misma función que en el ejemplo anterior, pero esta vez duerme a todos los procesos acabados en 0. El resto de hebras se ejecutan normalmente aunque haya una acabada en 0 que esté durmiendo, no se bloquean. Los mensajes aparecen entrelazados porque mientras la hebra acabada en duerme se siguen ejecutando otras, no hay exclusión mutua. Si añadimos synchronized al método escribir mensaje, esto no ocurre, ya que no se entrelazan los mensajes, porque se garantiza la exclusión mutua en el acceso al método por parte de las hebras.

Ejemplo 3:

- Cliente: Se inicializa el contador a 0 e invoca al método remoto incrementar del contador 1000 veces. Calcula el tiempo en el que realiza la operación e imprime los resultados.
- Servidor: Crea el rmiregistry y el objeto micontador de tipo contador para recibir las invocaciones remotas.

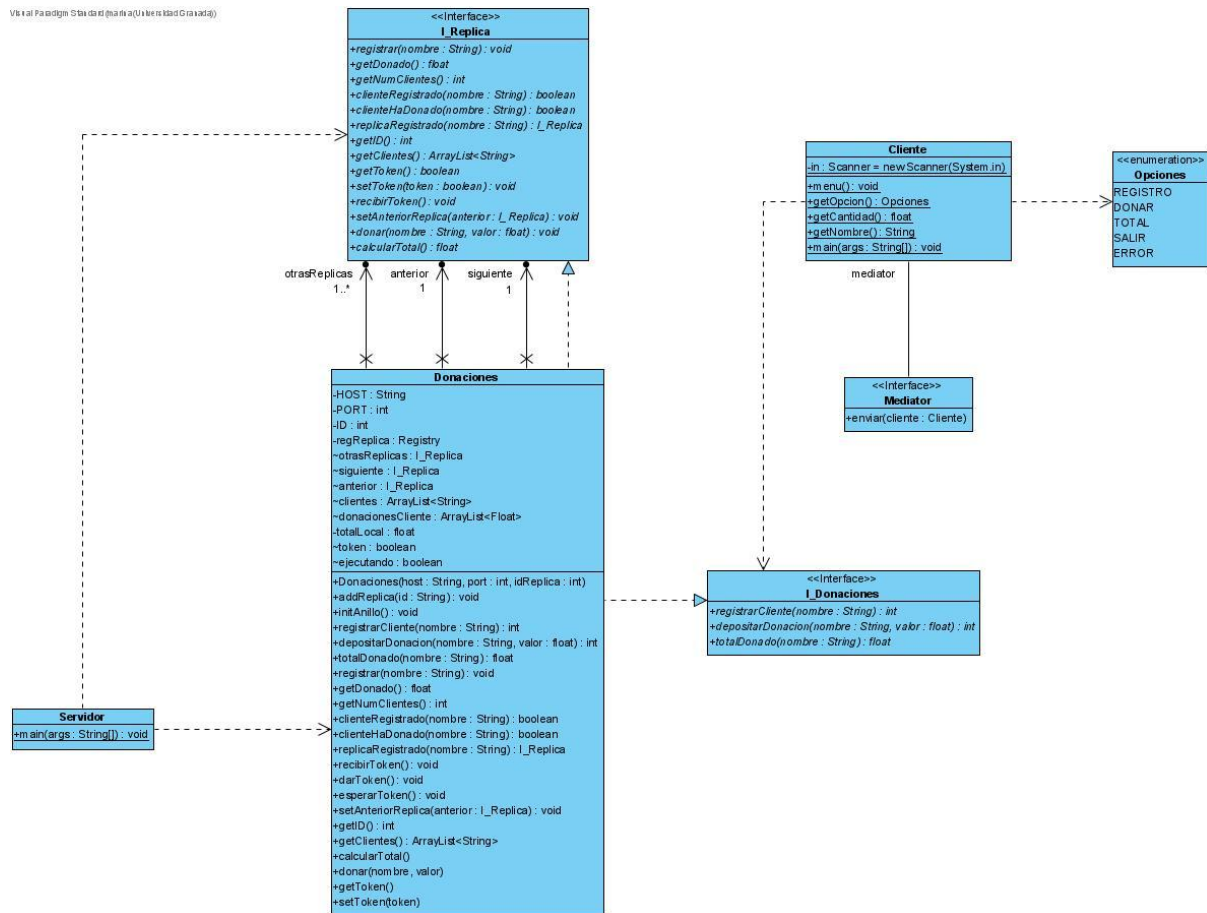
Servidor Replicado

Para implementar el ejercicio, comencé por definir dos interfaces:

- I_Donaciones que sería la que contendría las operaciones que invoca el cliente para conectarse al servidor.
- I_Replica que contiene las operaciones que permiten la comunicación entre réplicas.

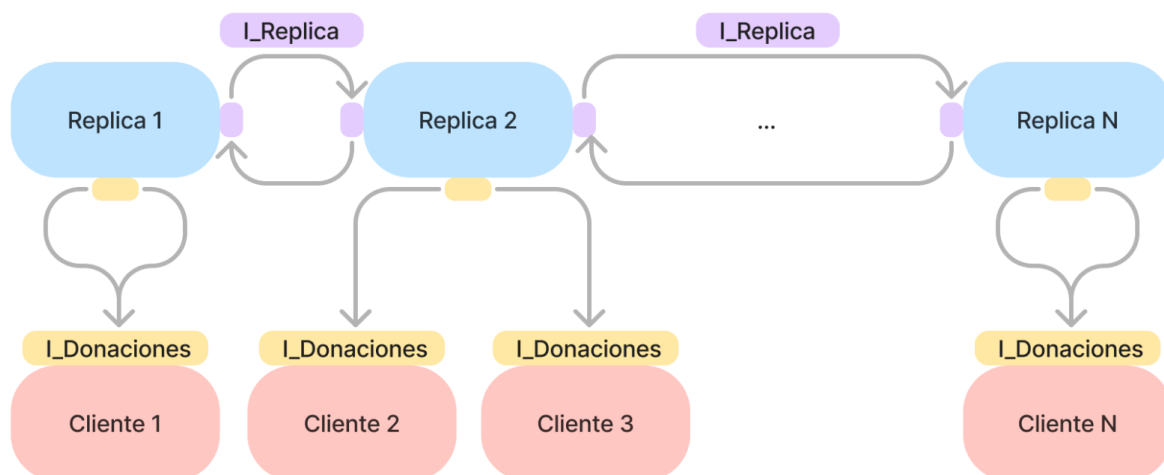
Tras esto, defino la clase Donaciones, que implementa ambas interfaces y será el objeto remoto que exportará el servidor. Además creo las clases cliente y servidor que albergaran los programas principales de ambas partes. Por último, para definir el menú interactivo, creo un enumerado de operaciones para poder invocarlas según la opción seleccionada del menú.

Diagrama de Clases:



Además de la parte básica, he implementado el algoritmo de Exclusión mutua en anillo y la posibilidad de lanzar más de dos réplicas.

Diagrama Ejemplo comunicación:



Programa Principal Cliente

El main del cliente acepta los siguientes parámetros: [Host] [Port] [idReplica].

- Host: dirección IP del servidor.
- Puerto en el que se encuentra el rmiregistry.
- idReplica: Número de la réplica a la que se quiere acceder.

Una vez localizado el rmiregistry y la réplica en la que va a trabajar el cliente, obtengo el nombre de usuario del cliente que ha lanzado el programa. Tras esto, muestro el menú de opciones disponibles y le pido al usuario que introduzca una opción. Según la opción elegida, invoco al método del objeto remoto correspondiente, paro la ejecución del programa o muestro un mensaje de error.

```
switch (opcion) {
    case REGISTRO:
        try {
            System.out.println("Cliente " + nombre + " registrado en replica " + donar.registrarCliente(nombre));
        } catch (RemoteException e) {
            System.out.println(e.getMessage());
        }
        break;

    case DONAR:
        float cantidad = getCantidad();
        try {
            System.out.println("Cliente " + nombre + " dona en replica " + donar.depositarDonacion(nombre, cantidad) + " " + cantidad + "€.");
        } catch (RemoteException e) {
            System.out.println(e.getMessage());
        }
        break;

    case TOTAL:
        try {
            System.out.println("Total donado = " + donar.totalDonado(nombre));
        } catch (RemoteException e) {
            System.out.println(e.getMessage());
        }
        break;

    case SALIR:
        break;
    case ERROR:
        System.out.println("Seleccione una opción correcta");
        break;
}
```

Programa Principal Servidor

El programa principal del servidor acepta tres parámetros: [Host] [Port] [nReplicas].

- Host: dirección IP del servidor.
- Puerto en el que se crea el rmiregistry.
- nReplicas: Número de Réplicas a lanzar.

Todas las operaciones se realizan a través del mismo registro, en el mismo puerto.

En el main del servidor creo el rmiregistry, a continuación lanzo tantas réplicas como se ha especificado por parámetro y hago el rebind de cada réplica con el nombre "Réplica[ID]" donde ID es un número entre 1 y el número total de réplicas.

Tras esto, añado a cada réplica el resto de réplicas mediante el método addReplica(nombreReplica). En este método se realiza la búsqueda del objeto Remoto

mediante lookup, una vez encontrada, la añade al vector de otras réplicas de la réplica en cuestión. De esta forma se guardan todas las réplicas para poder hacer consultas remotas a éstas.

Tras inicializar el vector de réplicas de cada una, para cada réplica inicializo el anillo, lo que significa que le indico cuál es la réplica anterior u la siguiente para poder circular el token.

Por último, hago que la primera réplica sea la que comienza teniendo el token.

Interfaz Cliente-Servidor

Las operaciones que puede realizar el cliente son:

- Registrarse.
- Donar.
- Ver el total donado en todas las réplicas.

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

public interface I_Donaciones extends Remote {
    int registrarCliente(String nombre) throws RemoteException;

    int depositarDonacion(String nombre, float valor) throws RemoteException;

    float totalDonado(String nombre) throws RemoteException;
}
```

Cuando el cliente invoca una de estas operaciones del servidor, las réplicas gestionan los posibles casos de error y si es correcto, invocan al método de la interfaz de réplicas que realiza la operación con la interfaz de réplica correspondiente en la que está registrado (o se va a registrar) el cliente.

Todas estas operaciones se realizan en exclusión mutua mediante el algoritmo del anillo.

Interfaz Servidor-Servidor

Contiene todas las operaciones que invocan unas réplicas sobre otras:

```

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.ArrayList;

public interface I_Replica extends Remote {
    // Se registra un cliente en la replica
    void registrar(String nombre) throws RemoteException;

    // Registra la donacion de un cliente a la replica.
    void donar(String nombre, float valor) throws RemoteException;

    //Devuelve el total donado local de la replica.
    float getDonado() throws RemoteException;

    // Calcula el total de donaciones de todas las replicas.
    public float calcularTotal() throws RemoteException;

    //Devuelve el numero de clientes están registrados en la réplica
    int getNumClientes() throws RemoteException;

    /**
     * Indica si el cliente correspondiente con el nombre de usuario pasado
     * como parámetro está registrado en la replica.
     */
    public boolean clienteRegistrado(String nombre) throws RemoteException;

    /**
     * Indica si el cliente correspondiente con el nombre de usuario pasado
     * como parámetro ha donado en la replica.
     */
    public boolean clienteHaDonado(String nombre) throws RemoteException;

    // Devuelve el ID de la replica.
    int getID() throws RemoteException;

    // Devuelve el array de clientes de la replica
    public ArrayList<String> getClientes() throws RemoteException;

    // Devuelve si la replica tiene o no el token
    boolean tieneToken() throws RemoteException;

    // Cambia el valor del token por el pasado como parámetro.
    void setToken(boolean token) throws RemoteException;

    /**
     * Invocado por la replica anterior para pedir el token.
     * Si la replica que recibe la invocación tiene el token y no esta
     * ejecutando ningun metodo invocado por un cliente, le da el token
     * a su siguiente replica. Si no lo tiene, le pide el token a la anterior.
     */
    void recibirToken() throws RemoteException;

    // Modifica el valor de la replica anterior.
    public void setAnteriorReplica(I_Replica anterior) throws RemoteException
;
}

```

Otros Métodos de las réplicas

- Constructor: inicializa los atributos y busca el registro con el host y el puerto que se le pasan como parámetro.
- addReplica: Busca la réplica en el registro mediante el nombre pasado como argumento y la añade al array de otras replicas.
- replica Registrado: devuelve la réplica en la que está registrado un determinado cliente, para poder invocar a las operaciones de ésta en caso de que el cliente se conecte a una réplica que no es en la que está registrado.
- initAnillo: inicializa el valor de la réplica siguiente y anterior para implementar la exclusión mutua en Anillo. La siguiente se calcula como la réplica del array de otras réplicas que se encuentra en la posición ID+1, mientras que para inicializar anterior, le manda a la siguiente réplica que ella es su anterior, de esta forma todas reciben un mensaje de su anterior.
- darToken: si la réplica tiene el token, se lo da a su siguiente.
- esperarToken: mientras que no tiene el token espera a que su anterior se lo mande.

Implementación Exclusión Mutua en Anillo

Para implementar la exclusión mutua en anillo defino los siguientes atributos en las réplicas:

```
// Replicas anterior y siguiente para EM
I_Replica siguiente, anterior;
// Indica si tiene o no el Token
boolean token;
//Indica si esta o no ejecutando una sección critica
boolean ejecutando;
```

En cada método que incluya una sección crítica, realizo los siguientes pasos:

```
//Espera a tener el token
esperarToken();

//Inicio SC
ejecutando = true;

//... Código SC

//Fin SC
ejecutando = false;

// Da el token a la siguiente replica
darToken();
```


Espera para recibir el token: Si no tiene el token, lo pide a su anterior y se bloquea hasta que lo recibe.

```
public void esperarToken() throws RemoteException{
    System.out.println("REPLICA " + getID() + " Esperando el token.");

    while(!token)
        anterior.recibirToken();
}
```

Tras esto, indica que se está ejecutando la sección crítica, para que si le piden el token no lo dé hasta que no termine de ejecutar. Cuando termina indica que ha terminado poniendo ejecutando a false y le da el token a su réplica siguiente.

```
public void darToken() throws RemoteException{

    if(token){
        System.out.println("REPLICA " + getID() + " da el token.");
        System.out.println("REPLICA " + siguiente.getID() + " Tiene el token.");

        siguiente.setToken(true);
        this.setToken(false);
    }
}
```

Cuando una réplica recibe una petición de token de su siguiente, comprueba que tiene el token y no esté ejecutando ninguna sección crítica, si es así se lo da. En caso de que no lo tenga, lo pide a su anterior, si lo tiene pero está ejecutando una sección crítica no hace nada.

```
public void recibirToken() throws RemoteException{
    if(token && !ejecutando)
        darToken(); // Si tiene el token y no esta ejecutando, se lo da
    else if(token == false)
        esperarToken();
    // Si no tiene el token, lo pide para darselo a la replica que se lo ha pedido
    //Si está ejecutando no hace nada.
}
```

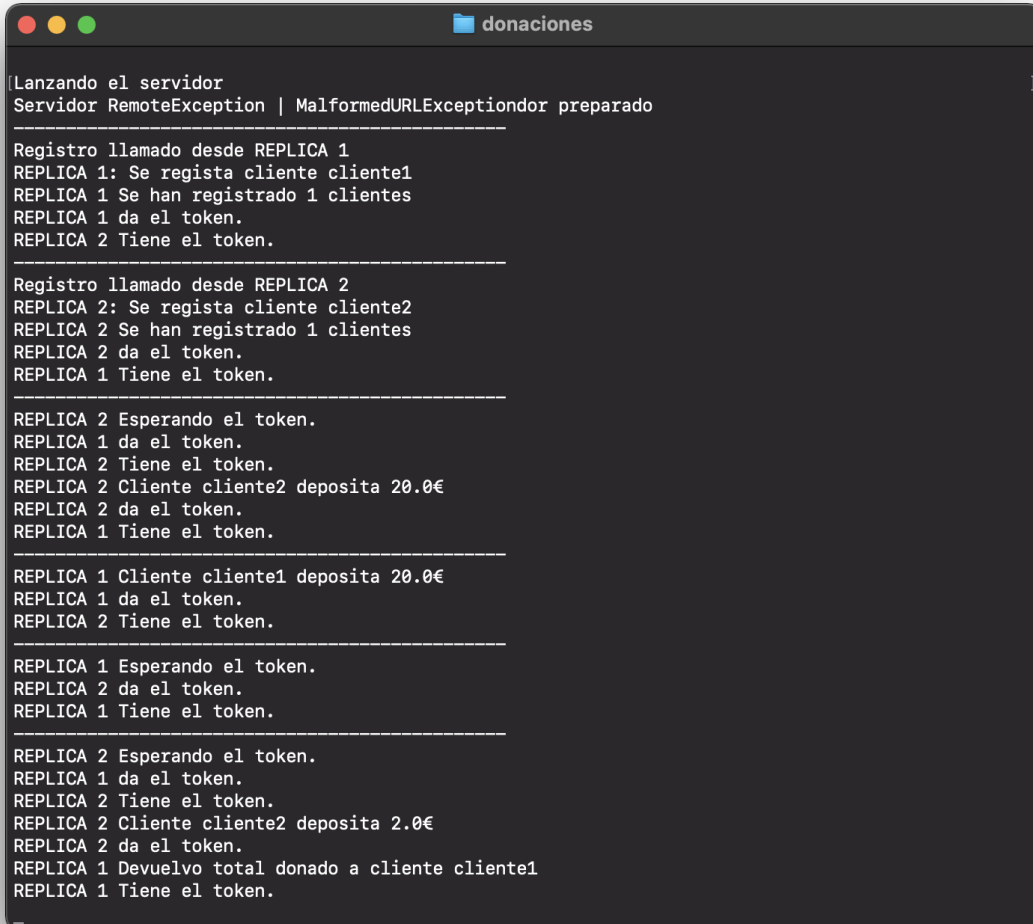
Este proceso lo hago en las tres operaciones principales que invocan los clientes, ya que cabe la posibilidad de que haya varios clientes accediendo a la vez a los datos de una misma réplica.

Para comprobar que funciona, pongo un sleep de 5 segundos en el método total donación antes de devolver el total a un cliente:

Lanzo el servidor con dos réplicas y dos clientes, cada uno se registra en una réplica: cliente1 en la 1, cliente2 en la 2.

Tras registrarlos y hacer una donación inicial cada uno de 20€, cliente1 invoca al método total donado, a continuación antes de que termine, cliente2 intenta donar en la réplica 2. En las salidas se observa que hasta que el cliente1 no ha terminado su operación, la réplica 1 no le da el token a la réplica 2 y ésta segunda se queda esperando hasta que la 1 se lo da para registrar la donación del segundo cliente.

Salida Servidor:



```
[Lanzando el servidor
Servidor RemoteException | MalformedURLExceptiononador preparado

-----
Registro llamado desde REPLICA 1
REPLICA 1: Se registra cliente cliente1
REPLICA 1 Se han registrado 1 clientes
REPLICA 1 da el token.
REPLICA 2 Tiene el token.

-----
Registro llamado desde REPLICA 2
REPLICA 2: Se registra cliente cliente2
REPLICA 2 Se han registrado 1 clientes
REPLICA 2 da el token.
REPLICA 1 Tiene el token.

-----
REPLICA 2 Esperando el token.
REPLICA 1 da el token.
REPLICA 2 Tiene el token.
REPLICA 2 Cliente cliente2 deposita 20.0€
REPLICA 2 da el token.
REPLICA 1 Tiene el token.

-----
REPLICA 1 Cliente cliente1 deposita 20.0€
REPLICA 1 da el token.
REPLICA 2 Tiene el token.

-----
REPLICA 1 Esperando el token.
REPLICA 2 da el token.
REPLICA 1 Tiene el token.

-----
REPLICA 2 Esperando el token.
REPLICA 1 da el token.
REPLICA 2 Tiene el token.
REPLICA 2 Cliente cliente2 deposita 2.0€
REPLICA 2 da el token.
REPLICA 1 Devuelvo total donado a cliente cliente1
REPLICA 1 Tiene el token.
```

Se puede ver que la réplica 2 espera el token hasta que la 1 se lo da para depositar los 2€. El mensaje de de Devuelvo total donado aparece debajo porque está fuera de la sección crítica y se ejecuta después. En el código entregado ya está arreglado ésto ultimo.

Salida Cliente 1:

```
donaciones

¿Qué operación desea realizar?
0: Registro
1: Donar
2: Ver Total Donado
3: Salir
1

¿Cuántos € desea donar?
20

Cliente cliente1 dona en replica 1 20.0€.

¿Qué operación desea realizar?
0: Registro
1: Donar
2: Ver Total Donado
3: Salir
2

Total donado = 40.0

¿Qué operación desea realizar?
0: Registro
1: Donar
2: Ver Total Donado
3: Salir
```

Salida Cliente 2:

```
donaciones

2: Ver Total Donado
3: Salir
1

¿Cuántos € desea donar?
20

Cliente cliente2 dona en replica 2 20.0€.

¿Qué operación desea realizar?
0: Registro
1: Donar
2: Ver Total Donado
3: Salir
1

¿Cuántos € desea donar?
2

Cliente cliente2 dona en replica 2 2.0€.

¿Qué operación desea realizar?
0: Registro
1: Donar
2: Ver Total Donado
3: Salir
```

Implementación N réplicas

Para tener más de dos réplicas, en cada réplica guardo un vector con el resto de réplicas del servidor. Realizo las mismas operaciones que haría para dos réplicas, pero cada vez que necesito obtener información de las otras réplicas recorro el vector completo.

En el programa principal del servidor obtengo por argumento el número de réplicas que se quieren lanzar y las lanzo una a una guardandolas en un vector, para posteriormente inicializar el vector de otras réplicas, la réplica anterior y la siguiente de cada una de ellas.

```
for (int i = 0; i < nReplicas ; i++) {
    Donaciones replica = new Donaciones(host, port, i+1);

    int id = i+1;
    Naming.rebind("Replica" + id, replica);
    replicas.add(replica);
}

for (int i = 0; i < replicas.size(); i++) {
    for (int j = 0; j < replicas.size(); j++) {
        int id = j+1;
        if( i!=j) {
            replicas.get(i).addReplica("Replica" + id);
        }
    }
}

for (Donaciones replica : replicas) {
    replica.initAnillo();
}
```

Scripts Ejecución

En el zip entregado incluyo los scripts de compilación y ejecución. Para ejecutar utilizar el script servidor de la siguiente forma:

```
./server.sh [numero de replicas]
```

Este script compila y ejecuta el servidor, para lanzar un cliente ejecutar:

```
./cliente.sh [id replica a la que quiere conectarse entre 1 y el total de réplicas]
```