Programación Distribuida y Tiempo Real Facultad de Informática - Universidad Nacional de La Plata Practica 2

Practica 2

Integrantes: Ana Mariela Cossio Aquino, Leandro David Svetlich

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.

Se puede decir que RMI no es de acceso completamente transparente, dado que el cliente, en este caso **AskRemote**, no es totalmente ajeno al hecho de que está trabajando con un objeto remoto. Esto se puede apreciar en la interfaz IfaceRemoteClass ya que el método sendThisBack arroja la excepción RemoteException. Por lo tanto, para utilizar este método el cliente debe asegurarse de capturar esta excepción. Si RMI fuera de acceso completamente transparente, AskRemote trataría al objeto remoto de igual manera que a un objeto local, cosa que no ocurre en este caso.

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

Servidor

- IfaceRemoteClass
- RemoteClass
- StartRemoteObject

Cliente:

- IfaceRemoteClass
- AskRemote

‣ *StartRemoteObject* es el punto de entrada para iniciar el objeto remoto y hacer que este pueda ser invocado desde otras JVMs.
‣ *IfaceRemoteClass* es la interfaz del objeto remoto, el cliente debe conocer esta interfaz para poder comunicarse con el objeto remoto.
‣ *RemoteClass* es la implementación del objeto remoto, allí se define como van a comportarse los métodos que pueden ser invocados de manera remota.
‣ *AskRemote* es quien hace de cliente y se encarga de invocar remotamente a un método de RemoteClass a través de su interfaz.

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

RMI permite descargar la definición de una clase en caso de no estar definida en la JVM del receptor. Los objetos son transmitidos mediante su clase, por lo tanto su comportamiento se

mantendrá al ser enviado a otra JVM. Esto permite introducir nuevos tipos y comportamientos de manera dinámica a JVMs remotas.
RMI trata a los objetos remotos de manera diferente. Cuando se pasan de una JVM a otra, en vez de hacer una copia de la implementación, RMI pasa un objeto “sustituto” que actúa como un proxy que representa al objeto remoto. Luego el cliente puede invocar métodos de este objeto sustituto, el cual en realidad se encarga de transmitir la invocación de dicho método al objeto remoto en el servidor.

Ya que al utilizar este mecanismo se va a ejecutar código del cual puede que se tenga completo conocimiento, es importante aplicar mecanismos de seguridad que protejan al sistema de código descargado que corra sobre la JVM. Para eso los programas RMI deben instalar algún Security Manager, sin él, RMI no descargará clases de objetos recibidos como parámetro o valores de retorno de invocaciones a métodos remotos. Estas restricciones aseguran que las operaciones llevadas a cabo por código que ha sido descargado bajo demanda estén sujetas a una política de seguridad.

3) Implementar con RMI un sistema de archivos remoto:
a.- Defina e implemente con RMI un servidor cuyo funcionamiento permita llevar a cabo las operaciones desde un cliente enunciadas informalmente como (definiciones copiadas aquí de la práctica anterior):

> **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.**

Para implementar el servidor, se tomó como base el código provisto por la cátedra, comenzando por modificar la interfaz IfaceRemoteClass.

```
public interface IfaceRemoteClass extends Remote {
  public IResponse leer(String nombre, int posicion, int cantidad) throws RemoteException;
  public int escribir( String nombre, int cantidad, byte[] buffer ) throws RemoteException;
}
```

IResponse es la interfaz de un objeto serializable que se usa simplemente para almacenar la cantidad de bytes leídos y los bytes en sí mismos, y poder retornarlos al ejecutar la invocación remota.
Las implementaciones de estos métodos se encuentran en la clase RemoteClass, y funcionan exactamente como se describen en el enunciado.

leer

```
InputStream input = new FileInputStream(file);
int leido = 0; // cantidad leida en un read
byte[] buffer = new byte[bufferSize];
input.skip(posicion);
leido = input.read(buffer, 0, cantidad);
respuesta.setCantidad(leido);
respuesta.setDatos(buffer);
```

escribir

```
OutputStream output = new FileOutputStream(file, true);// true para que
no se pisen los datos
        output.write(buffer, 0, cantidad);
        output.close();
```

El código para arrancar el objeto remoto se encuentra en la clase Servidor, que renombramos a partir de la clase StartRemoteObject.

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 esta parte, se definió una clase Cliente que implementa los métodos leer y escribir, pero vistos desde el lado del cliente.
Se eligió un tamaño de buffer de 8192 bytes, es decir que en cada invocación a un método remoto, se tratará de leer/escribir esa cantidad de bytes. Por lo tanto, para transferir un archivo, ya sea leer o escribir, se deberán hacer sucesivas invocaciones a los métodos remotos.
Para leer un archivo del servidor, comienza por invocar al método remoto leer del servidor, indicando el nombre del archivo, el desplazamiento que inicialmente es cero, y la cantidad de bytes a leer que como se mencionó previamente es 8192.

```
response = remote.leer(name, 0, bufferSize);
```

Si la cantidad que efectivamente se leyó es mayor a -1, por única vez se elimina dicho archivo del almacenamiento local del cliente y se vuelve a crear. Mediante FileInputStream se escriben los bytes leídos en el archivo. Luego, se actualiza el desplazamiento para la próxima iteración, y esto se repite hasta que en algún momento la invocación remota a leer retorne -1, ya sea porque no hay más bytes que leer, o porque ocurrió algún error.

```
while (response.getCantidad() > -1) {
    response = remote.leer(name, (int) offset, bufferSize);
    if (response.getCantidad() > -1) {
        output.write(response.getDatos(), 0, response.getCantidad());
        offset = offset + response.getCantidad();
    }
}
```

Para la escritura de un archivo en el servidor se procede de manera bastante similar.
La principal dificultad de esta operación, es que en la operación escribir del servidor no se indica el desplazamiento, sino que por defecto los bytes recibidos se agregan al final del archivo. Por este motivo fue necesario implementar una operación adicional en la clase RemoteClass; el método crearArchivo indica que el archivo con ese nombre debe crearse en caso de que no exista, y re-crearse si es que ya existe. De esta manera, si un cliente quiere sobreescribir un archivo en el servidor, tiene posibilidad de hacerlo sin que los bytes se agreguen al final del archivo viejo.

```
public interface IfaceRemoteClass extends Remote {
  public IResponse leer(String nombre, int posicion, int cantidad) throws RemoteException;
  public int escribir( String nombre, int cantidad, byte[] buffer ) throws RemoteException;
  public boolean crearArchivo(String nombre) throws RemoteException;
}
```

Por lo tanto en la operación de escritura del lado del cliente, se comienza por invocar este nuevo método remoto.
Agregamos el parámetro asCopy para saber si se desea guardar el archivo con otro nombre en el servidor; por ejemplo si el nombre original era tp2.pdf, el nombre de la copia será tp2 copy.pdf.

```
private static boolean escribirCliente(String name, boolean asCopy);
```

Luego se comienza a leer el archivo local mediante FileInputStream.read(). Mientras el read retorne algo mayor a -1, se continúa iterando e invocando al método remoto escribir, indicando el nombre del archivo, la cantidad de bytes a escribir y los bytes en sí mismos.

```
int read = 0;
byte[] buffer = new byte[bufferSize];
while (read > -1) {
    read = inputStream.read(buffer, 0, bufferSize);
    if (read > 0) // si no hay error ni se termino el archivo
        remote.escribir(nameInServer, read, buffer);
}
```

Para realizar la secuencia de transferencias descrita en el enunciado, simplemente llama a su método leer, indicando el nombre del archivo deseado, y una vez terminado, llama a su método escribir, indicando que se desea renombrar como copia.

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.

Es posible hacer muchas invocaciones a métodos remotos y que se ejecuten concurrentemente, ya que cada invocación se ejecuta en un hilo diferente. RMI no especifica un límite para la cantidad de hilos, sino que lo deja abierto a cada implementación.
Para demostrar que efectivamente esto ocurre de esta manera, se realizó un experimento que consiste en hacer simultáneas invocaciones a un método remoto que simplemente loopea indefinidamente, y lo que hace es dormirse durante cinco segundos, y luego imprime un identificador del cliente que realizó la invocación.

Allí puede verse claramente que múltiples invocaciones remotas sobre el mismo método están siendo ejecutadas de manera concurrente, y no es necesario esperar a que las otras invocaciones terminen para ejecutarse.

```
Servidor -> escuchando...
Servidor -> Atendiendo a : Cliente 3
Servidor -> Atendiendo a : Cliente 2
Servidor -> Atendiendo a : Cliente 1
Servidor -> Atendiendo a : Cliente 3
Servidor -> Atendiendo a : Cliente 2
Servidor -> Atendiendo a : Cliente 1
Servidor -> Atendiendo a : Cliente 3
Servidor -> Atendiendo a : Cliente 2
Servidor -> Atendiendo a : Cliente 1
Servidor -> Atendiendo a : Cliente 3
Servidor -> Atendiendo a : Cliente 2
Servidor -> Atendiendo a : Cliente 1
```

Cabe aclarar que esto puede variar dependiendo de la política de scheduling del sistema operativo.

En el servidor de archivos implementado en el ejercicio anterior, la ejecución simultánea de métodos que operan sobre los mismos archivos, ya sea leyendo o escribiendo puede ocasionar errores de sobreescritura y pérdida de información. Esto se debe a dos motivos; El primero es que la operación de escritura del FileOutputStream no es atómica, por lo que puede haber varios hilos escribiendo o leyendo el archivo al mismo tiempo. Lo mismo sucede con el read del FileInputStream. El otro motivo es que para guardar o leer un archivo en el servidor, el cliente debe hacer reiterados llamados los métodos remoto escribir y leer, entonces, aunque las operaciones write/read del FileOutputStream/FileInputStream fueran atómicas, todavía habría problemas de sincronización que resolver.
Una posible solución al primer problema sería utilizar File locks para garantizar la exclusión mutua en la escritura/lectura de los archivos.

Para el segundo problema, podrían implementarse métodos remotos que permitan que los clientes indiquen que se va a comenzar/terminar de escribir un archivo del servidor. El servidor debería tomar un identificador del cliente, y solo permitir operar sobre el archivo a este cliente, hasta que este indique que finalizó de operar sobre el archivo. En cuanto a la lectura de un archivo del servidor, tanto el comienzo como la finalización puede inferirse de la operación leer sin necesidad de que el cliente lo indique explícitamente. El comienzo se infiere cuando la posición para empezar a leer es cero, y la finalización cuando el valor de retorno de la operación read del FileInputStream es -1. El servidor deberá operar con exclusión mutua para ver o para establecer qué cliente está trabajando con cada 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 se implementó un método remoto que simplemente retorna sin hacer nada. A este método lo llamamos *metodoInutil.* El cliente simplemente hace sucesivos llamados a este método remoto, tomando un timestamp antes de invocar al método, y otro tras su retorno. Para cada invocación al método se calcula la diferencia entre los timeouts y almacena este valor en un arreglo, para luego ser procesado y calcular tanto el promedio como la desviación estándar.

Resultados obtenidos:

- Tamaño de muestra: **10000**
- Total: 473.0ms
- **Promedio: 0.0473ms**
- **Desviación estándar: 0.21227979178431725ms**

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.

Si bien la API pública de Java RMI no especifica ningún timeout, sus implementaciones suelen definir estas propiedades y permitir su configuración.
Por ejemplo, la implementación de Sun Microsystems sun.rmi soporta, entre otras, las siguientes propiedades con respecto a timeouts:

- **sun.rmi.transport.tcp.readTimeout:** timeout en milisegundos a utilizar para nuevas conexiones TCP. Este valor se pasa a la capa de Socket, y se usa en casos en los que el cliente no cierra una conexión que no está siendo utilizada. Su valor por defecto es de dos horas.

- **sun.rmi.transport.tcp.responseTimeout:** representa el valor que el cliente RMI va a usar como timeout para lectura del socket, en una conexión JRMP (protocolo de transporte utilizado por RMI). Esta propiedad se puede usar para establecer un límite en la espera del retorno de la invocación al método remoto. Su valor por defecto es cero, lo cual indica que no se requiere timeout.

- **sun.rmi.transport.tcp.threadKeepAliveTime**: tiempo que un hilo usado para esperar conexiones permanecerá ocioso antes de ser dado de baja. El valor por defecto es de un minuto.

- **sun.rmi.transport.connectionTimeout**: tiempo en el que un socket puede permanecer ocioso antes de que se cierre la conexión. El valor por defecto es de 15 segundos.

Estas propiedades se establecen mediante *System.setProperty()*.

Si se está utilizando solo la API pública de RMI estas propiedades quedan sujetas a los mecanismos de comunicación subyacentes.

Una manera de evitar que un cliente quede esperando indefinidamente podría ser haciendo la invocación al método remoto en un nuevo hilo mediante un Executor, y estableciendo un timeout para dicho hilo.