

# Práctica 3

## Parte 1

En esta primera parte hemos estudiado los casos que se nos han dejado en prado, estos dos son:

- RmiHello
- RmiChatApplication

Hemos analizado el código de estos dos ficheros y vamos a continuación a proceder a explicarlo.

### RmiHello

Este es un programita realmente sencillo que se compone de 3 ficheros alojados todos en un paquete común. El primer fichero que nos interesa explicar es "HelloService.java", en este fichero definimos la interfaz donde tenemos los métodos que se implementarán en otras clases posteriormente (recordemos que el uso de interfaces nos permite poner de acuerdo a los clientes con los servidores al menos en el formato de los datos). El siguiente fichero interesante es "HelloServant.java", en él se encuentra la clase que implementa los métodos de la interfaz. En "client.java" se recupera el objeto llamado "hello" haciendo una llamada a Naming.lookup(...). A diferencia de los ejemplos mostrados en las transparencias aquí se simplifica bastante todo el código de las funciones main debido a que las clases que implementan las interfaces heredan de `UnicastRemoteObject`. El fichero "server.java" crea una instancia de la interfaz y le asigna un nombre a dicho objeto, registrándolo en el registry.

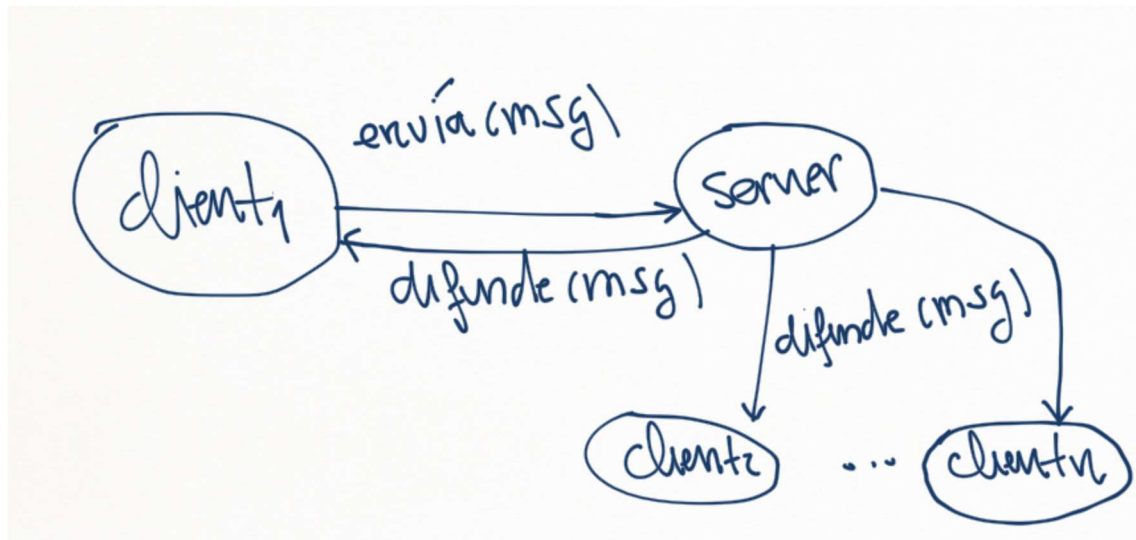
El resultado final es un objeto que existe en el servidor y registrado en el rmi registry para poder ser usado por clientes. Los clientes buscan el objeto en cuestión mediante dirección ip y puerto del servidor además del nombre del objeto, el servicio lo devuelve y a partir de ese momento se puede operar con dicho objeto como si fuese otro objeto local más.

Lo que hace el programa RmiHello es buscar el objeto hello e invocar su método echo, que se ejecuta en el servidor. Al final obtenemos un valor de retorno que consiste en el String que se le pase a "echo" precedido por "From server: " (cadena que añade el servidor al recibir la petición).

### RmiChat

Su estructura es parecida a la del ejemplo anterior, pero se organiza de manera diferente. En este caso tenemos dos paquetes (o carpetas) uno dedicado al cliente y otro al servidor. Tenemos que el cliente es concurrente y se pueden tener varias hebras. Cada cliente manda al servidor un mensaje y el servidor se encarga de difundirlo a todos los clientes que se hubiesen registrado previamente. Tenemos dos interfaces pues, una para los clientes y otra para el servidor. En la función main de los clientes se obtiene el objeto remoto correspondiente al servidor y se lanza una hebra con un nuevo chatClient que tiene el nombre del cliente y el objeto remoto servidor. En el chatClient se invoca al método correspondiente para registrar al nuevo cliente en el servidor. En la hebra mencionada previamente se lee la entrada y se va llamando al método `broadcastMessage` pasándole lo que se leyó. El servidor recibe el mensaje y lo difunde a todos los clientes que se registraron previamente incluido el que envió el mensaje para ello se recorren todos los clientes registrados y se llama al método `retrieveMessage`. Los objetos

que se encuentran registrados son referencias al objeto remoto que se registró previamente. A continuación, se presenta un dibujillo con lo que sucede cuando un cliente envía un mensaje.



Sobre la versión del RmiChatApplication que hemos estudiado, hemos realizado algunos cambios para que sea más interactiva e interesante. Hemos inhabilitado que el servidor envíe al cliente un mensaje que él mismo envió y hemos añadido sentencias en las que podemos ver más claramente lo que está sucediendo.

Dejamos a continuación una captura de un caso práctico ejecutado.

```
ejercicio 5 tmp - RmiChatMio/src/es/ugr/dsd2/rmichat/client/ChatClientDriver.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Console
ChatServerDriver [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (23 abr. 2021 14:55:32)
Servidor arrancado ...
[Servidor difunde] Paquito : Buenos dias grupooo
[Servidor difunde] Manolo : Cómo estamos manolo? todo bien?
[Servidor difunde] Paquito : Por aqui todo bien y los demás que?
[Servidor difunde] Manolo : pues creo que somos los únicos aquí en el grupo eh
[Servidor difunde] Paquito : Se fueron todos?
[Servidor difunde] Manolo : Eso parece si jeje

Console
ChatClientDriver [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (23 abr. 2021 14:55:38)
----- Paquito -----
Buenos dias grupooo

[Recibido mensaje] Manolo : Cómo estamos manolo? todo bien?
Por aqui todo bien y los demás que?

[Recibido mensaje] Manolo : pues creo que somos los únicos aquí en el grupo eh
Se fueron todos?

[Recibido mensaje] Manolo : Eso parece si jeje

Console
ChatClientDriver [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (23 abr. 2021 14:55:41)
----- Manolo -----

[Recibido mensaje] Paquito : Buenos dias grupooo
Cómo estamos manolo? todo bien?

[Recibido mensaje] Paquito : Por aqui todo bien y los demás que?
pues creo que somos los únicos aquí en el grupo eh

[Recibido mensaje] Paquito : Se fueron todos?
Eso parece si jeje
```

## Parte 2

En esta parte de la práctica resolvemos el ejercicio del servidor de donaciones replicado de varias maneras. Usamos el IDE Eclipse-Java, basta con importar el proyecto exportado o crear un nuevo proyecto con los ficheros en “src/” y pulsar el botón de compilar para que se ejecute el programa. A continuación, se describen las diferentes versiones:

### VERSIÓN BÁSICA

En esta versión hemos resuelto el problema del servidor replicado tal y como se plantea, es decir, tenemos un servidor que se encuentra replicado (solo 2 réplicas) que tiene en cada replica un subtotal de donaciones y que atiende peticiones de clientes. Se distribuyen los clientes entre las réplicas en base a la carga de estos. Los clientes se comunican con las réplicas mediante una interfaz (**ServidorConClienteI**) y las réplicas entre ellas mediante otra (**ServidorConServidorI**), además se define una interfaz para que las réplicas se comuniquen con los clientes (**ClienteI**). Los clientes llevan a cabo dos operaciones básicas, donar y obtener el total de donaciones. Las réplicas son las encargadas de registrar nuevos clientes y comunicarse entre ellas.

*La clase que implementa los métodos de las interfaces del servidor descritas previamente es **ServidorReplicado**, la clase que implementa los métodos de la interfaz del cliente **Cliente**. La ejecución de los programas se hace en dos clases que implementan el método **main**, para las réplicas **ServidorReplicadoDriver**, para los clientes **ClienteDriver**.*

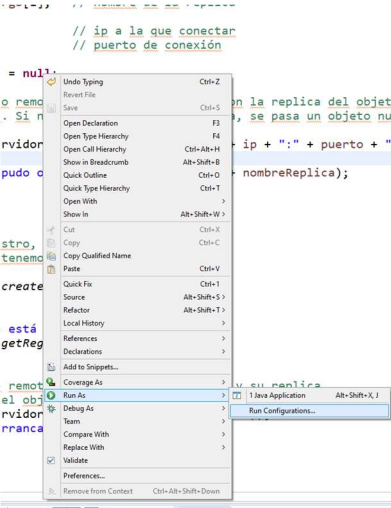
Para ejecutar una réplica ejecutamos ServidorDriver pasándole dos argumentos, el primero es el nombre del servidor a ejecutar y el segundo es el de su réplica. Cuando se inicia la ejecución, se intenta crear un nuevo registro, si este existía previamente entonces no se crea y se obtiene (Locate.getRegistry(<puerto>)). A continuación, se intenta obtener el objeto remoto correspondiente a la réplica, si falla este paso se atrapa la excepción, se informa por pantalla y se sigue la ejecución del programa (es normal que ocurra esto, no se pueden lanzar dos replicas exactamente al mismo tiempo, una irá antes que otra). Al lanzar otra réplica, esta vez le pasamos como argumentos, su nombre de servidor y el nombre de la réplica ya lanzada, en este caso ejecutará exactamente los mismos pasos y no fallará (no por la excepción anterior). Cuando se obtiene el objeto remoto de la réplica previamente lanzada, se construye la nueva réplica pasándole este como argumento y se registra. En el constructor se invoca el método setReplica(<ServidorConServidorI>) de la réplica previa para que obtenga la referencia a la nueva réplica.

REPLICA1	REPLICA2
Iniciamos primera réplica e indicamos nombre de la segunda	
Crea registro y registra nueva réplica (sin conocer su réplica)	
	Iniciamos segunda réplica e indicamos nombre de la primera
	Intenta crear el registro, si fue creado antes, lo obtiene
	Obtiene primera réplica
	Informa a primera réplica, que ella (la segunda) es su réplica
Obtiene mensaje de su réplica y la guarda	

Para ejecutar un cliente debemos ejecutar la clase **ClienteDriver**, podemos pasarle como argumento un nombre, para identificar mejor al cliente. El cliente por defecto intenta conectarse a una réplica, nuestra réplica por defecto es la réplica1. Cuando se obtiene dicha réplica, se invoca el método **registrarseEnServidor(<Clientel>)**, en ese método se busca la réplica con menor número de clientes (carga) registrados y se le asigna al cliente mediante el método **setReplica(<ServidorConCliente>)**.

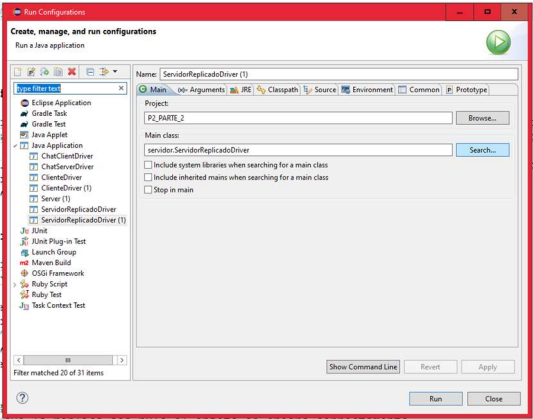
CLIENTE	RÉPLICA
Obtiene objeto remoto réplica	
Solicita registro en réplica	
	Recibe intento registro y busca réplica con menor número de carga
	Asigna replica elegida a cliente
Guarda la nueva réplica	

Casos prácticos de ejecución



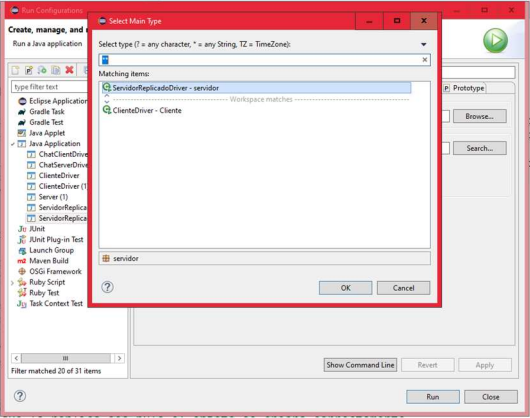
Hacemos clic derecho sobre el fichero “ServidorReplicadoDriver” y escogemos la opción “run as”.

Nos aparecerá la siguiente ventana. Pulsamos el botón search (o buscar en español) para buscar el fichero que queremos ejecutar.

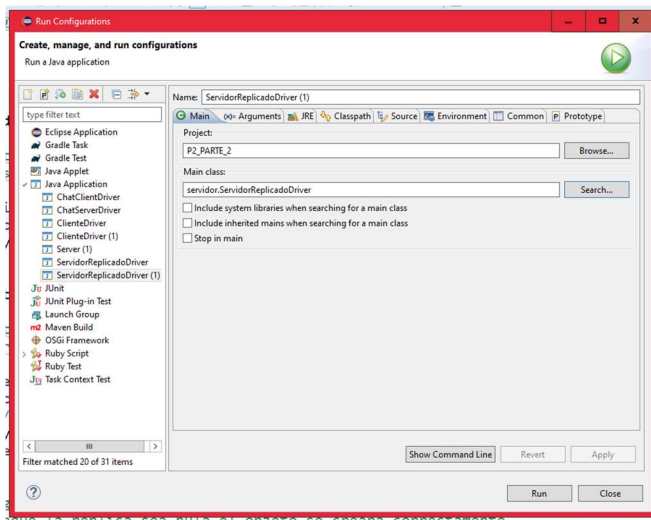


Al pulsar dicho botó nos aparece la siguiente

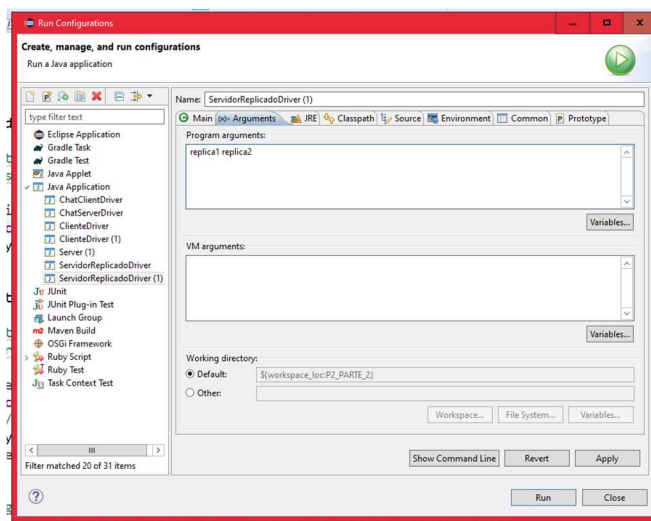
ventana, en ella aparecen los ficheros que podemos compilar, seleccionaremos



ServidorReplicadoDriver en esta ocasión. Habría que hacer exactamente la misma secuencia de pasos para ejecutar el cliente, escogiendo ClienteDriver.

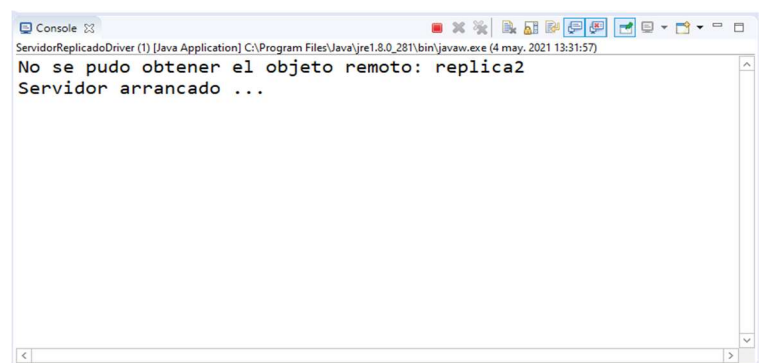


Una vez elegida la clase a compilar como main nos vamos a ir a la pestaña arguments, en ella podemos especificar qué argumentos le pasaremos al programa servidor. La pestaña arguments está en el panel derecho en la barra de arriba justo al lado de main.



En el recuadro Program Arguments, vamos a escribir dos palabras, una es “replica1” y otra “replica2” son los correspondientes nombres del servidor a ejecutar y de su réplica. Pulsamos a continuación el botón inferior izquierdo “run” para compilar y ejecutar definitivamente.

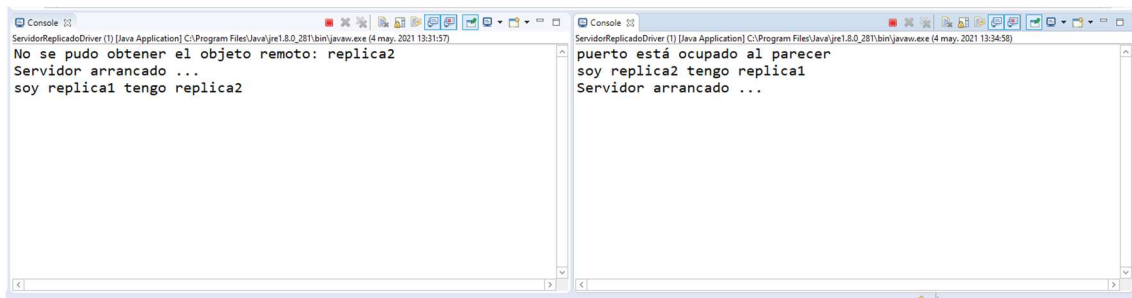
El resultado de la ejecución es el siguiente: al ser la primera instancia del servidor que se lanza, no se encuentra a su réplica y nos lo notifica, la ejecución sigue sin errores y con un servidor arrancado y a la espera de peticiones.



Repetimos todo el proceso de nuevo y ahora en el recuadro program arguments escribimos “replica2” “replica1”, para lanzar la segunda replica e indicarle que su réplica es la primera.



Tenemos ya las dos réplicas funcionales y conectadas entre sí.

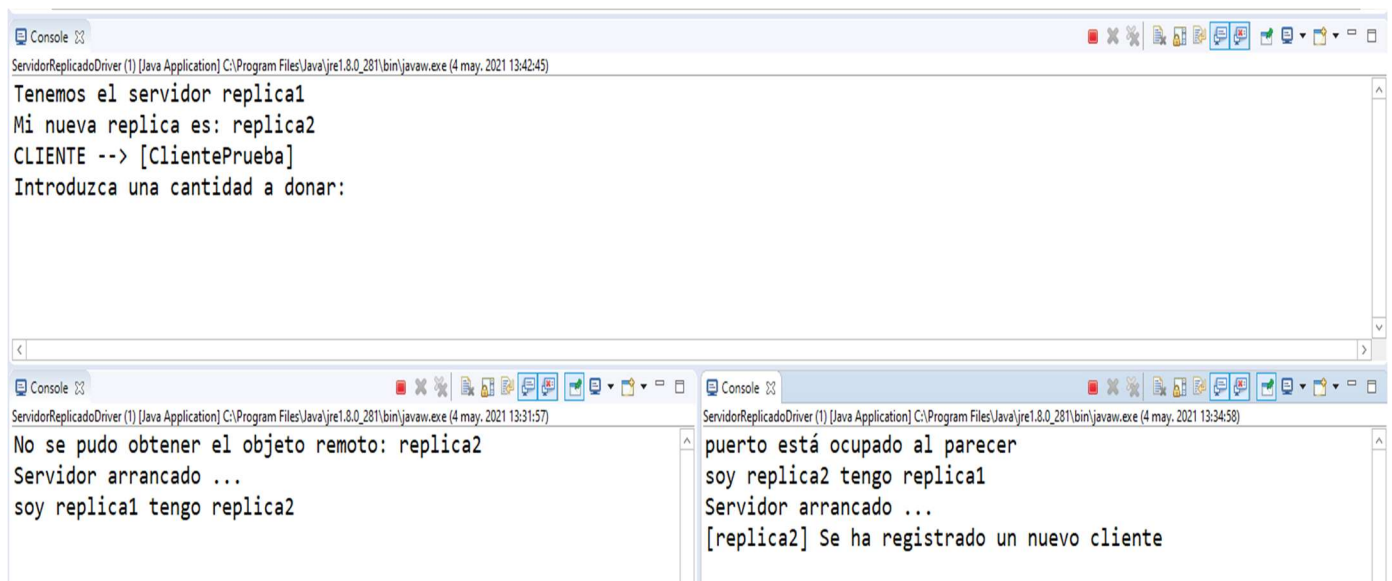


```
Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:31:57)
No se pudo obtener el objeto remoto: replica2
Servidor arrancado ...
soy replica1 tengo replica2

Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:34:58)
puerto está ocupado al parecer
soy replica2 tengo replica1
Servidor arrancado ...
```

Vemos que en la segunda réplica se ha intentado crear un puerto, pero al estar ocupado se entendió que ya fue creado, por lo que lo obtiene para registrarse en él como nuevo objeto remoto. Se le asigna la replica1 a la 2 y a la 1 se le notifica y se le asigna la 2. La primera réplica recibe dicho mensaje correctamente y lo muestra por pantalla.

Vamos a crear un cliente ahora que realice peticiones. Para ello vamos a seguir de nuevo todo el proceso anterior, en este caso al pulsar el botón search escogeremos el fichero ClientDriver y en argumentos pasaremos solo el nombre del nuevo cliente o ninguno. Al ejecutar el cliente con nombre “ClientePrueba” obtenemos el siguiente resultado:



```
Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:42:45)
Tenemos el servidor replica1
Mi nueva replica es: replica2
CLIENTE --> [ClientePrueba]
Introduzca una cantidad a donar:

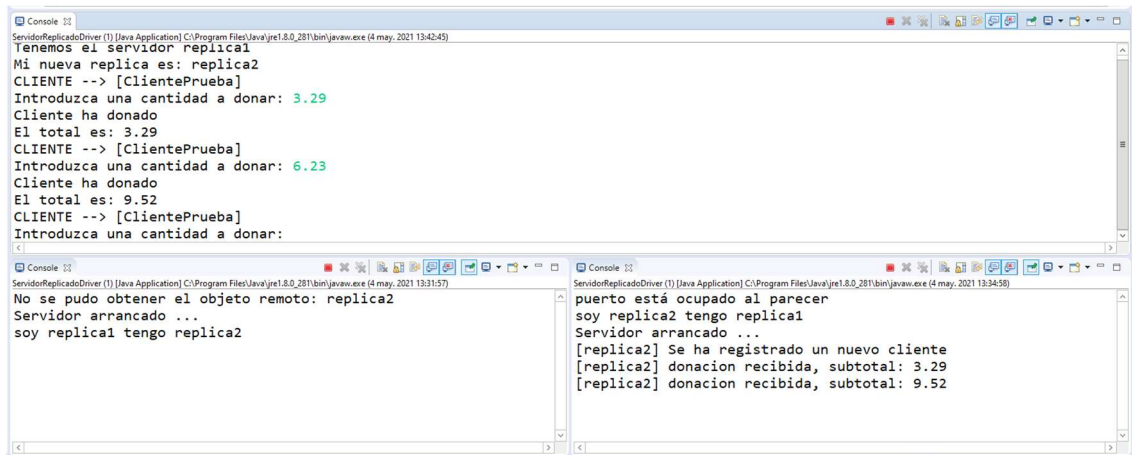
Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:31:57)
No se pudo obtener el objeto remoto: replica2
Servidor arrancado ...
soy replica1 tengo replica2

Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:34:58)
puerto está ocupado al parecer
soy replica2 tengo replica1
Servidor arrancado ...
[replica2] Se ha registrado un nuevo cliente
```

El cliente comienza teniendo la primera réplica como su réplica, le hace una petición a esta y se le acaba asignando la replica2. La segunda replica registra al nuevo cliente y lo notifica por pantalla. Tenemos ahora dos réplicas comunicadas entre sí y un cliente registrado con una réplica asignado listo para hacer donaciones.



Probemos ahora a hacer unas cuantas donaciones. Al hacer una donación, el cliente invoca el método `donar` del objeto remoto de su réplica y esta actualiza su campo `subtotal`, acto seguido (en el `main`) el cliente invoca el método **`getTotal()`** de su réplica y esta devuelve la suma del campo `subtotal` de toda réplica.

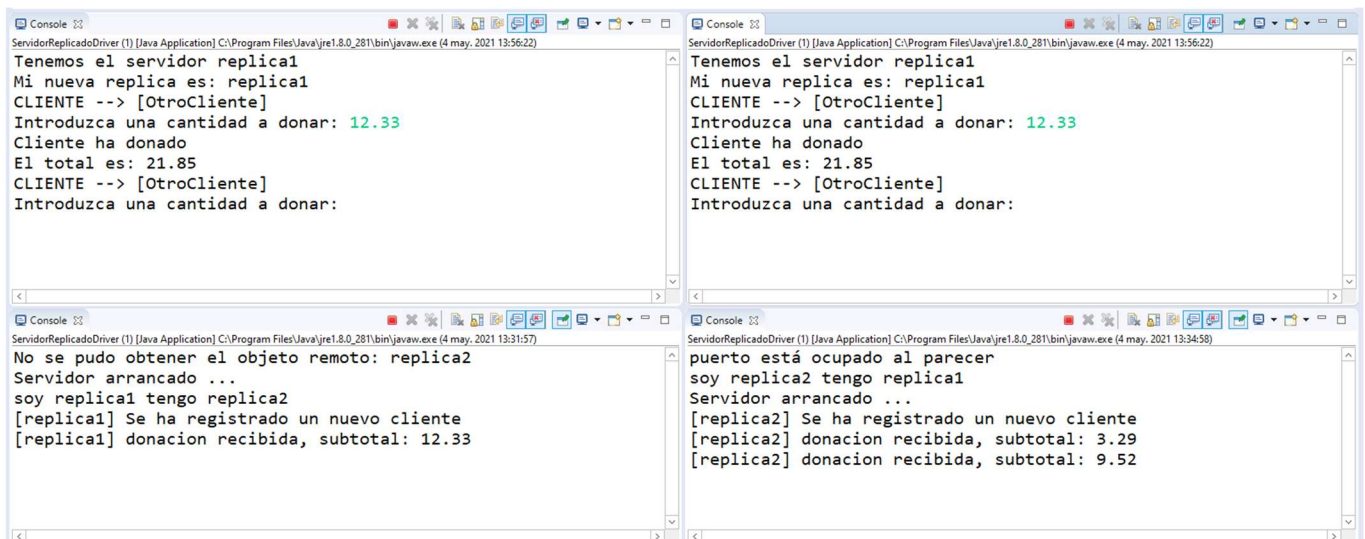


```
Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:42:45)
Tenemos el servidor replica1
Mi nueva replica es: replica2
CLIENTE --> [ClientePrueba]
Introduzca una cantidad a donar: 3.29
Cliente ha donado
El total es: 3.29
CLIENTE --> [ClientePrueba]
Introduzca una cantidad a donar: 6.23
Cliente ha donado
El total es: 9.52
CLIENTE --> [ClientePrueba]
Introduzca una cantidad a donar:

Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:31:57)
No se pudo obtener el objeto remoto: replica2
Servidor arrancado ...
soy replica1 tengo replica2

Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:34:58)
puerto está ocupado al parecer
soy replica2 tengo replica1
Servidor arrancado ...
[replica2] Se ha registrado un nuevo cliente
[replica2] donacion recibida, subtotal: 3.29
[replica2] donacion recibida, subtotal: 9.52
```

Para ver mejor el funcionamiento del método **`getTotal()`** vamos a crear otro cliente.



```
Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:56:22)
Tenemos el servidor replica1
Mi nueva replica es: replica1
CLIENTE --> [OtroCliente]
Introduzca una cantidad a donar: 12.33
Cliente ha donado
El total es: 21.85
CLIENTE --> [OtroCliente]
Introduzca una cantidad a donar:

Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:31:57)
No se pudo obtener el objeto remoto: replica2
Servidor arrancado ...
soy replica1 tengo replica2
[replica1] Se ha registrado un nuevo cliente
[replica1] donacion recibida, subtotal: 12.33

Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:56:22)
Tenemos el servidor replica1
Mi nueva replica es: replica1
CLIENTE --> [OtroCliente]
Introduzca una cantidad a donar: 12.33
Cliente ha donado
El total es: 21.85
CLIENTE --> [OtroCliente]
Introduzca una cantidad a donar:

Console: ServidorReplicadoDriver (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (4 may. 2021 13:34:58)
puerto está ocupado al parecer
soy replica2 tengo replica1
Servidor arrancado ...
[replica2] Se ha registrado un nuevo cliente
[replica2] donacion recibida, subtotal: 3.29
[replica2] donacion recibida, subtotal: 9.52
```

En la primera réplica se registra el nuevo cliente. Hacemos una donación desde el nuevo cliente y vemos que la recibe “replica1”. El nuevo cliente recibe el total de donaciones, que según vemos es la suma de todas las donaciones.

En conclusión, el sistema funciona como se esperaba, resuelve el problema que se plantea en el enunciado del ejercicio. No obstante, esta es una solución que deja mucho que desear:

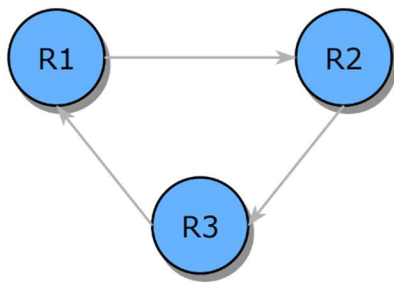
- No es escalable.
- Si una réplica pierde la conexión con otra réplica no se sabe.
- Si un cliente falla la conexión con la primera réplica abortaría su ejecución.

Para solventar algunos de los defectos de esta solución, vamos a proponer una versión extendida que además garantice el acceso en exclusión mutua.

## VERSIÓN EXTENDIDA

La versión extendida parte de la versión básica, en la que teníamos dos réplicas siempre fijas y un conjunto de clientes que se registra en una de ellas y realiza peticiones. En la versión extendida cambiamos ligeramente el planteamiento anterior, en este caso tendremos un número variable de réplicas que puede ir desde 1 hasta “infinito” (teóricamente hablando).

Inspirándonos en el algoritmo de elección en anillo, organizamos las réplicas en forma de anillo lógico, es un anillo lógico porque no hay ninguna conexión física como tal y el orden de estas es totalmente independiente. Nuestro sistema no tiene la figura de “líder” o “coordinador”, cada nodo podría serlo.

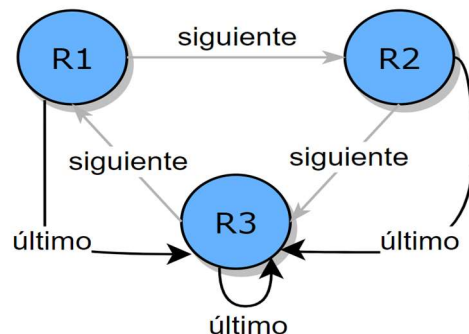


Un criterio para ordenar este anillo podría ser la localización geográfica, que cada réplica se comunicase con la más próxima a ella, así reduciríamos los tiempos de comunicación. En nuestro caso el criterio de ordenación del anillo lógico es meramente temporal, es decir, cada réplica se comunica con la que se añadió después de ella y la última en añadirse se comunicará con la primera.

R1 fue añadido antes que R2, quien fue añadido antes que R3, quien apunta a la primera réplica.

Cada réplica (a partir de ahora podemos llamarlas nodos indistintamente) apunta a la siguiente réplica, en la implementación se usa una referencia a objeto remoto. También tenemos que cada réplica apunta (o sabe) quién es el último nodo que se añadió.

Como podemos ver en la imagen de la derecha, incluso la última réplica indica que ella es la última.



A diferencia de la anterior versión, en esta no tenemos un subtotal para cada réplica, en su lugar tenemos un dato compartido entre todas ellas, que es el total, dicho total se actualiza en cada donación y se implementan mecanismos de exclusión mutua para que dicho dato sea siempre consistente.

Las operaciones más importantes de nuestro sistema son las siguientes:

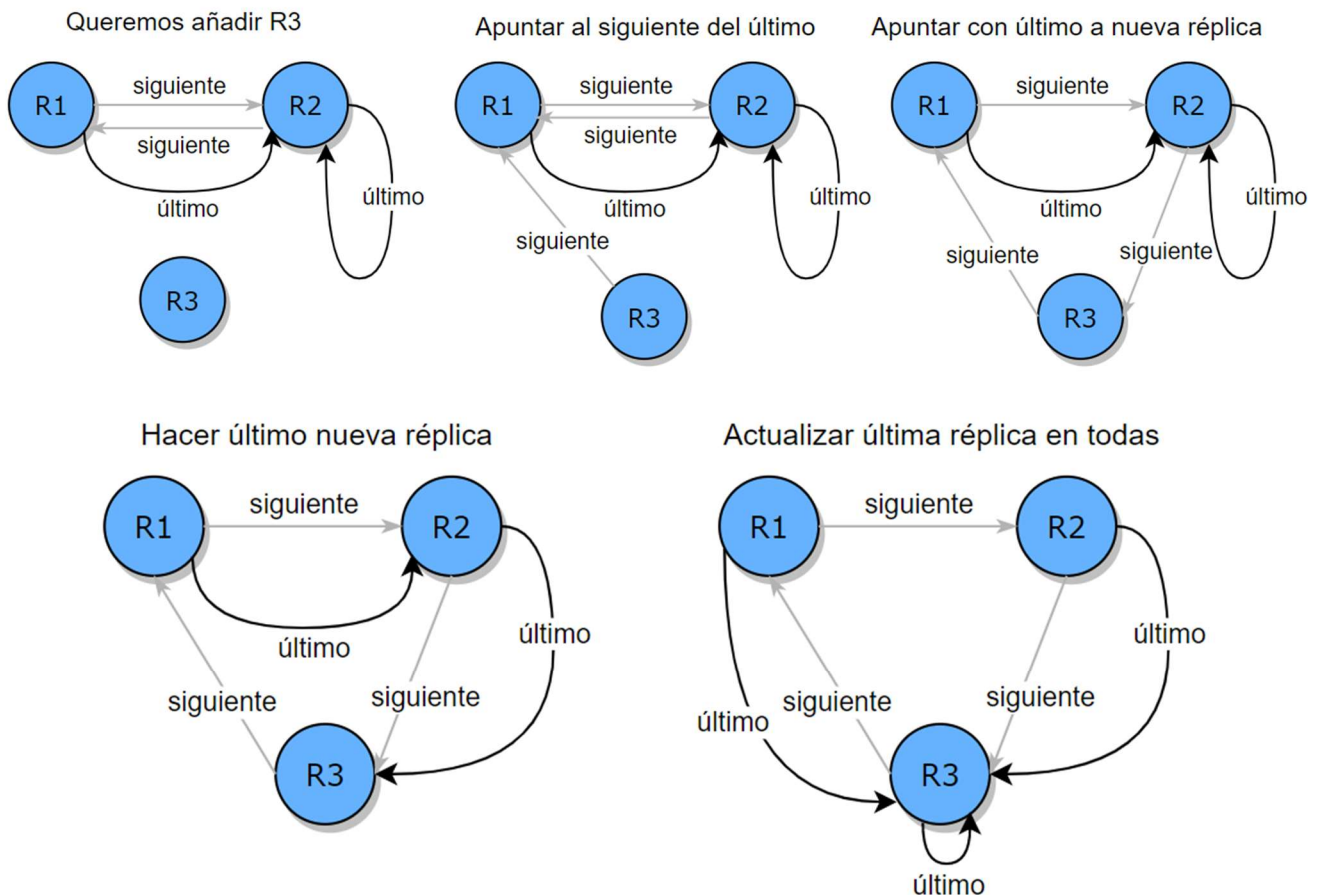
- Añadir una nueva réplica al conjunto de réplicas
- Registrar un nuevo cliente
- Buscar la réplica con menos clientes (menos cargada)
- Actualizar en todas las réplicas el total de donaciones consistentemente
- Detectar si una réplica cayó y dar cobertura a sus clientes y a las demás réplicas
- Detectar cuando el cliente no puede conectarse al servidor y tomar medidas para ello



### Añadir nueva réplica

Para esto llevamos a cabo una operación de inserción en listas indexadas, es decir, tenemos un anillo lógico en el que cada nodo “apunta” al siguiente y el último al primero. Para añadir un nuevo nodo, lo hacemos entonces como si fuese una inserción en una lista enlazada. El algoritmo sería el siguiente:

- Obtener último nodo
- Obtener siguiente nodo del último nodo (es el nodo primero)
- Apuntar la nueva réplica a dicho nodo siguiente
- Apuntar con el último nodo a la nueva réplica
- Hacer que último nodo sea la nueva réplica
- Informar a todas las demás réplicas del último



En las imágenes mostradas arriba podemos ver un ejemplo en donde queremos añadir a las réplicas R1 y R2 la réplica R3. Para añadir R2 a R1, se haría exactamente igual. Para añadir por primera vez una réplica, es decir, cuando no existe ninguna, se hace asignando todos los campos a la nueva réplica, es decir, la nueva réplica es la última, la primera, la siguiente, etc.

### **Registrar nuevo cliente**

Cualquier réplica puede recibir esta petición, cuando ocurre se comienza una fase de búsqueda en la que se pregunta si alguna réplica registró previamente al nuevo cliente, si es así se le asigna al cliente la réplica que lo registro previamente y se finaliza el procedimiento. Si el cliente no fue registrado previamente, entonces, se comienza una fase de búsqueda de la réplica con menor número de clientes. Cuando se obtiene dicha réplica, se añade el cliente a su lista de clientes y se le asigna la réplica al cliente que solicitó el registro.

### **Buscar la réplica con menos clientes (menos cargada)**

Este proceso de búsqueda es recursivo. Una réplica, invoca este método haciendo uso de la referencia remota a su siguiente réplica, le pasa como parámetro a ella misma (this), que hace referencia a la réplica de origen. Usaremos el parámetro “réplica de origen” como condición de parada, ya que el anillo lógico que usamos es cíclico y las operaciones son recursivas.

Dentro del procedimiento se pregunta si la réplica que ejecuta dicho procedimiento es la de origen, si lo es se devuelve a sí misma. Si no lo es, entonces se invoca nuevamente este procedimiento pasándole como réplica origen la que se pasó originalmente. Cuando esta invocación devuelve un objeto, quién la invoco compara el número de clientes de la réplica devuelta con el suyo y devuelve la réplica con menor número de clientes.

buscaMenorNumeroClientes(origen):

- Si no soy origen
  - replicaMenor = miSiguiente.buscaMenorNumeroClientes(origen)
  - si replicaMenor tiene más clientes que yo
    - Me devuelvo a mí mismo
  - En otro caso devuelvo replicaMenor
- Si soy origen
  - Me devuelvo a mí mismo

De aquí en adelante la mayoría de los métodos que implementen operaciones recursivas, tendrán la misma estructura definida en el algoritmo de arriba, es decir:

- **nombreMetodo(<posibles argumentos>, origen).**

Dicho método se invoca en una réplica de la siguiente manera:

- **siguienteReplica.nombreMetodo(<poisibles argumentos>, this);**

### **Actualizar en todas las réplicas el total de donaciones consistentemente**

Cuando una réplica recibe una donación se comienza el proceso de actualización del total. Este proceso realmente incrementa el valor del total en la cantidad que se especifique (dicha cantidad se corresponderá con la donación realizada).

Como hemos descrito en la página anterior, este es un método recursivo y tiene la estructura previamente descrita. Una réplica recibe una donación e invoca el método **actualizarTotal(cantidad, origen)** de su réplica siguiente. Se realiza el proceso recursivamente hasta llegar a origen, en cada iteración (invocación realmente) se hace la operación **total += cantidad**.

Para resolver los problemas de exclusión mutua, delegamos en los **synchronized statements**, un mecanismo que nos provee RMI. Mediante este mecanismo, garantizamos el acceso único a una sección crítica de código. De esta manera, evitamos complicar el diseño del sistema, lo hacemos más fiable y menos propenso a errores debido a que usamos una herramienta testeada y verificada por más usuarios, es decir, no es un modelo experimental.

Con este mecanismo permitimos que cualquier proceso invoque el método actualizar (incluso en distinto orden) y este se lleve a cabo de manera normal y corriente, pero cuando se va a ejecutar la sentencia de escritura en la variable, entonces usamos los **synchronized statements**. Solo un proceso escribe a la vez en la variable “total” y el orden es indiferente ya que es una suma y eso no altera el resultado final.

### **Detectar si una réplica cayó y dar cobertura a sus clientes y a las demás réplicas**

Para detectar cuando cayó una réplica, hacemos uso de los **RemoteException**, cuando obtenemos un código de error del tipo “**connection refused**” y no podemos hacer uso de la réplica correspondiente, entonces, comenzamos un proceso de sustitución de la réplica caída.

Debido a la arquitectura de nuestro sistema, la réplica que antes notará que cayó un nodo, es la réplica anterior al nodo caído, es decir, aquella que “apuntaba” al nodo caído. Cuando esto ocurre se invoca un procedimiento llamado, **ocuparLugarSiguierte**. En dicho proceso se informa de que cayó la réplica siguiente y lleva a cabo una operación de dar de baja de RMI a la réplica caída (conocemos su nombre). Una vez dada de baja dicha réplica, registramos bajo el mismo nombre (de la réplica caída) a la réplica anterior a esta y obtenemos el objeto remoto que era el siguiente del siguiente (también conocemos su nombre).

El resultado final, es una réplica que responde a varios nombres de objeto remoto. Así conseguimos que el sistema siga funcionando como si nada hubiese pasado. Los clientes de dicha réplica caída podrán seguir realizando peticiones sin notar la diferencia y recibiendo las mismas respuestas.

### **Detectar cuando el cliente no puede conectarse al servidor y tomar medidas para ello**

Cuando es un cliente quien detecta que su réplica asignada ha caído, entonces solicita registrarse en la réplica “origen”. La réplica origen es una réplica que se crea automáticamente cuando se intenta añadir una réplica y se asocia a la primera réplica añadida. Su nombre “origen” no varía ya que no se recibe de ninguna entrada, por lo que siempre existirá mientras exista alguna réplica. En la petición de registro del nuevo cliente (realmente estaba registrado) se detectará que cayó dicha réplica y se ocupará su lugar.

## Casos prácticos de ejecución

Todos los detalles de implementación están en el código que se adjunta con este fichero. El código está lo suficientemente comentado como para entender el funcionamiento de los métodos y que hace cada bloque de código.

Procedemos ahora a mostrar un caso práctico de uso. Vamos a lanzar una réplica, para esto lo hacemos como en la versión básica, solo que en este caso necesitaremos un argumento en lugar de dos. Dicho argumento se corresponde con el nombre de la réplica.

```
Console
replica1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 11:38:18)
No se encontró origen y se tuvo que crear: origen
Soy el primero nodo
[replica1] siguiente replica1 siguiente del siguiente replica1
Servidor arrancado ...
```

Llamamos “replica1” a nuestra primera réplica, al lanzarla nos informa que no existe la réplica origen, por lo que se ha creado y asociado a “replica1”. Nos informa de los nombres de sus siguientes réplicas, que al ser la primera y única, es ella.

```
Console
replica2 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 11:50:04)
puerto ya existente
[replica2] siguiente replica1 siguiente del siguiente replica2
Servidor arrancado ...
```

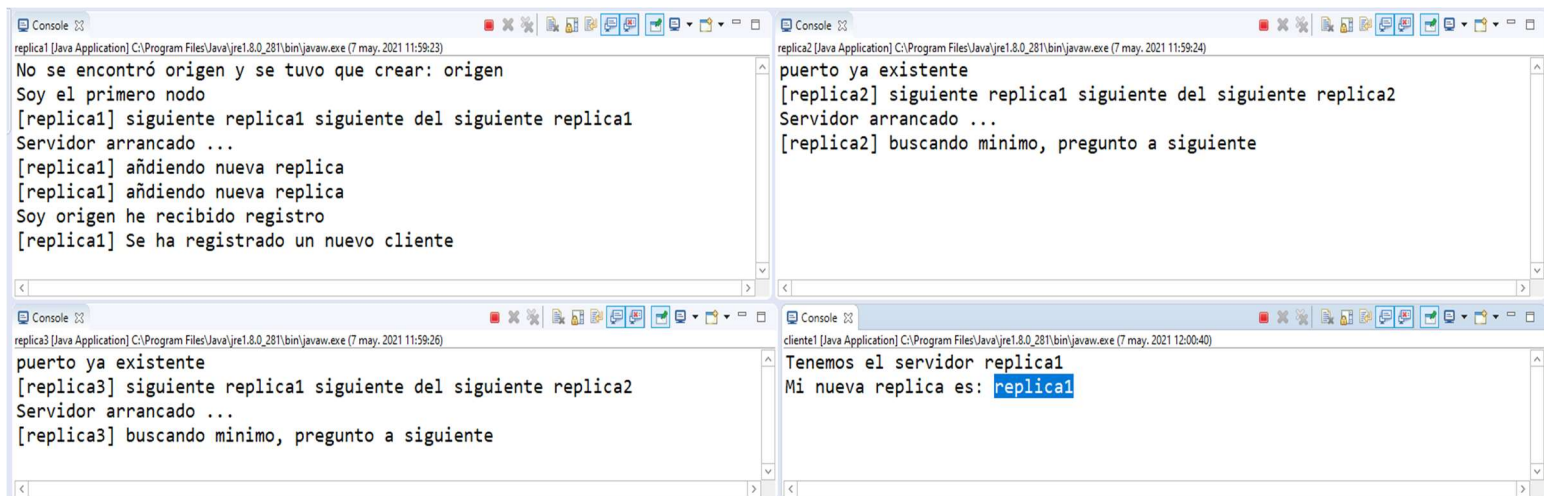
```
Console
replica1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 11:38:18)
No se encontró origen y se tuvo que crear: origen
Soy el primero nodo
[replica1] siguiente replica1 siguiente del siguiente replica1
Servidor arrancado ...
[replica1] añadiendo nueva replica
```

Lanzamos una nueva réplica llamada “replica2”. Nos informa de que el puerto ya existe, por lo que lo obtiene en lugar de crearlo. Para añadirse al conjunto de réplicas invoca al método añadir réplica del objeto remoto “origen”. Origen se encuentra asociado por ahora a “replica1”. Vemos que replica 1 ha recibido dicha petición y ha sido capaz de añadirla correctamente. Replica2 nos muestra el nombre de sus siguientes réplicas y vemos que es correcto.

```
Console
replica3 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 11:54:46)
puerto ya existente
[replica3] siguiente replica1 siguiente del siguiente replica2
Servidor arrancado ...
```

Añadimos una tercera réplica y vemos que el proceso es igual que en el paso anterior. Nos muestra sus siguientes réplicas y vemos que son correctas, por lo que se ha creado correctamente.

Vamos a ejecutar un nuevo cliente llamado “cliente1”, el nombre de los clientes es meramente simbólico, podría no pasarse ningún argumento al ejecutar ClienteDriver y el resultado sería el mismo, solo que usaríamos el nombre por defecto del cliente (“anónimo”).



```
replica1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may, 2021 11:59:23)
No se encontró origen y se tuvo que crear: origen
Soy el primero nodo
[replica1] siguiente replica1 siguiente del siguiente replica1
Servidor arrancado ...
[replica1] añadiendo nueva replica
[replica1] añadiendo nueva replica
Soy origen he recibido registro
[replica1] Se ha registrado un nuevo cliente

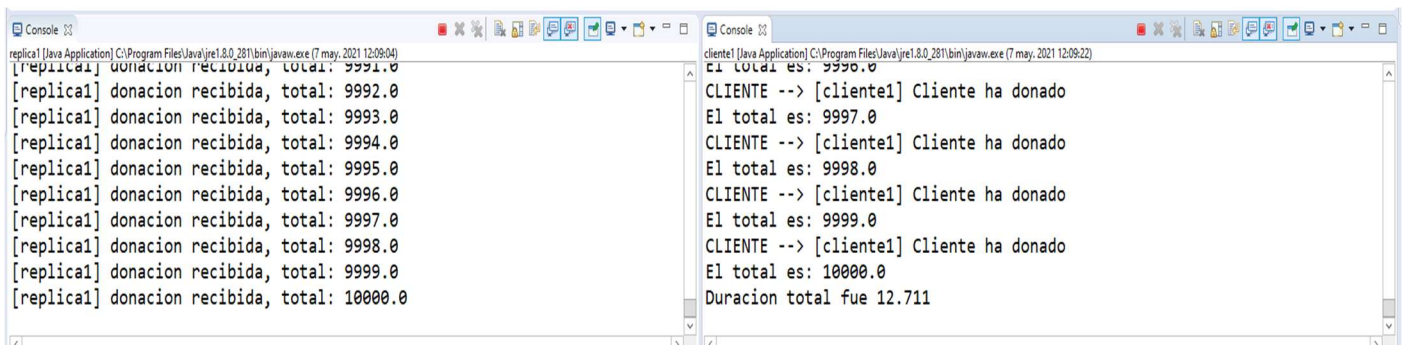
replica2 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may, 2021 11:59:24)
puerto ya existente
[replica2] siguiente replica1 siguiente del siguiente replica2
Servidor arrancado ...
[replica2] buscando minimo, pregunto a siguiente

replica3 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may, 2021 11:59:26)
puerto ya existente
[replica3] siguiente replica1 siguiente del siguiente replica2
Servidor arrancado ...
[replica3] buscando minimo, pregunto a siguiente

cliente1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may, 2021 12:00:40)
Tenemos el servidor replica1
Mi nueva replica es: replica1
```

Al lanzar el cliente, este por defecto realiza una invocación al objeto “origen” dicho objeto está asociado en este caso a “replica1”, por lo que vemos en cliente1 que tiene el servidor replica1. Realiza pues una invocación de registro a replica1 y este comienza la fase de búsqueda del nodo con menor número de clientes. Podemos ver que las demás réplicas han recibido una invocación y han realizado la misma a sus siguientes réplicas como se describió anteriormente.

Para probar el sistema vamos a ejecutar automáticamente en cada cliente un número de donaciones y una cantidad. En nuestro caso, se realizarán por cada cliente 10.000 donaciones y cada donación será de “1” (euro, item, moneda, la unidad que sea, etc).



```
replica1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may, 2021 12:09:04)
[replica1] donacion recibida, total: 9991.0
[replica1] donacion recibida, total: 9992.0
[replica1] donacion recibida, total: 9993.0
[replica1] donacion recibida, total: 9994.0
[replica1] donacion recibida, total: 9995.0
[replica1] donacion recibida, total: 9996.0
[replica1] donacion recibida, total: 9997.0
[replica1] donacion recibida, total: 9998.0
[replica1] donacion recibida, total: 9999.0
[replica1] donacion recibida, total: 10000.0

cliente1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may, 2021 12:09:22)
El total es: 9996.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 9997.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 9998.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 9999.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 10000.0
Duracion total fue 12.711
```

El cliente realiza las 10.000 donaciones y las recibe su réplica. Podemos ver que el total es correcto ya que el cliente pregunta en cada iteración cuál es el total y vemos que coincide con la cantidad esperada. Esto es trivial pues no ha habido realmente concurrencia ni se han podido pisar las donaciones. Vamos a añadir dos clientes más, los llamaremos cliente2 y un nuevo cliente1. Estos harán donaciones de manera concurrente a sus respectivas réplicas.

Como vemos en la imagen siguiente, cliente2 fue asignado a replica3 y cliente1 fue asignado a replica2. Realizan donaciones de manera concurrente y distribuida. Sumadas a las 10.000 donaciones anteriores, tenemos las 10.000 de cada cliente, que en total da 30.000, por lo que podemos asegurar que se ha mantenido consistente el dato “total”.



```
cliente1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:13:59)
El total es: 29996.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 29997.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 29998.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 29999.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 30000.0
Duracion total fue 10.233

replica2 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:09:06)
[replica2] donacion recibida, total: 29991.0
[replica2] donacion recibida, total: 29992.0
[replica2] donacion recibida, total: 29993.0
[replica2] donacion recibida, total: 29994.0
[replica2] donacion recibida, total: 29995.0
[replica2] donacion recibida, total: 29996.0
[replica2] donacion recibida, total: 29997.0
[replica2] donacion recibida, total: 29998.0
[replica2] donacion recibida, total: 29999.0
[replica2] donacion recibida, total: 30000.0

replica3 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:09:07)
[replica3] donacion recibida, total: 26753.0
[replica3] donacion recibida, total: 26755.0
[replica3] donacion recibida, total: 26757.0
[replica3] donacion recibida, total: 26759.0
[replica3] donacion recibida, total: 26761.0
[replica3] donacion recibida, total: 26763.0
[replica3] donacion recibida, total: 26765.0
[replica3] donacion recibida, total: 26767.0
[replica3] donacion recibida, total: 26769.0
[replica3] donacion recibida, total: 26771.0
```

Vamos a lanzar las 3 réplicas de nuevo, en esta ocasión vamos a comprobar que ocurre cuando cae una réplica. Lanzamos un cliente y mientras realiza las donaciones y se intente mantener consistente el dato “total” cae una réplica del conjunto. Como podemos ver en la imagen:

```
replica1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:20:24)
[replica1] donacion recibida, total: 6792.0
[replica1] donacion recibida, total: 6794.0
[replica1] donacion recibida, total: 6795.0
[replica1] donacion recibida, total: 6796.0
[replica1] donacion recibida, total: 6797.0
[replica1] donacion recibida, total: 6798.0
[replica1] donacion recibida, total: 6799.0
[replica1] donacion recibida, total: 6800.0

replica2 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:20:26)
puerto ya existente
[replica2] siguiente replica1 siguiente del siguiente replica2
Servidor arrancado ...
[replica2] buscando minimo, pregunto a siguiente
[replica2] cayo replica3, ocupando lugar
[replica2] siguiente replica1 siguiente del siguiente replica2

cliente1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:21:05)
El total es: 6345.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 6346.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 6347.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 6348.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 6349.0
CLIENTE --> [cliente1] Cliente ha donado
```

Hemos tumbado replica3 mientras cliente1 hacía donaciones a replica1, en ese instante replica2, quien era la anterior a replica3, se da cuenta y ocupa su lugar.

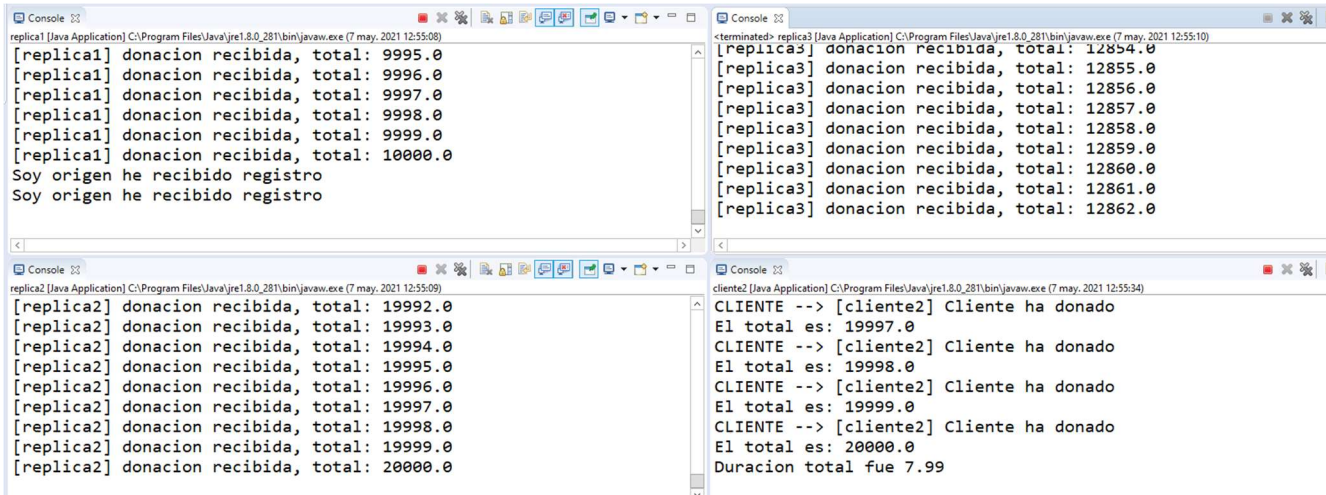
```
cliente1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:21:05)
El total es: 9996.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 9997.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 9998.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 9999.0
CLIENTE --> [cliente1] Cliente ha donado
El total es: 10000.0
Duracion total fue 10.262
```

El resultado final es correcto, a pesar de que cayó una réplica del anillo, no se vio afectado el dato “total” y otra réplica fue capaz de detectar y subsanar (al menos reducir el daño, ya que el sistema está funcionando con una réplica menos) sin que afectase al cliente1.



Dicho cliente no hacía donaciones directamente a replica3. Veamos qué pasaría en dicho caso.

Para ver qué sucede cuando una réplica con clientes asignados los cuales se encuentran haciendo peticiones cae, lanzamos desde 0 las 3 réplicas y lanzamos un cliente1. Cuando finaliza cliente1 tenemos 10.000 donaciones hechas (a replica1). Lanzamos ahora un cliente2, este se asigna a réplica3. Mientras realiza las donaciones tumbamos replica3.



```
replica1 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:55:08)
[replica1] donacion recibida, total: 9995.0
[replica1] donacion recibida, total: 9996.0
[replica1] donacion recibida, total: 9997.0
[replica1] donacion recibida, total: 9998.0
[replica1] donacion recibida, total: 9999.0
[replica1] donacion recibida, total: 10000.0
Soy origen he recibido registro
Soy origen he recibido registro

replica2 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:55:09)
[replica2] donacion recibida, total: 19992.0
[replica2] donacion recibida, total: 19993.0
[replica2] donacion recibida, total: 19994.0
[replica2] donacion recibida, total: 19995.0
[replica2] donacion recibida, total: 19996.0
[replica2] donacion recibida, total: 19997.0
[replica2] donacion recibida, total: 19998.0
[replica2] donacion recibida, total: 19999.0
[replica2] donacion recibida, total: 20000.0

cliente2 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:55:34)
<terminated> replica3 [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (7 may. 2021 12:55:10)
[replica3] donacion recibida, total: 12854.0
[replica3] donacion recibida, total: 12855.0
[replica3] donacion recibida, total: 12856.0
[replica3] donacion recibida, total: 12857.0
[replica3] donacion recibida, total: 12858.0
[replica3] donacion recibida, total: 12859.0
[replica3] donacion recibida, total: 12860.0
[replica3] donacion recibida, total: 12861.0
[replica3] donacion recibida, total: 12862.0
CLIENTE --> [cliente2] Cliente ha donado
El total es: 19997.0
CLIENTE --> [cliente2] Cliente ha donado
El total es: 19998.0
CLIENTE --> [cliente2] Cliente ha donado
El total es: 19999.0
CLIENTE --> [cliente2] Cliente ha donado
El total es: 20000.0
Duracion total fue 7.99
```

Replica3 estaba recibiendo las donaciones de cliente2, esta fue tumbada, el cliente detectó este suceso e invocó un registro a la réplica origen. En el proceso de registrar al nuevo cliente, la réplica anterior a la replica3 (replica2) detecta que cayó y ocupa su lugar. El resultado es satisfactorio, tenemos realizadas las 20.000 donaciones esperadas.

## Conclusiones

El sistema está muy lejos de ser perfecto. Algunos de los inconvenientes que presenta están relacionados con que se caigan las réplicas. Por ejemplo:

- Si cae una réplica y se intenta ocupar su lugar, pero la siguiente a la réplica caída también cayó, no se podrá saber quién era la siguiente a estas y por lo tanto la arquitectura de anillo queda rota, al igual que todo el sistema.
- Cuando cae una réplica y se ocupa su lugar por otra réplica, su estado se pierde. Para recuperar su estado se debería tener en cada réplica una copia (clonación) de la réplica siguiente. Esto es costoso, ya que, al ser una arquitectura en anillo, eso supondría la posibilidad de una copia recursiva infinita, además de un alto coste de procesamiento para poder actualizar dichos datos cada vez que se modificasen. La solución ideal sería que el estado de las réplicas fuese guardado en almacenamiento periódicamente, tal vez en una base de datos a la que pudiesen acceder todas las réplicas.
- Cuando una réplica sustituye a otra, les da cobertura a sus clientes, estos podrán seguir realizando peticiones, no obstante, esta réplica estará sobrecargada y ni si quiera podemos saber cuánto pues desconocemos el número de clientes. Una solución sería un balanceador de carga que periódicamente redistribuyese la carga.
- El nodo “origen” es el más importante y el más expuesto. Puede estar asociado a cualquier réplica, pero si cae y no se recupera el sistema colapsaría inevitablemente, por lo que centraliza un tanto la arquitectura y la hace peligrar.

Como hemos visto este es un sistema que está muy lejos de ser perfecto y que se puede refinar bastante. Requeriría más tiempo y conocimientos para ello.

Este sistema se basa en el algoritmo de elección en anillo. A diferencia de dicho algoritmo no tenemos la figura de líder o coordinador como tal ya que implementamos la exclusión mutua de manera diferente (haciendo uso de mecanismos del lenguaje de programación). Sí usamos el sistema de comunicación mediante rondas, es decir, cada nodo invoca al siguiente hasta que se llega al que originó la ronda de comunicación (lo hemos descrito previamente).

Para cualquier duda o aclaración no dude en escribirme convocarme a una reunión si lo ve pertinente o un correo a [ahmedmkoubaa@correo.ugr.es](mailto:ahmedmkoubaa@correo.ugr.es)

### **Detalles de los requisitos técnicos**

- JDK 8.0 o superior
- Eclipse JAVA cualquier versión (recomendada posterior a 2019)
- Se puede compilar el código por línea de comandos

### **Enlaces de interés**

[https://en.wikipedia.org/wiki/Java\\_remote\\_method\\_invocation](https://en.wikipedia.org/wiki/Java_remote_method_invocation)

<https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>

<http://lsd.ls.fi.upm.es/educationoldfolder/sistemas-distribuidos-fundamentos-y-tecnologia-doctorado-distributed-systems-phd-level/course-2009-2010/diapositivas-ernesto/eleccionLider.pdf>

<https://www.arcos.inf.uc3m.es/infodsd/wp-content/uploads/sites/38/2017/10/Algoritmos-distribuidos.pdf>

[https://es.wikipedia.org/wiki/Exclusi%C3%B3n\\_mutua\\_\(inform%C3%A1tica\)#::~:~:text=La%20t%C3%A9cnica%20que%20se%20emplea,mitad%20de%20la%20secci%C3%B3n%20cr%C3%ADtica](https://es.wikipedia.org/wiki/Exclusi%C3%B3n_mutua_(inform%C3%A1tica)#::~:~:text=La%20t%C3%A9cnica%20que%20se%20emplea,mitad%20de%20la%20secci%C3%B3n%20cr%C3%ADtica)