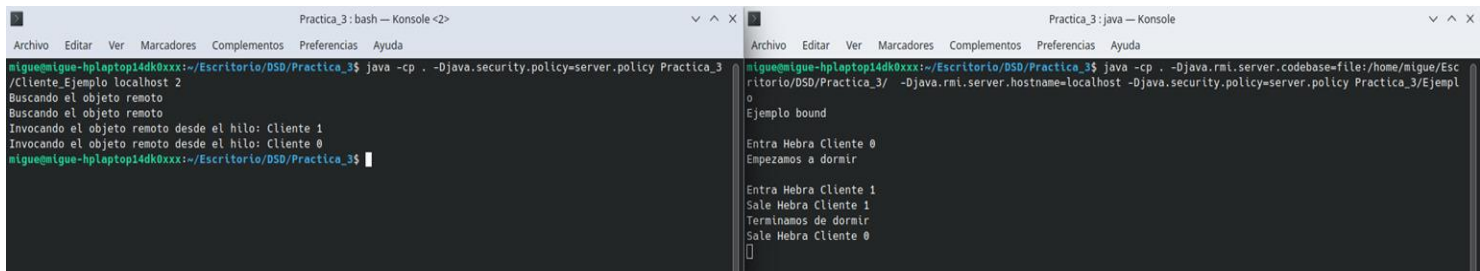


# Ejercicio 1

El programa simula un cliente que se conecta a un servidor remoto utilizando RMI (Remote Method Invocation), un mecanismo que permite que un programa en una máquina invoque métodos en un objeto que se ejecuta en otra máquina, de forma transparente. Este cliente puede ejecutar múltiples hilos, lo que significa que puede hacer múltiples peticiones simultáneas al servidor. El propósito es realizar invocaciones concurrentes, lo cual es útil para probar la capacidad del servidor de manejar múltiples solicitudes de manera eficiente. El cliente puede hacer n peticiones simultáneas al servidor, y cada hilo del cliente se comporta como un "usuario independiente" realizando una acción en el servidor.



```
Practica_3: bash — Konsole <2>
Archivo Editar Ver Marcadores Complementos Preferencias Ayuda
mlgue@mlgue-hplaptop14dk0xxx:~/Escritorio/DSD/Practica_3$ java -cp . -Djava.security.policy=server.policy Practica_3
/ClienteEjemplo localhost 2
Buscando el objeto remoto
Buscando el objeto remoto
Invocando el objeto remoto desde el hilo: Cliente 1
Invocando el objeto remoto desde el hilo: Cliente 0
mlgue@mlgue-hplaptop14dk0xxx:~/Escritorio/DSD/Practica_3$

Practica_3: java — Konsole
Archivo Editar Ver Marcadores Complementos Preferencias Ayuda
mlgue@mlgue-hplaptop14dk0xxx:~/Escritorio/DSD/Practica_3$ java -cp . -Djava.rmi.server.codebase=file:/home/mlgue/Escritorio/DSD/Practica_3/ -Djava.rmi.server.hostname=localhost -Djava.security.policy=server.policy Practica_3/Ejemplo
Ejemplo bound
Entra Hebra Cliente 0
Empezamos a dormir
Entra Hebra Cliente 1
Sale Hebra Cliente 1
Terminamos de dormir
Sale Hebra Cliente 0
[]
```

## Ejemplo\_I.java

La interfaz Ejemplo\_I extiende la interfaz Remote de Java RMI. Esto es necesario porque en RMI, los objetos remotos deben implementar esta interfaz. La interfaz Remote actúa como un marcador que indica que los objetos de esa clase pueden ser invocados de manera remota a través de la red. Permite permitir que un objeto sea accedido desde una máquina diferente, debe ser declarado como un objeto "remoto".

En este caso, la interfaz Ejemplo\_I extiende Remote y tiene un método escribir\_mensaje, que toma un String mensaje como argumento y puede lanzar una RemoteException. Esto es típico en aplicaciones distribuidas para indicar que la llamada remota podría fallar debido a problemas de red, por ejemplo.

- ⑩ public interface Ejemplo\_I extends Remote : Define una interfaz pública Ejemplo\_I que extiende Remote. Esto significa que cualquier implementación de esta interfaz debe ser accesible de forma remota.
- ⑩ public void escribir\_mensaje(String mensaje) throws RemoteException;: Es el método que define la acción remota: en este caso, escribir un mensaje. Como cualquier operación remota, puede lanzar una RemoteException si algo sale mal durante la comunicación remota.

## Ejemplo.java

La clase Ejemplo implementa la interfaz Ejemplo\_I:El método escribir\_mensaje está implementado para mostrar un mensaje en la consola. Si el mensaje termina con el número "0", el servidor "duerme" durante 5 segundos usando Thread.sleep(5000) antes de continuar . Si hay algún error en el proceso se captura y muestra un mensaje de error.

Utilizando `UnicastRemoteObject.exportObject(prueba, 0)`, el servidor hace que el objeto prueba sea accesible de manera remota a través de RMI. El segundo parámetro (0) indica que el objeto debe escuchar en un puerto aleatorio. Se obtiene el registro RMI utilizando `LocateRegistry.getRegistry()`, que permite acceder al registro de objetos remotos en el servidor.

Se registra el objeto remoto con un nombre identificador ("ServidorMensaje") en el registro RMI. Esto significa que los clientes pueden buscar este nombre para invocar métodos sobre el servidor: `registry.rebind(nombre_objeto_remoto, stub)`;

Si todo se ha configurado correctamente, el mensaje "Ejemplo bound" se imprime en la consola, lo que indica que el servidor está en funcionamiento y listo para aceptar solicitudes.

## **Cliente\_Ejemplo.java**

Implementa `Runnable`, lo que permite que se ejecute en un hilo separado. Cada vez que se ejecuta el cliente, se crea un hilo que hace la llamada al servidor. Este enfoque permite realizar invocaciones concurrentes al servidor, lo cual es útil si el servidor está diseñado para manejar múltiples peticiones al mismo tiempo.

Método `run()`: Este método se ejecuta cuando el hilo es iniciado. Se conecta al registro RMI del servidor utilizando el método `LocateRegistry.getRegistry(server)` y busca el objeto remoto con `registry.lookup(nombre_objeto_remoto)`. Una vez que obtiene una referencia al objeto remoto, invoca el método `escribir_mensaje` pasando el nombre del hilo actual (`Thread.currentThread().getName()`).

## **Que ocurre con las hebras cuyo nombre acaba en 0? ¿Qué hacen las demás hebras? ¿Se entrelazan los mensajes?**

El programa comprueba si el mensaje (que es el nombre del hilo) termina en "0" y, si es así, hace que la hebra se "duerma" durante 5 segundos:

```
if (mensaje.endsWith("0")) {  
    try {  
        System.out.println("Empezamos a dormir");  
        Thread.sleep(5000); // Duerme la hebra durante 5 segundos  
        System.out.println("Terminamos de dormir");  
    } catch (Exception e) {  
        System.err.println("Ejemplo exception:");  
        e.printStackTrace();  
    }  
}
```

El hilo entra en una pausa durante 5 segundos. Esto se logra mediante el método `Thread.sleep(5000)`. Durante este tiempo, el hilo no hace nada más; simplemente espera. Esto simula una situación donde un hilo realiza una tarea costosa o bloqueante que requiere esperar por un tiempo determinado.

Para las hebras cuyo nombre no termina en "0", ejecutan el código normalmente sin entrar en la pausa de 5 segundos. Realizan la invocación del método remoto `escribir_mensaje` y luego continúan con su ejecución.

Esto significa que estas hebras se completan mucho más rápido que las hebras que entran en la pausa de 5 segundos. Sin embargo, todas las hebras aún comparten el mismo recurso de la consola (ya que todas escriben en ella), lo que genera la posibilidad de que sus mensajes se entrelacen .

### ¿Se entrelazan los mensajes?

Y efectivamente los mensajes se entrelazan debido a que todos los hilos están escribiendo en la misma consola (o flujo estándar de salida), y los hilos están ejecutándose en paralelo. Esto significa que las líneas de texto de diferentes hilos se mezclan en la consola de manera no controlada.

1. El hilo 1 (que no termina en "0") empieza a escribir en la consola.
2. El hilo 2 (que termina en "0") comienza, pero entra en la pausa de 5 segundos.
3. Mientras el hilo 2 está "durmiendo", los hilos 3 y 4 (que no terminan en "0") siguen escribiendo en la consola.

El orden en el que los mensajes se escriben puede variar y no es necesariamente el mismo que el orden de creación de los hilos, debido a la ejecución concurrente. Un ejemplo de salida entrelazada podría ser:

```
Cliente 1: Buscando el objeto remoto
Cliente 2: Buscando el objeto remoto
Cliente 3: Buscando el objeto remoto
Cliente 2: Empezamos a dormir
Cliente 3: Invocando el objeto remoto desde Cliente 3
Cliente 1: Invocando el objeto remoto desde Cliente 1
Cliente 2: Terminamos de dormir
Cliente 4: Buscando el objeto remoto
```

### Prueba a introducir el modificador synchronized en el método de la implementación remota

El modificador synchronized en Java se utiliza para garantizar que un bloque de código o un método solo pueda ser ejecutado por un hilo a la vez. Al aplicarlo a un método, como escribir\_mensaje, se asegura que, aunque varios hilos intenten ejecutar este método de manera concurrente, solo uno podrá hacerlo en un momento dado, y los demás deberán esperar su turno.

**Acceso Exclusivo al Método:** Al añadir synchronized en el método, aseguras que solo un hilo pueda ejecutarlo a la vez en cada instancia del objeto remoto. Si el servidor tiene varias instancias del objeto remoto, cada instancia tendrá su propia "zona crítica" para el método sincronizado.

**Bloqueo de Hilos Concurrentes:** Si varios hilos intentan llamar a escribir\_mensaje en el mismo objeto remoto al mismo tiempo, los hilos adicionales deben esperar a que el hilo actual termine su ejecución antes de poder acceder al método. Esto introduce una forma de sincronización, lo que puede evitar ciertos problemas de concurrencia (como las condiciones de carrera) pero también puede llevar a bloqueos si hay muchos hilos esperando su turno.