

Memoria Práctica 3: RMI

30 de abril del 2021

INTRODUCCIÓN

Esta memoria se dividirá en dos partes, en la primera implementaremos los ejemplos proporcionados en el guión, probaremos y explicaremos su funcionamiento además de comentar las diferencias existentes entre estos.

En la segunda parte, implementaremos el servidor replicado propuesto pero añadiéndole una serie de funcionalidades extra como el poder utilizar más de dos réplicas, implementar un algoritmo de exclusión mutua y poder ejecutar cliente y servidor en distintos ordenadores.

ÍNDICE

INTRODUCCIÓN	1
ÍNDICE	1
PARTE 1: Implementación de ejemplos	2
Ejemplo 1	2
Problemas encontrados y ejecución	2
Análisis de su funcionamiento	4
Ejemplo 2	6
Principales diferencias	6
Análisis de su funcionamiento	6
Ejecución	7
Uso del modificador “synchronized”	8
Ejemplo 3	9
Principales diferencias	9
Análisis de su funcionamiento	9
Ejecución	10
PARTE 2: Servidor replicado	11
Introducción	11
Comunicación entre réplicas	11
Explicación de métodos utilizados	12
Servidor	13

Cliente	14
Exclusión mutua basado en anillo	15
Funcionamiento en distintos ordenadores y sus limitaciones	15
Gestión de excepciones	17
Instrucciones de uso	18
Ejemplo de uso	19

PARTE 1: Implementación de ejemplos

Ejemplo 1

Comencemos observando el funcionamiento del ejemplo para, a continuación, entrar más en detalle en cómo todo está implementado.

Para ello, copiamos el código proporcionado en el guión además del script de bash para facilitar su ejecución.

Problemas encontrados y ejecución

Desafortunadamente, tuvimos que hacer frente a una serie de problemas antes de conseguir ejecutar el ejemplo correctamente:

- **Javac no presente en el PATH:** Para solucionar esto se especificó en el script de bash la ruta del compilador de java en el sistema

```
→ Ejemplo1 ./script.sh

Lanzando el ligador de RMI...

Compilando con javac ...
./script.sh: línea 10: javac: orden no encontrada
→ Ejemplo1
```

```
echo
echo "Compilando con javac ..."
/usr/lib/jvm/java/bin/javac *.java
```

- **Problemas con el gestor de seguridad:** El servidor producía una excepción al ejecutarse debido a que el parámetro que especificaba la política de seguridad no estaba siendo correctamente introducido. Faltaba un espacio antes del -Djava.security...

```
-Djava.rmi.server.hostname=localhost-Djava.security.policy=server.policy Ejemplo
```

```
-Djava.rmi.server.hostname=localhost -Djava.security.policy=server.policy Ejemplo
```

```
→ Ejemplo1 ./script.sh
```

```
Lanzando el ligador de RMI...
```

```
Compilando con javac ...
```

```
Lanzando el servidor
```

```
Ejemplo excepción
```

```
java.security.AccessControlException: access denied ("java.net.SocketPermission" "127.0.0.1:1099" "connect,resolve")
    at java.base/java.security.AccessControlContext.checkPermission(AccessControlContext.java:472)
    at java.base/java.security.AccessController.checkPermission(AccessController.java:897)
    at java.base/java.lang.SecurityManager.checkPermission(SecurityManager.java:322)
    at java.base/java.lang.SecurityManager.checkConnect(SecurityManager.java:824)
    at java.base/java.net.Socket.connect(Socket.java:604)
    at java.base/java.net.Socket.connect(Socket.java:558)
    at java.base/java.net.Socket.<init>(Socket.java:454)
    at java.base/java.net.Socket.<init>(Socket.java:231)
    at java.rmi/sun.rmi.transport.tcp.TCPDirectSocketFactory.createSocket(TCPDirectSocketFactory.java:40)
    at java.rmi/sun.rmi.transport.tcp.TCPEndpoint.newSocket(TCPEndpoint.java:617)
    at java.rmi/sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:209)
    at java.rmi/sun.rmi.transport.tcp.TCPChannel.newConnection(TCPChannel.java:196)
    at java.rmi/sun.rmi.server.UnicastRef.newCall(UnicastRef.java:343)
    at java.rmi/sun.rmi.registry.RegistryImpl_Stub.rebind(RegistryImpl_Stub.java:150)
    at Ejemplo.main(Ejemplo.java:36)
```

```
AC
```

```
→ Ejemplo1
```

- **El script no avanza de la ejecución del servidor:** Ya que el script proporcionado ejecuta el servidor en primer plano, no es posible pasar a la ejecución de los clientes debido a que el servidor sigue ejecutándose. Para solucionarlo, modificaremos el script para ejecutar el servidor en segundo plano escribiendo "&" al final de la orden que lo ejecuta.

Finalmente, podemos ver el antes (izquierda) y el después (derecha) de la ejecución del ejemplo:

```
-Djava.security.policy=server.policy Ejemplo 8
```

```
→ Ejemplo1 pkill rmiregistry
→ Ejemplo1 ./script.sh

Lanzando el ligador de RMI...

Compilando con javac ...

Lanzando el servidor
Ejemplo bound
```

```
→ Ejemplo1 ./script.sh

Lanzando el ligador de RMI...

Compilando con javac ...

Lanzando el servidor
Ejemplo bound

Lanzando el primer cliente

Buscando el objeto remoto
Invocando al objeto remoto
Recibida una petición del proceso: 0
Empezamos a dormir
Terminamos de dormir

Hebra 0

Lanzando el segundo cliente

Buscando el objeto remoto
Invocando al objeto remoto
Recibida una petición del proceso: 3

Hebra 3
→ Ejemplo1
```

Análisis de su funcionamiento

Este ejemplo presenta un funcionamiento muy básico. Un servidor llamado “Ejemplo” crea un objeto remoto con un solo método en su interfaz, “escribir_mensaje”. Este método recibe un int que representa el id del cliente como parámetro y lo imprime por pantalla, con el añadido de que si se trata del proceso 0 quien accede al stub del objeto remoto, el servidor dormirá durante 5 segundos antes de imprimir el id recibido por pantalla.

1. Archivo de políticas de seguridad:

```
grant codeBase "file:./" {

    permission java.security.AllPermission;

};
```

Con este código, a todos los archivos incluidos en el directorio especificado en “file:” se le concede el permiso llamado java.security.AllPermission que, como su nombre indica, otorga todos los permisos posibles. Este será el archivo de políticas de seguridad utilizado en todos los ejemplos.

-
2. **Definición de la interfaz remota:** En “Ejemplo_I.java” definimos la interfaz remota indicando que hereda de la clase “Remote”. En esta interfaz, definimos el método “escribir_mensaje” e indicamos que lanza una excepción de tipo “RemoteException” para poder ser tratada posteriormente en caso de fallo durante la ejecución del método remoto.
 3. **Implementación de la interfaz remota (Servidor):** La clase “Ejemplo” implementará la interfaz comentada anteriormente y realizará todo el procedimiento necesario para poner a disposición de los clientes el objeto remoto creado.

El método “escribir_mensaje” simplemente imprime el id recibido por pantalla y pone a dormir el servidor previamente si el id recibido es 0.

El main de la clase “Ejemplo” empieza implementando el gestor de seguridad para evitar ejecución de código remoto malintencionado. Se consulta si se ha creado un gestor de seguridad previamente, si no, se crea uno.

```
if (System.getSecurityManager() == null) {  
  
    System.setSecurityManager(new SecurityManager());  
  
}
```

A continuación continuamos con la implementación del objeto remoto. Empezamos instanciando tanto este como su stub mediante

```
Ejemplo_I prueba = new Ejemplo();  
  
Ejemplo_I stub = (Ejemplo_I)  
UnicastRemoteObject.exportObject(prueba, 0);
```

Obtenemos una referencia del registro que utilizaremos para exportar el objeto remoto creado anteriormente, aportando el stub y el nombre por el que lo queremos registrar.

```
Registry registry = LocateRegistry.getRegistry();  
  
registry.rebind(nombre_objeto_remoto, stub);
```

Por último se gestionan las posibles excepciones que puedan suceder durante el proceso.

4. **Implementación del cliente:** El cliente comienza también implementando el gestor de seguridad al igual que el servidor. A continuación, simplemente obtiene una referencia al registro RMI mediante

```
Registry registry = LocateRegistry.getRegistry(args[0]);
```

además de obtener una referencia al objeto remoto para ser usada localmente como una instancia de cualquier clase normal

```
Ejemplo_I instancia_local = (Ejemplo_I)  
registry.lookup(nombre_objeto_remoto);
```

Por último, se llama al método “escribir_mensaje”

```
instancia_local.escribir_mensaje(Integer.parseInt(args[1]));
```

Ejemplo 2

Principales diferencias

Este ejemplo es bastante similar al anterior pero con la principal diferencia de que en lugar de tener varios clientes que entren al stub del método remoto, se tendrá un cliente que ponga en marcha n hebras que serán las que hagan el papel de clientes realizando las llamadas al método remoto.

Otra diferencia con respecto al ejemplo anterior es que el método remoto que utilizaremos toma como parámetro un String en lugar de un int.

Análisis de su funcionamiento

Pasamos a comentar el código utilizado en este ejemplo. Este código es bastante similar al del ejemplo anterior salvo por las siguientes diferencias:

- Respecto al **servidor y la interfaz** que implementa no hay nada que destacar más allá del cambio del tipo de parámetro que recibe el método remoto.

- Es en el main del **cliente** donde apreciamos el principal cambio mencionado anteriormente. Aquí es donde el cliente pone en marcha las hebras que harán de cliente al ejecutar el método run().

```
int n_hebras = Integer.parseInt((args[1]));

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) {

    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

```
+ Ejemplo2 java -cp . -Djava.rmi.server.codebase=file:./ -Djava.rmi.server.hostname=localhost -Djava.security.p
olicy=server.policy Ejemplo &

[2] 37508
+ Ejemplo2 Ejemplo bound

+ Ejemplo2 java -cp . -Djava.security.policy=server.policy Cliente_Ejemplo_Multi_Threaded localhost 3

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

Entra Hebra CLiente 2

Entra Hebra CLiente 0
Empezamos a dormir
Sale Hebra CLiente 2
Invocando el objeto remoto

Entra Hebra CLiente 1
Sale Hebra CLiente 1
Terminamos de dormir
Sale Hebra CLiente 0
+ Ejemplo2 █
```

Podemos ver cómo las distintas hebras van llamando al método remoto y se van entrelazando sus mensajes en el servidor. Observamos también, cómo las hebras cuyo nombre termina en 0 ponen a dormir la ejecución del método remoto en el servidor pero sin impedir que el objeto remoto siga sirviendo al resto de hebras. Esto es debido a que RMI es multihebrado, siendo capaz de gestionar concurrentemente las peticiones de los clientes.

A continuación, veremos cómo influye el uso del modificador `synchronized` en el orden en el que aparecen los mensajes.

Uso del modificador “synchronized”

Al añadir “synchronized” al método remoto observamos que el método pasa a ser de exclusión mutua, permitiendo la entrada de una sola hebra/cliente a la vez. No es hasta que sale la hebra que ha entrado que puede proceder la siguiente.

```
public synchronized void escribir_mensaje(String mensaje) {
```

```
→ Ejemplo2 rmiregistry &
[2] 6475
→ Ejemplo2 /usr/lib/jvm/java/bin/javac *.java
→ Ejemplo2 java -cp . -Djava.rmi.server.codebase=file:./ -Djava.rmi.server.hostname=localhost -Djava.security.policy=server.policy Ejemplo &

[3] 6541
→ Ejemplo2 Ejemplo bound

→ Ejemplo2 java -cp . -Djava.security.policy=server.policy Cliente_Ejemplo_Multi_Thread localhost 5

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

Entra Hebra Cliente 2
Sale Hebra Cliente 2

Entra Hebra Cliente 1
Sale Hebra Cliente 1

Entra Hebra Cliente 3
Sale Hebra Cliente 3

Entra Hebra Cliente 0
Empezamos a dormir
Invocando el objeto remoto
Terminamos de dormir
Sale Hebra Cliente 0

Entra Hebra Cliente 4
Sale Hebra Cliente 4
→ Ejemplo2 █
```

Ejemplo 3

Principales diferencias

La principal diferencia de este ejemplo en comparación con los dos anteriores es la forma en la que está organizada el código. En este ejemplo, la clase del que será el objeto remoto (Contador.java) y la del servidor (Servidor.java) están separadas. El servidor creará una instancia de Contador y exportará los métodos presentes en la interfaz (icontador.java) para que los clientes puedan interactuar con el objeto remoto de la clase Contador.

Además, también es necesario destacar las diferencias existentes respecto a los métodos relacionados con RMI utilizados. Analizaremos estas diferencias a continuación.

Análisis de su funcionamiento

El funcionamiento de este ejemplo consiste en un servidor que contiene una instancia de una clase Contador el cual será inicializado a 0 por un cliente para posteriormente realizar 1000 incrementos de 1, imprimiendo el valor final del contador junto con el tiempo medio de respuesta de las invocaciones al método remoto de incrementar.

El funcionamiento del código es bastante simple y similar a los ejercicios anteriores pero es necesario comentar las diferencias en el proceso de interacción con el registro RMI

- En el **servidor**:
 - El proceso de inicialización del registro RMI se hace en el código en lugar de hacerlo por terminal mediante “rmiregistry &”

```
Registry reg = LocateRegistry.createRegistry(1099);
```

- Para registrar el objeto remoto, en lugar de general el stub, obtener la referencia al registro RMI y registrar el objeto, se acorta este proceso con el método

```
Naming.rebind("micontador", micontador);
```

- En el **cliente**:
 - Se obtiene la referencia al registro especificando también el puerto además del servidor aunque esto es completamente opcional.

```
Registry mireg = LocateRegistry.getRegistry("127.0.0.1",  
1099);
```

Ejecución

```
+ Ejemplo3 java -cp . -Djava.rmi.server.codebase=file:./ -Djava.rmi.server.hostname=localhost -Djava.security.p
olicy=server.policy Servidor &

[1] 11057
+ Ejemplo3 java -cp . -Djava.security.policy=server.policy Cliente
Poniendo contador a 0
Incrementando...
Media de las RMI realizadas = 0.095 msecs
RMI realizadas: 1000
+ Ejemplo3 java -cp . -Djava.security.policy=server.policy Cliente
Poniendo contador a 0
Incrementando...
Media de las RMI realizadas = 0.06 msecs
RMI realizadas: 1000
+ Ejemplo3 java -cp . -Djava.security.policy=server.policy Cliente
Poniendo contador a 0
Incrementando...
Media de las RMI realizadas = 0.06 msecs
RMI realizadas: 1000
+ Ejemplo3 █
```

Observamos que el ejemplo se ejecuta correctamente y se devuelve el número de incrementos realizado junto con el tiempo media de respuesta de las invocaciones remotas a incrementar.

PARTE 2: Servidor replicado

Introducción

En esta parte, se ha implementado el servidor de donaciones replicado que se pedía y se le han realizado alguna de las extensiones de funcionalidad extra como pueden ser la posibilidad de usar más de 2 réplicas, la posibilidad de utilizar ordenadores distintos para la comunicación cliente-servidor, la gestión de excepciones más extensiva o la implementación de uno de los algoritmos vistos en clase para exclusión mutua, el algoritmo de EM basado en anillo.

Comunicación entre réplicas

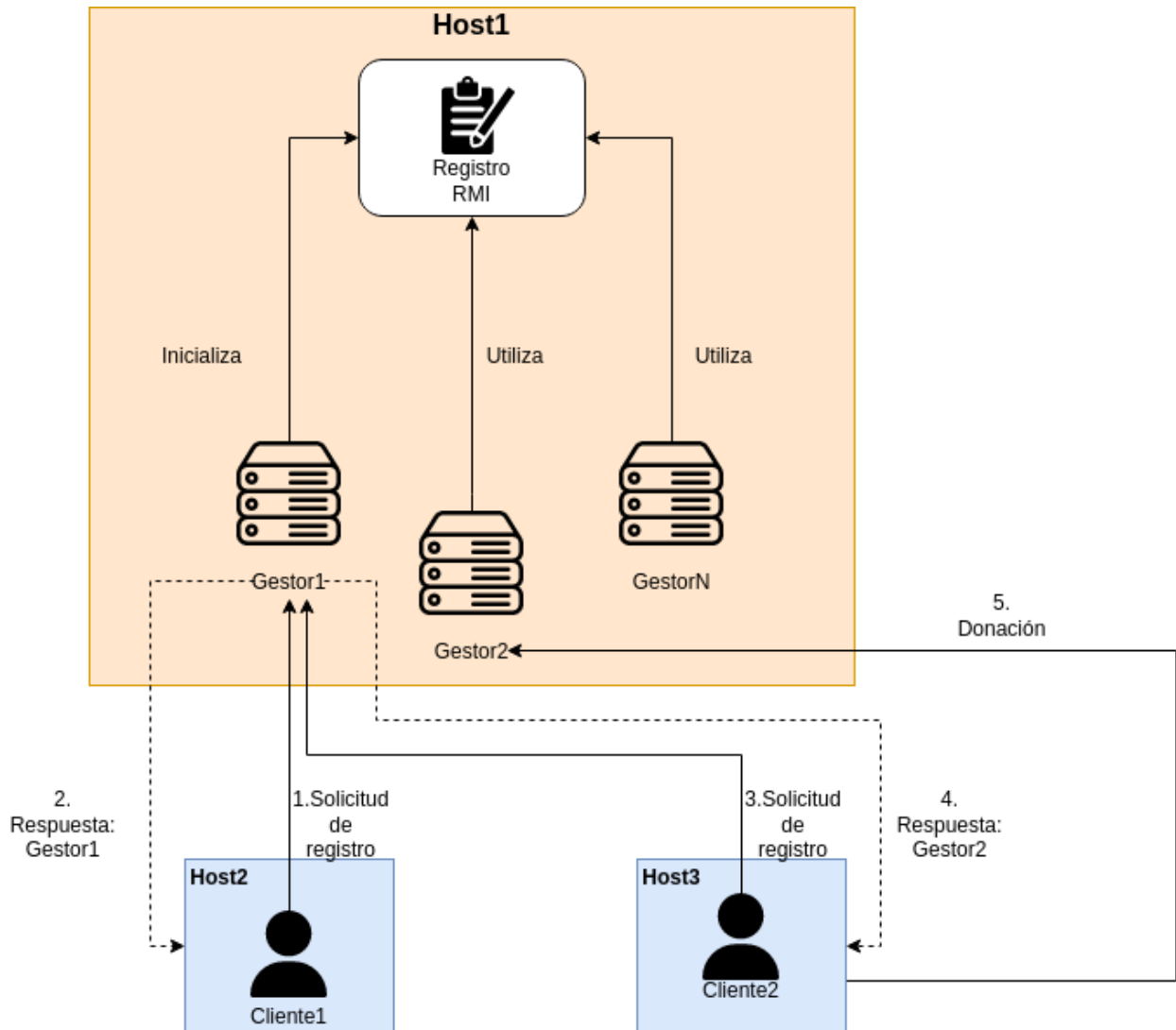
Como hemos mencionado anteriormente, en esta aplicación se podrán utilizar tantas réplicas como se desee. La primera réplica es la que inicializa el registro RMI (en el caso de que no se haya especificado ningún host para el registro), mientras que las n siguientes detectarán si ya existe un registro en funcionamiento y lo utilizarán para registrarse y darse a conocer al resto de réplicas. Para ello, cada réplica almacenará un array con referencias al resto de réplicas el cual actualizará mediante el uso del método `Naming.list()`. Este método devuelve el listado de nombres de objetos remotos (en este caso, réplicas del servidor) registrados en el registro RMI hasta el momento.

Nos interesa que todas las réplicas/gestores de donaciones se conozcan entre sí principalmente para solucionar dos problemas:

- **La redirección de clientes a su réplica correspondiente:** Debido a que deseamos que un cliente pueda registrarse de forma transparente en la réplica que menos clientes tenga registrados independientemente de a la que haya mandado la solicitud de registro, querremos que la réplica que está manejando la solicitud de registro pueda preguntarle al resto su número de clientes para así decidir a cual redirigir el registro del usuario. Una vez el cliente esté registrado en la réplica correspondiente, se le responderá con la referencia a la réplica con la que se tendrá que comunicar a partir de ahora.
- **Consistencia de la variable replicada del total de donaciones:** Las réplicas tendrán una variable para almacenar el total donado al sistema. Como es lógico, se tendrán que implantar medidas para mantener la consistencia de esta variable a lo largo de todas las réplicas utilizadas. Para ello, cuando un cliente realiza una donación a una réplica, esta realiza un mensaje de tipo broadcast al resto comunicándoles la donación recibida para que puedan mantener actualizado el total de donaciones. Además, cuando una nueva réplica se registra, necesitará consultar si hay otras réplicas en funcionamiento para poder solicitarles el total donado actual y partir con un valor actualizado. Por último, para

reforzar la consistencia de esta variable replicada, se ha hecho uso del algoritmo de exclusión mutua basado en anillo del cual hablaremos más adelante.

Podemos ver representado en este diagrama un posible caso de uso de la aplicación con tres gestores y dos clientes corriendo en distintos ordenadores:



Explicación de métodos utilizados

En este apartado comentaremos el funcionamiento y la utilidad de los métodos utilizados más relevantes. Los métodos marcados con * serán comentados en el apartado de “Exclusión mutua basado en anillo”.

Servidor

- **GestorDonacionesDriver**

- **main():** Es aquí donde se crean y registran las réplicas que gestionarán las donaciones realizadas por los clientes. Antes de comentar su funcionamiento es necesario destacar las dos formas de poner en marcha una réplica en base a los parámetros con los que se ejecute el main. En primer lugar, si el gestor se pone en marcha especificando únicamente el id del gestor, se asumirá que el host para el registro será localhost y se intentará crear un registro RMI en el puerto 9991. En caso de estar ya creado, se registrará al gestor en este.

En segundo lugar, si se pasan como parámetros idGestor y host, se intentará registrar al gestor en el host proporcionado sin crear un nuevo registro.

Una vez obtenido el registro a utilizar, se consultará si hay gestores ya registrados en este por dos motivos:

- En caso de no ser el primer gestor en registrarse, será necesario preguntarle a otro gestor por el valor de la variable del total de donaciones para partir con un valor actualizado.
- En caso de ser el primer gestor en registrarse, este tendrá que portar el token que se utilizará para la EM basada en anillo.

Tras realizar estas comprobaciones se registra el gestor y se pone en marcha el circulamiento del token para el algoritmo de EM.

- **addControladorCierre():** Este método se encarga de añadir un manejador de señales para tratar un posible Ctrl + c en caso de querer cerrar el servidor. Esto se hace para poder eliminar el gestor del registro RMI antes de salir y así evitar problemas en la comunicación con el resto de gestores.

- **GestorDonaciones**

- **registrarCliente():** Es aquí donde se halla la lógica de la redirección del usuario a su servidor correspondiente. La idea principal es devolver al usuario un Pair con la referencia al gestor al que tendrá que comunicarse de aquí en adelante en conjunto con un entero que indicará el número de clientes que posee dicho gestor. Este entero se establecerá a -1 si el cliente ya estaba registrado anteriormente en alguna de las réplicas, es decir, si el cliente ha iniciado sesión en lugar de registrarse.

El funcionamiento es el siguiente:

1. Se comprueba si el usuario está registrado en la réplica con la que se ha comunicado, en caso de estarlo se le devuelve una referencia al gestor actual y un -1.
 2. Si no está registrado en dicha réplica se va consultando réplica a réplica si dicho usuario está registrado en ella. En caso de no estarlo se le pide el número de clientes que tiene. Esta referencia pasará a ser el gestor candidato para el registro si su número de usuarios es menor que el mínimo actual.
 3. En caso de descubrir que una de las réplicas tiene ya registrado al usuario se detiene la búsqueda y se le devuelve al usuario la referencia a dicha réplica.
- **donar():** Este método presenta un funcionamiento muy simple. En caso de que un cliente solicite realizar una donación mientras que el gestor tiene el token de EM, se accede a sección crítica y se actualiza tanto el subtotal del gestor como su total de donaciones para posteriormente comunicar esa donación al resto de réplicas y salir de SC. En caso de no poseer el token en el momento de la donación, esta queda encolada para ser realizada cuando el gestor reciba el token.
 - **gestionarToken()***
 - **getSiguienteReplica()***
 - **actualizarListadoReplicas():** Es el método encargado de mantener actualizado el listado de réplicas que permite a un gestor conocer al resto. Como hemos comentado anteriormente, esto lo hace utilizando el método Naming.list()
 - **despacharDonacionesPendientes():** Este será el método al que se llame cuando el gestor reciba el token y tenga donaciones encoladas, es decir, cuando esté a la espera de acceder a SC.

Cliente

- **ClienteDonacionesDriver**
 - **main():** Este main toma dos parámetros, la id del gestor al que solicitar el registro y el host en el que se halla el registro RMI. Aquí simplemente se instala el gestor de seguridad y se crea una hebra con un cliente al que se le pasará el gestor con la id especificada como parámetro.
- **ClienteDonaciones**

-
- **registrarme():** En este método se realiza la primera comunicación con el gestor especificado como parámetro del main de ClienteDonacionesDriver. Se le pedirá un nombre de usuario al cliente para posteriormente intentar el registro en el gestor. De la respuesta de la petición de registro el cliente actualizará su referencia al gestor con el que comunicarse y en caso de que el segundo valor de la pareja obtenida en la respuesta sea -1, sabrá que ya estaba registrado previamente.
 - **procesarInput():** Este método simplemente gestiona el input del usuario y la respuesta por parte del gestor junto con algunas excepciones.

Exclusión mutua basado en anillo

Este algoritmo consigue exclusión mutua mediante el continuo circulamiento de un token entre las réplicas. El poseedor del token será el único capaz de entrar a sección crítica (en este caso, modificar el valor de la variable replicada del total de donaciones). El algoritmo funciona correctamente aunque no se han implementado medidas de regeneración del token en caso de que uno de los gestores muera siendo portador del token, aunque el programa si es capaz de recuperarse si el token no se pierde gracias a que todas las réplicas se conocen entre ellas, siendo tan fácil como que los gestores pasen a tener un nuevo siguiente gestor al que pasarle el token.

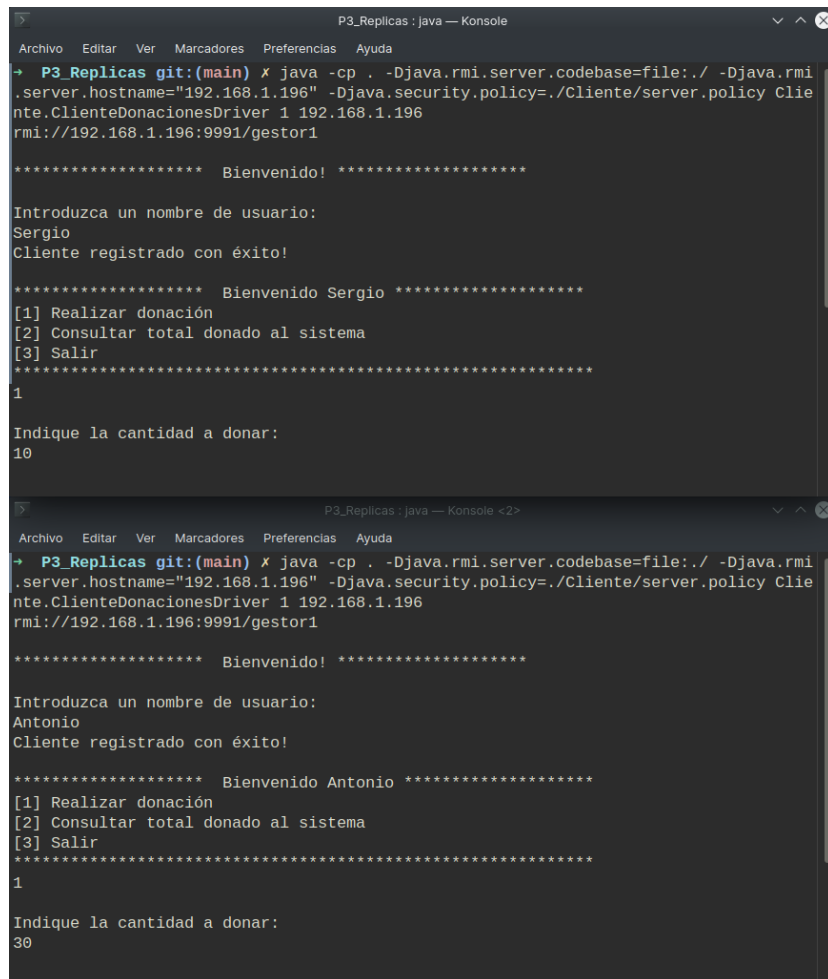
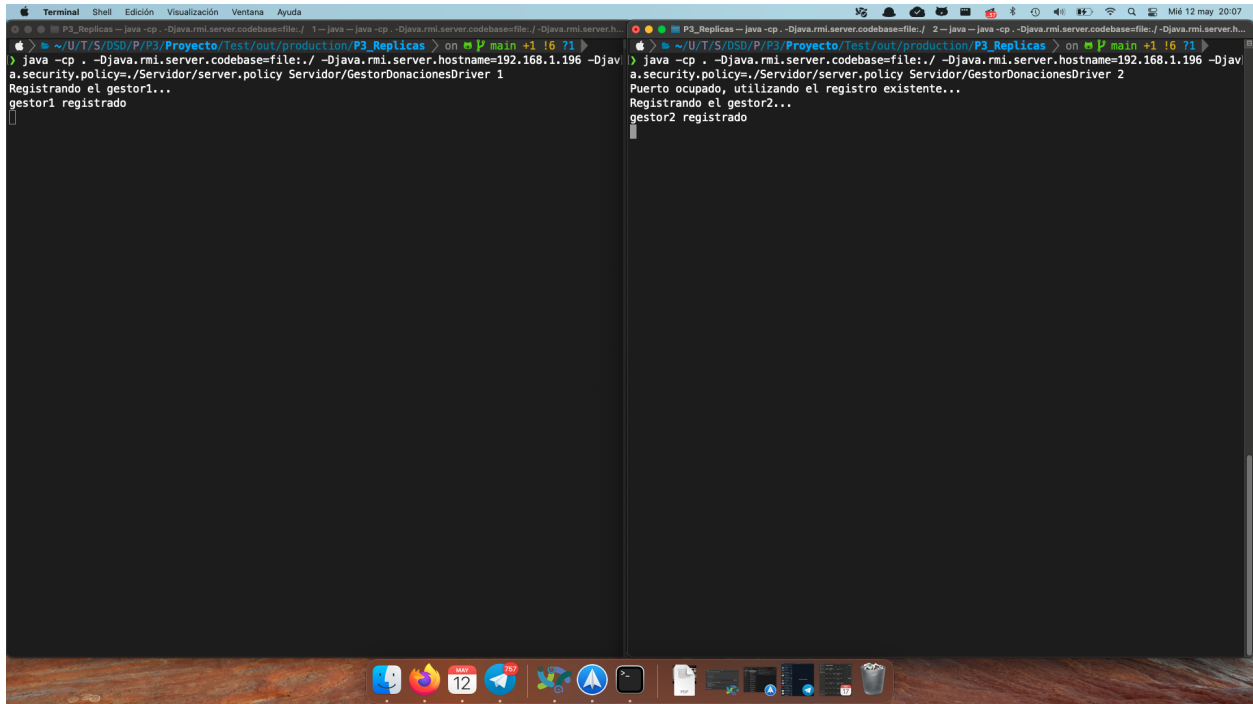
Esta elección de la siguiente réplica a la que pasarle el token siguiendo una disposición de anillo se realiza en el método de **getSiguienteReplica()**. En este, se busca de entre todas las réplicas la que posea el id inmediato siguiente al id del gestor actual. Teniendo tres gestores, gestor1, gestor 2 y gestor3, la circulación del token sería la siguiente:

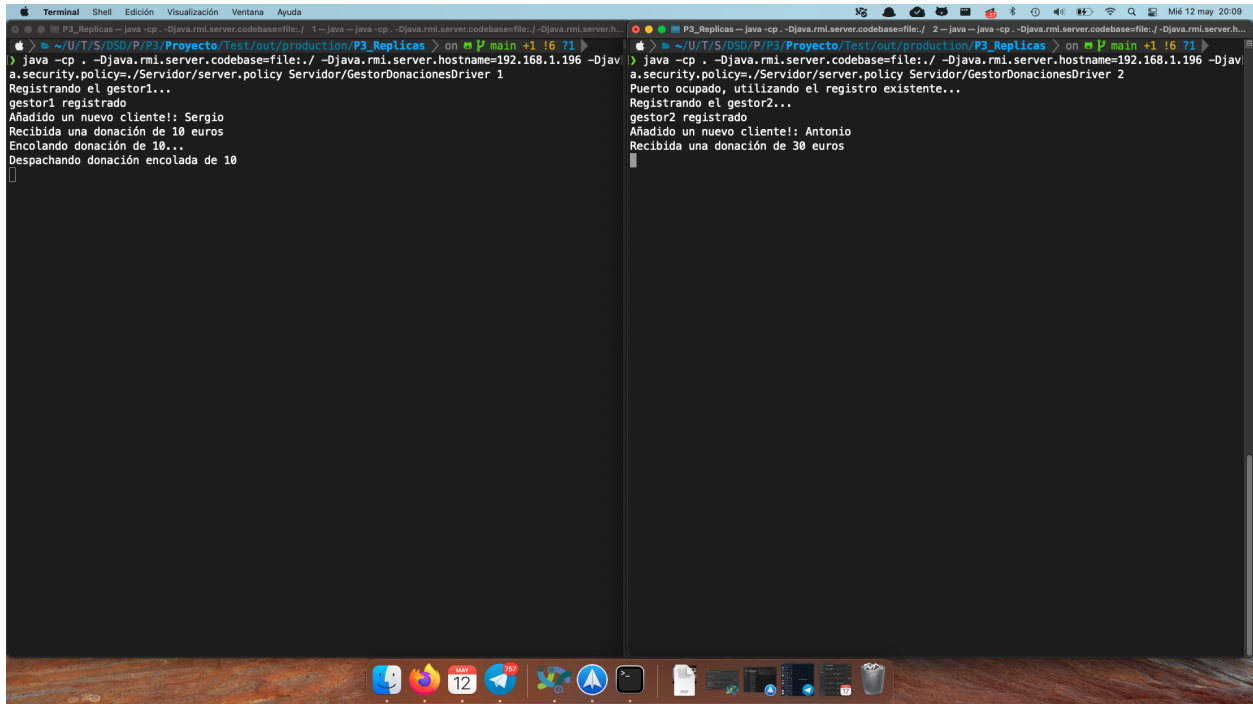
gestor1 -> gestor2 -> gestor3 -> gestor1.

Podemos ver un vídeo del algoritmo en funcionamiento en este [enlace](#).

Funcionamiento en distintos ordenadores y sus limitaciones

La aplicación admite el uso de distintos ordenadores para la comunicación cliente servidor, es decir, se podría tener un servidor con n réplicas y n distintos ordenadores haciendo de clientes. Para conseguir esto, se ha parametrizado el host a utilizar para el registro RMI. Para poner a prueba esta funcionalidad se han puesto en marcha dos réplicas corriendo en un portátil Mac y dos clientes que se ejecutarán en un ordenador de sobremesa corriendo Fedora Linux.





```
Terminal Shell Edición Visualización Ventana Ayuda
~/U/T/S/OSD/P/P3/Proyecto/Test/out/production/P3_Replicas > on P main +1 16 71
> java -cp . -Djava.rmi.server.codebase=file:/ -Djava.rmi.server.hostname=192.168.1.196 -Djava.security.policy=/Servidor/server.policy Servidor/GestorDonacionesDriver 1
Registrando el gestor1...
gestor1 registrado
Añadido un nuevo cliente!: Sergio
Recibida una donación de 10 euros
Encolando donación de 10...
Despachando donación encolada de 10

~/U/T/S/OSD/P/P3/Proyecto/Test/out/production/P3_Replicas > on P main +1 16 71
> java -cp . -Djava.rmi.server.codebase=file:/ -Djava.rmi.server.hostname=192.168.1.196 -Djava.security.policy=/Servidor/server.policy Servidor/GestorDonacionesDriver 2
Puerto ocupado, utilizando el registro existente...
Registrando el gestor2...
gestor2 registrado
Añadido un nuevo cliente!: Antonio
Recibida una donación de 30 euros
```

En estas capturas podemos observar cómo especificando el host que contiene el registro RMI es posible separar los clientes de las réplicas en distintos ordenadores, además de ver también cómo aún habiendo ambos clientes contactado con el mismo gestor, la redirección se realiza correctamente y de forma transparente.

También cabe destacar que la arquitectura que implementa la aplicación presenta limitaciones, ya que la aplicación está diseñada para hacer uso de un único registro RMI para todas las réplicas. Esto trae como consecuencia el no poder tener réplicas en distintos ordenadores ya que Java RMI prohíbe el binding en registros remotos.

Gestión de excepciones

Ya que estamos tratando con invocaciones de métodos remotos, es necesario tener en cuenta el amplio catálogo de excepciones que pueden surgir durante la ejecución de la aplicación. Estas excepciones pueden ir desde una básica `RemoteException` (excepción que han de tener obligatoriamente los métodos en la interfaz de los objetos remotos en RMI) hasta otras como `ServerException`, `MalformedURLException`, `ConnectException` y similares. El objetivo era tener controlados los distintos casos en los que se podía dar cada excepción y manejarla acordeamente, en caso de ser una excepción asumible se avisa por consola y se prosigue con la ejecución de la aplicación. En caso de ser excepciones irremediables, se avisa por consola y se aborta la ejecución con `System.exit(-1)` para devolver un código de error en la salida. Algunos ejemplos de esta gestión de excepciones son:

```
try {
    registry.bind(nombre, gestor);
    System.out.println(nombre + " registrado");
} catch (AlreadyBoundException e) {
    System.out.println("Id de gestor ya en uso, inténtelo con otro identificador");
    System.exit( status: -1);
} catch (ServerException e) {
    System.out.println("Por motivos de seguridad, no es posible registrar un gestor en un servidor remoto, " +
        "mantenga los gestores en un mismo host");
    System.exit( status: -1);
}
```

```
System.out.println("\nIndique la cantidad a donar: ");
if (input.hasNextInt()) {
    cantidad = input.nextInt();
    try {
        gestor.donar(cantidad, username);
        System.out.println("\nDonación de " + cantidad + " realizada con éxito");
    } catch (RemoteException | InterruptedException e) {
        System.out.println("Error al realizar la donación, inténtelo de nuevo");
    }
}
```

Instrucciones de uso

La aplicación será distribuida como un proyecto de IntelliJ IDEA. Para su utilización será necesario descomprimir el .zip del proyecto, abrir como proyecto en IntelliJ la carpeta “P3_Replicas” y a continuación, deberemos de pulsar el martillo para compilar el proyecto:



En ocasiones, IntelliJ da problemas y muestra el proyecto lleno de errores en los imports, para solucionarlo es necesario acceder a File > Invalidate Caches, marcar también la primera opción y pulsar en “Invalidate and Restart”.

Si todo ha ido bien, debería de haber aparecido una carpeta out en P3_Replicas/out con los resultados del proceso de compilación. Abrimos una terminal y nos colocamos en **P3_Replicas/out/production/P3_Replicas/**. Una vez aquí empezamos ejecutando al menos un **gestor** con la siguiente orden

```
java -cp . -Djava.rmi.server.codebase=file:./ -Djava.rmi.server.hostname=localhost
-Djava.security.policy=./Servidor/server.policy Servidor/GestorDonacionesDriver 1
```

Los parámetros que recibe el gestor son:

-
- En primer lugar y de forma obligatoria un identificador (el cual deberá establecerse en incrementos de 1 para el resto de réplicas futuras).
 - En segundo lugar y optativamente un host al que solicitarle el registro RMI.

Una vez tengamos funcionando el gestor podremos poner en marcha más gestores simplemente incrementando el id en 1.

Para los **clientes** utilizaremos la siguiente orden en otras ventanas de terminal:

```
java -cp . -Djava.rmi.server.codebase=file:./ -Djava.rmi.server.hostname="localhost"  
-Djava.security.policy=./Cliente/server.policy Cliente.ClienteDonacionesDriver 1 localhost
```

Sustituyendo localhost por la dirección del host en el que se encuentren los gestores. En este caso ambos parámetros son necesarios.

Se podrá indicar cualquier identificador de gestor válido que se desee para poner en marcha un cliente aunque posteriormente dicho gestor redirija al cliente a su gestor correspondiente.

Ejemplo de uso

Podemos ver un ejemplo de uso de la aplicación en el siguiente [vídeo](#). En este ejemplo usamos tres clientes y dos gestores todos corriendo en la misma máquina. Realizamos algunas comprobaciones como que un cliente no se pueda registrar varias veces, que no pueda consultar el total sin haber donado antes, que la redirección de clientes funcione correctamente o que se mantenga actualizado el total de donaciones en las distintas réplicas.