Programación Distribuida y Tiempo Real

Alumnos:

- Felipe Dioguardi - 16211/4
- Julian Marques de Abrantes - 15966/0

PRACTICA 2

1. **Utilizando como base el programa ejemplo de RMI:**

    a. **Analice si RMI es de acceso completamente transparente (access transparency, tal como está definido en Coulouris-Dollimore-Kindberg). Justifique.**

    RMI no es de acceso completamente transparente, ya que para que lo sea se debería poder acceder a los objetos remotos mediante la misma interfaz que usa para acceder a los locales.

    Si bien al acceder a objetos remotos utilizando RMI se utilizan llamados a objetos comunes (igual que en la interacción local), para acceder a los remotos es necesario tener en cuenta aspectos relacionados a la conexión con el servidor, y a errores que se pudieran producir. Por ejemplo, hay que lidiar con `RemoteExceptions`, aspecto no requerido para el acceso local.

    b. **Enumere los archivos .class que deberían estar del lado del cliente y del lado del servidor y que contiene cada uno.**

    Del lado del servidor tienen que conocerse aquellos objetos que permiten levantar el servicio. Estos son: `IfaceRemoteClass` (que define los métodos que implementará el objeto remoto), `RemoteClass` (que implementa el comportamiento definido por la `IfaceRemoteClass`), y `StartRemoteObject` (que registra al objeto remoto y le asigna una dirección donde estará escuchando).

    El cliente necesita conocer la interfaz `IfaceRemoteClass` para saber la firma de los métodos que puede responder el objeto remoto. Además, debería tener alguna clase que utilice para efectuar la comunicación, a modo de cliente. En los archivos provistos, esta se llama `AskRemote`.

A su vez, ambos deberían conocer cualquier objeto que sea enviado o devuelto en la comunicación (o alguna interfaz que represente su comportamiento). Por ejemplo, si la interfaz `IfaceRemoteClass` define que el método `ClaseEjemplo read()` devuelve una instancia de `ClaseEjemplo`, entonces tanto el cliente como el servidor deberían tener el ClaseEjemplo.class.

2. **Investigue porque con RMI puede generarse el problema de desconocimiento de clases en las JVM e investigue como se resuelve este problema.**

   El problema del desconocimiento de clases puede producirse cuando se trabaja con RMI, dado que este permite que el servidor y el cliente no estén alojados en el mismo sitio. Esto implica que ambos intentarían interactuar por medio de objetos que alguno de los dos no sabe como manejar (no conoce su tipo ni los métodos que implementa).

   Como solución a este problema, Java implementa lo que se conoce como **descarga dinámica de código** (Dynamic Code Loading). Mediante este mecanismo, se obtiene la posibilidad de descargar la implementación específica de un objeto.

   ¿Cómo se consigue? Supongamos que un cliente quiere recibir un objeto que implementa la interfaz `IfacePaquete`. Para eso, llama al método `dameElPaquete()` de un servidor. El servidor, en forma de objeto remoto, implementaría el mensaje `IfacePaquete dameElPaquete()`, donde retornaría una instancia de un objeto que implemente la interfaz. Para que el paquete pueda ser enviado, necesita, además, implementar la interfaz `Serializable`. De otra manera, este no podría ser enviado mediante la red.

   Lo que sucedería al querer llamar al método `dameElPaquete()` es:

   1. El cliente envía el mensaje a su referencia del objeto remoto.
   2. El servidor recibe el mensaje y devuelve una instancia de la clase concreta `Paquete()`.
   3. El servidor serializa la instancia a devolver y la envía.
   4. La JVM del cliente la recibe y se da cuenta de que no conoce la implementación de esa clase.
   5. La JVM del cliente descarga el Paquete.class de una dirección especificada por el servidor (llamada `codebase`) al inicio de la conexión.

Entonces, el desconocimiento de clases puede producirse porque el cliente o el servidor no conozcan la clase de alguno de los objetos que usan. Para solucionarlo, se puede copiar la implementación de los objetos en ambos extremos de la conexión, o delegar el problema a la JVM permitiendo el Dynamic Code Loading. Cualquier programa que quiera enviar a otro programa un objeto desconocido por el receptor deberá fijar la propiedad `codebase`. De esta forma el receptor podrá conocer desde donde puede descargar dicho código, si no lo tiene disponible de forma local.

3. **Implementar con RMI el mismo sistema de archivos remoto implementado con RPC en la práctica anterior:**

   a. **Defina e implemente con RMI un servidor cuyo funcionamiento permita llevar a cabo desde un cliente las operaciones enunciadas informalmente como:**

   — leer: dado un nombre de archivo, una posición y una cantidad de bytes a leer, retorna

   1) La cantidad de bytes del archivo pedida a partir de la posición dada, o, en caso de haber menos bytes, se retornan los bytes que haya y

   2) la cantidad de bytes que efectivamente se retornan leídos.

   — escribir: dado un nombre de archivo, una cantidad de bytes determinada, y un buffer a partir del cual están los datos, se escriben los datos en el archivo dado. Si el archivo existe, los datos se agregan al final, si el archivo no existe, se crea y se le escriben los datos. En todos los casos se retorna la cantidad de bytes escritos.

   b. **Implemente un cliente RMI del servidor anterior que copie un archivo del sistema de archivos del servidor en el sistema de archivos local y genere una copia del mismo archivo en el sistema de archivos del servidor. En todos los casos se deben usar las operaciones de lectura y escritura del servidor definidas en el ítem anterior, sin cambios específicos del servidor para este ítem en particular. Al finalizar la ejecución del cliente deben quedar tres archivos en total: el original en el lado del servidor, una copia del original en el lado del cliente y una copia en el servidor del archivo original. El comando `diff` no debe identificar ninguna diferencia entre ningún par de estos tres archivos.**

Para realizar lo indicado creamos 3 paquetes. En `server` se encuentra el código correspondiente al servidor, con el `StartRemoteObject` que lo inicia, la implementación del objeto remoto `RemoteClass`, y la implementación del objeto que retornará el servidor en el mensaje `leer()`, `RemoteFile`. En `client` está la clase `AskRemote`, que funciona como cliente en la comunicación. Por último, en `shared` están definidas las interfaces `IfaceRemoteFile` e `IfaceRemoteClass`. Tanto el cliente como el servidor necesitan conocer estas interfaces para que puedan interactuar completamente.

Tanto el cliente como el servidor tienen una carpeta `files` en la que guardan sus archivos.

Para el manejo de archivos en Java utilizamos las clases `File`, `FileInputStream`, y `FileOutputStream`.

Agregamos una función en el objeto remoto que permita consultar si existe un archivo con cierto nombre o no. Con esto, el cliente pregunta si contiene el archivo a copiar, y de no ser así, le indica al servidor que lo escriba. Luego, inicia la lógica requerida por el punto (b):

1. El cliente llama al `leer()` del servidor, pidiendo todo el archivo.
2. El servidor recupera el archivo pedido y retorna una instancia de `RemoteFile` que encapsula su tamaño y su contenido.
3. El cliente upcastea el resultado a `IfaceRemoteFile` y verifica que represente un archivo existente. De no ser así, significaría que el servidor no tenía el archivo solicitado, y el programa finalizaría.
4. El cliente escribe localmente una copia del archivo retornado.
5. El cliente llama al `escribir()` del servidor, enviando su copia del archivo.
6. El servidor escribe localmente una copia de los datos recibidos.

Para comprobar esta funcionalidad, se puede correr el archivo `run.sh` desde el directorio `3`.

4. **Investigue si es posible que varias invocaciones remotas estén ejecutándose concurrentemente y si esto es apropiado o no para el servidor de archivos del ejercicio anterior. En caso de que no sea apropiado, analice si es posible proveer una solución (enunciar/describir una solución, no es necesario implementarla). Nota: diseñe un experimento con el que se**

pueda demostrar fehacientemente que dos o más invocaciones remotas se ejecutan concurrentemente o no.

En RMI es posible ejecutar varias invocaciones remotas concurrentemente. Esto se debe a que Java RMI se encarga automáticamente de desplegar los threads requeridos para dotar de concurrencia a un servicio implementado usando esta tecnología. Aunque esta característica es beneficiosa, el programador debe de ser consciente de que los métodos remotos en el servidor se ejecutan de manera concurrente, debiendo establecer mecanismos de sincronización en caso de que sea necesario.

En el caso de un servidor de archivos, la concurrencia no resulta apropiada ni beneficiosa. Dos clientes podrían intentar escribir el mismo archivo por medio de varios mensajes (por ejemplo para archivos muy grandes que superen el tamaño máximo de paquete). El resultado sería un archivo inconsistente por culpa de la mezcla de información.

Para demostrar la ejecución concurrente de RMI decidimos ejecutar varios clientes que intentan escribir distintos archivos en un mismo destino simultáneamente. Esta escritura se realiza en varias tandas Algunos clientes envían un archivo “a.txt” que contiene un string con la letra “a” repetida varias. Otros hacen lo propio con los archivos “b.txt” y “c.txt”.

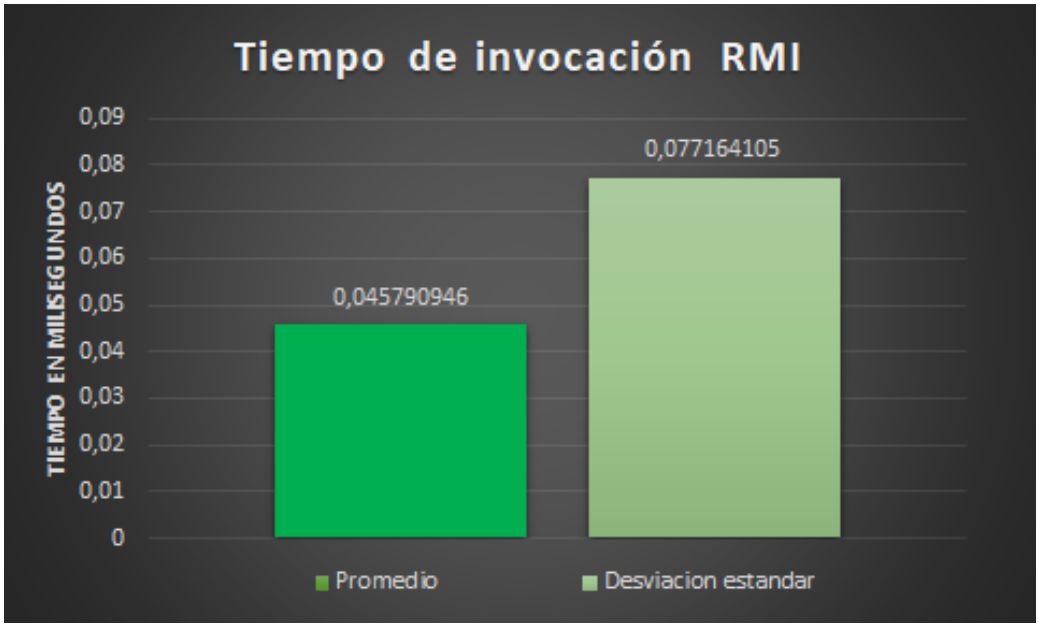
Luego de ejecutarlo, pudimos ver que archivo “result.txt” en el servidor posee los datos mezclados enviados por los clientes mezclados, intercalando a_s, b_s y c_s.

Una solución a este problema podría ser haciendo que cada cliente pueda escribir en un directorio distinto del lado del servidor. Así, dos clientes nunca podrían escribir el mismo archivo.

5. **Tiempos de respuesta de una invocación:**
    a. **Diseñe un experimento que muestre el tiempo de respuesta mínimo de una invocación con JAVA RMI. Muestre promedio y desviación estándar de tiempo respuesta.**

        Para este experimento utilizamos un cliente que realiza 100000 invocaciones remotas a un método vacío para poder verificar el tiempo mínimo de la invocación. Los resultados fueron los siguientes:

        - **Tiempo minino de respuesta:** 0,019273 milisegundos
        - El siguiente gráfico muestra el tiempo promedio de respuesta y la desviación estándar.



b. **Investigue los timeouts relacionados con RMI. Como mínimo, verifique si existe un timeout predefinido. Si existe, indique de cuanto es el tiempo y si podría cambiarlo. Si no existe, proponga alguna forma de evitar que el cliente quede esperando indefinidamente.**

   Investigando acerca de los timeouts en Java RMI, encontramos que, si bien no existen timeouts predefinidos dentro de `java.rmi`, sí existen diversos tipos definidos en el paquete `sun.rmi`. Estas son algunas de las propiedades más interesantes en Java 1.1 (y todas pueden ser modificadas por el usuario):

   - `sun.rmi.transport.tcp.handshakeTimeout`: permite configurar el tiempo que toma el Java RMI en decidir si una conexión TCP aceptada por el servidor remoto es utilizable, ya sea porque la entidad que escucha en el puerto host no es un Java RMI server o porque no está funcionando correctamente. El tiempo máximo de espera es `Integer.MAX_VALUE`, cero representa un tiempo infinito y el valor por defecto es 1 minuto.

     Esto es lo que sucede cuando se ejecuta el cliente con un `handshakeTimeout` muy bajo:

```
   4 ; ./run.sh
java.rmi.ConnectIOException: error during JRMP connection establishment; nested exception is:
        java.net.SocketTimeoutException: Read timed out
        at java.rmi/sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:300)
        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.lookup(RegistryImpl_Stub.java:116)
        at java.rmi/java.rmi.Naming.lookup(Naming.java:101)
        at client.AskRemote.main(AskRemote.java:31)
Caused by: java.net.SocketTimeoutException: Read timed out
        at java.base/sun.nio.ch.NioSocketImpl.timedRead(NioSocketImpl.java:283)
        at java.base/sun.nio.ch.NioSocketImpl.implRead(NioSocketImpl.java:309)
        at java.base/sun.nio.ch.NioSocketImpl.read(NioSocketImpl.java:350)
        at java.base/sun.nio.ch.NioSocketImpl$1.read(NioSocketImpl.java:803)
        at java.base/java.net.Socket$SocketInputStream.read(Socket.java:976)
        at java.base/java.io.BufferedInputStream.fill(BufferedInputStream.java:244)
        at java.base/java.io.BufferedInputStream.read(BufferedInputStream.java:263)
        at java.base/java.io.DataInputStream.readByte(DataInputStream.java:271)
        at java.rmi/sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:239)
        ... 5 more
```

- `sun.rmi.transport.tcp.responseTimeout`:se utiliza para determinar cuanto esperará cada cliente por la respuesta a la invocación de un método del servidor remoto antes de lanzar una excepción. El tiempo máximo de espera es `Integer.MAX_VALUE`, y el cero representa un tiempo infinito, es decir, sin timeout. Este último es el valor por defecto.

  Esto es lo que sucede cuando se ejecuta el cliente con un `responseTimeout` muy bajo:

```
   4 ; ./run.sh
java.rmi.UnmarshalException: Error unmarshaling return header; nested exception is:
        java.net.SocketTimeoutException: Read timed out
        at java.rmi/sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:254)
        at java.rmi/sun.rmi.server.UnicastRef.invoke(UnicastRef.java:380)
        at java.rmi/sun.rmi.registry.RegistryImpl_Stub.lookup(RegistryImpl_Stub.java:123)
        at java.rmi/java.rmi.Naming.lookup(Naming.java:101)
        at client.AskRemote.main(AskRemote.java:31)
Caused by: java.net.SocketTimeoutException: Read timed out
        at java.base/sun.nio.ch.NioSocketImpl.timedRead(NioSocketImpl.java:283)
        at java.base/sun.nio.ch.NioSocketImpl.implRead(NioSocketImpl.java:309)
        at java.base/sun.nio.ch.NioSocketImpl.read(NioSocketImpl.java:350)
        at java.base/sun.nio.ch.NioSocketImpl$1.read(NioSocketImpl.java:803)
        at java.base/java.net.Socket$SocketInputStream.read(Socket.java:976)
        at java.base/java.io.BufferedInputStream.fill(BufferedInputStream.java:244)
        at java.base/java.io.BufferedInputStream.read(BufferedInputStream.java:263)
        at java.base/java.io.DataInputStream.readByte(DataInputStream.java:271)
        at java.rmi/sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:240)
        ... 4 more
```