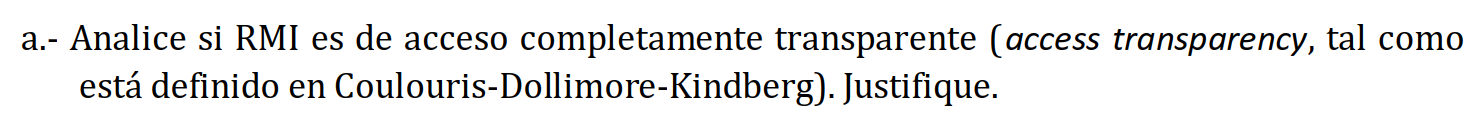
**TRABAJO PRACTICO 2**

Vamos a mostrar los ejercicios 4 y 5

## 

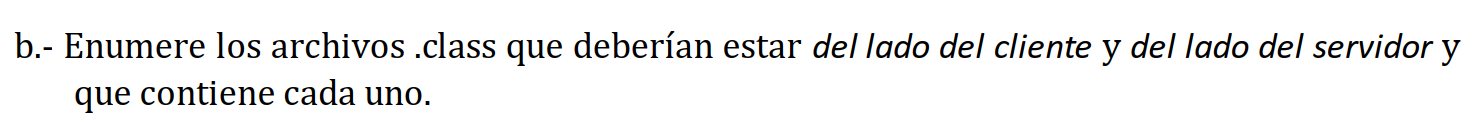
## 1)



### a-

RMI es transparente en cuanto a la invocación de métodos de objetos remotos, tanto la invocación remota como la invocación local se realiza de la misma manera. La diferencia entre un objeto remoto y uno local está en la interfaz, los objetos remotos RMI implementan la interfaz remote y pueden lanzar RemoteExceptions.

En el código de ejemplo que tenemos de RMI, la clase RemoteClass implementa la interfaz IfaceRemoteClass, que hereda de la interfaz Remote, y sus métodos pueden lanzar la excepción RemoteException.



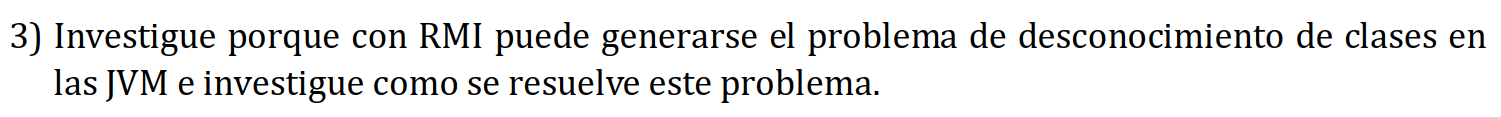
### b-

La interfaz ​IfaceRemoteClass​ tiene los métodos remotos que serán definidos por cliente y servidor.  
La clase ​RemoteClass ​es la que va a implementar los métodos remotos del lado del  
servidor definidos en la interfaz.  
La clase​ StartRemoteObject ​crea y registra los objetos remotos, instancia un objeto de la clase Remote Class y lo asocia con un nombre/puerto.  
La clase ​AskRemote busca el objeto remoto y utiliza sus servicios por medio de las operaciones definidas en la interfaz (que actúa como cliente).  
En resumen:

En el servidor están los archivos .class de las clases ​IfaceRemoteClass​, RemoteClass y StartRemoteObject.

En el cliente están los archivos .class de las clases ​IfaceRemoteClass ​y ​AskRemote.

## 2) No se hace…

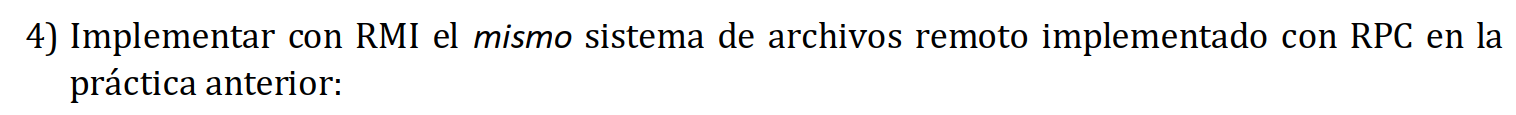


## 3)

El problema que puede generarse es que al ser esquemas distribuidos, puede que el server esté en un lugar diferente al cliente.

Si en la interfaz definiéramos una operación que retornara un objeto de una clase, la cual solo conoce el server, lo que pasará cuando compilemos el cliente es que lanzará una excepción ​ClassNotFoundExceptionporque no “conoce” la clase que retorna el método de la interfaz.  
Si localmente el cliente y el servidor están en el mismo lugar, RMI lo resuelve por nosotros, indicando el classpath al ejecutar, en caso de que no lo indique, busca las clases en el directorio actual.

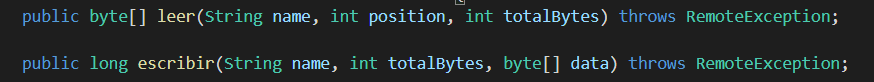
Pero en caso de que se encuentren en ​**espacios**​ ​**remotos**​ hay que seguir los siguientes  
pasos según la documentación oficial:  
**1.** Toda clase que se quiera enviar debe implementar la interfaz ​*Serializable*​. La cual le permitirá al compilador de Java determinar que el objeto se debe serializar para enviar a través de la red. Esto es necesario tanto para espacios locales como remotos.  
**2.** Se debe definir una interfaz común a ambos y toda clase que se desea enviar a través de RMI deberá implementar esta interfaz, esto permitirá que el compilador no arroje errores en tiempo de compilación ya que ambos conocerán la interfaz y luego harán un upcasting del objeto recibido.  
**3.** Para evitar vulnerabilidades de seguridad se debe especificar un objeto llamado *RMISecurityManager* en ambos lados de la conexión. En caso de no especificarlo Java no permitirá descargar clases desconocidas.  
**4.** Se debe comprimir en un .​*jar*​ la interfaz común a ambos y luego ejecutar tanto el Server como el Client con el parámetro: -Djava.rmi.server.codebase=<location>  
En donde location representará la ubicación del archivo ​*.jar* ​en dónde se guardarán las clases común a ambos, se puede implementar a través de http o ftp del lado del servidor. Esto es lo que le permitirá al Cliente poder cargar la interfaz para queimplemente el objeto que se quiere enviar como parámetro.



## 4)

### a-

Primero se definieron los métodos leer y escribir en la interfaz IfaceRemoteClass con los requisitos pedidos:

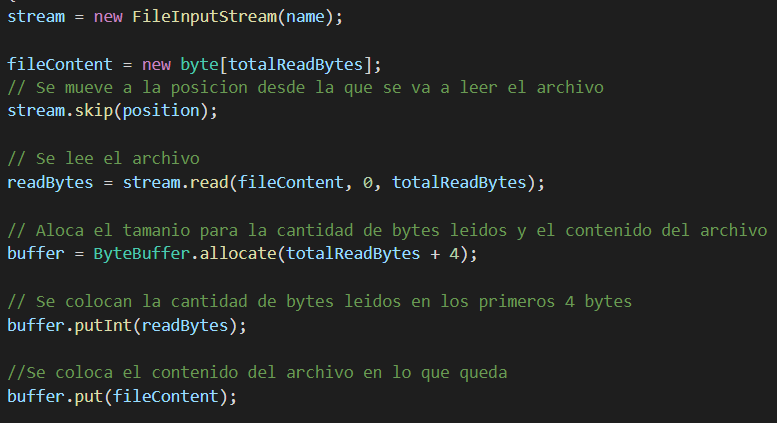


En el caso del leer se retornara un arreglo que contiene en sus primeros 4 bytes el tamaño del archivo y el resto sería el contenido

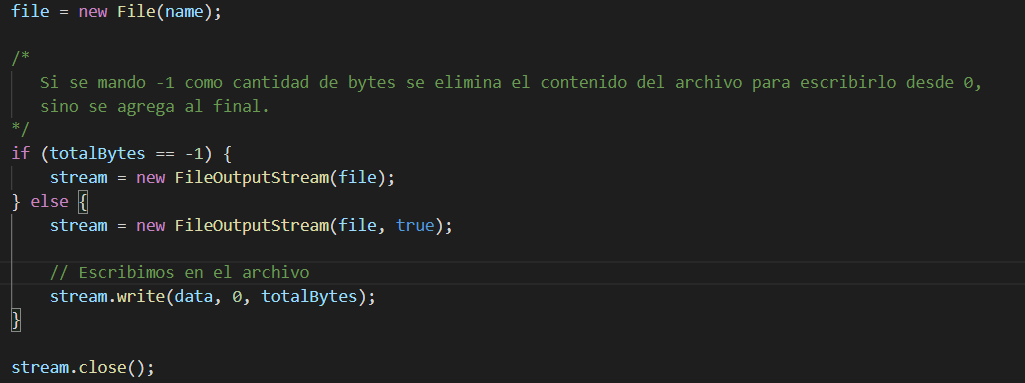
Después en la implementación de ambos métodos dentro de RemoteClass se hizo lo siguiente:

En ambos se utilizó un try catch para atrapar excepciones en caso de que suceda algún tipo de error durante la lectura y apertura de un archivo, como puede ser no tener el archivo solicitado con cierto nombre.

En el caso del leer, utilice “FileInputStream” para obtener los bytes del archivo y poder recorrer el archivo desde la posición indicada con el .skip, luego se lee el archivo con el stream implementado, el cuál nos retorna el tamaño del archivo en los primeros 4 bytes y el contenido en los demás. Se guarda eso en “bytesRead” para tener el tamaño del archivo, ahora alocamos el total de bytes leidos + 4 (tamaño del archivo), y por último se almacena en el buffer dentro de los 4 primeros bytes el tamaño del archivo y en lo que queda el contenido. Al final se retorna el buffer bytes.



En el caso del escribir si se recibe -1 en la cantidad de bytes se creara el archivo eliminando el que había con la clase “FileOutputStream”, en caso contrario se escribe al final del archivo utilizando la función write de la misma, y se devuelven los bytes escritos.



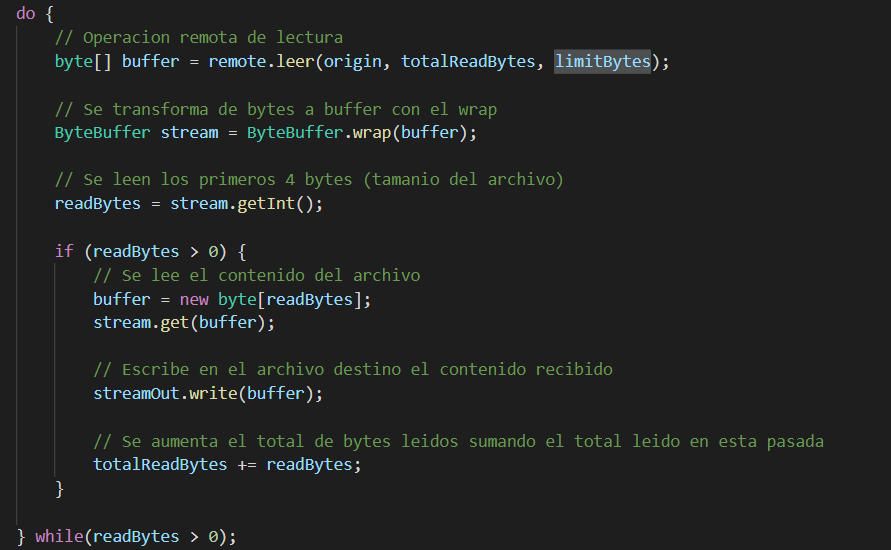
## 

### b-

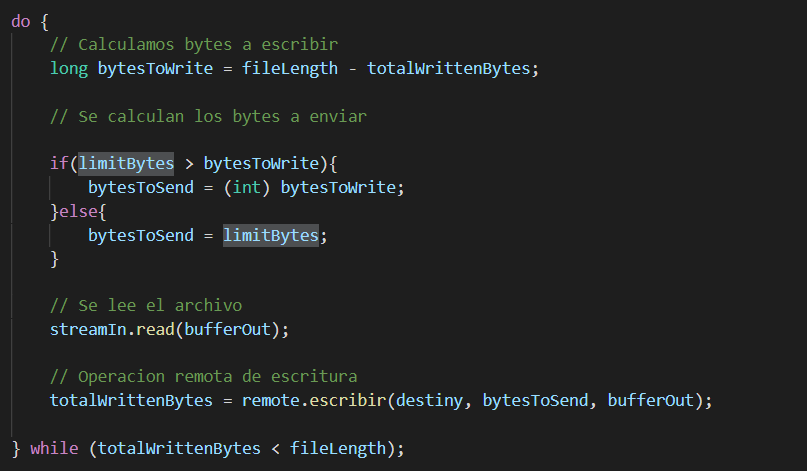
Para implementar esto se recibiran por parametros los strings del archivo original, la copia del cliente y el servidor.

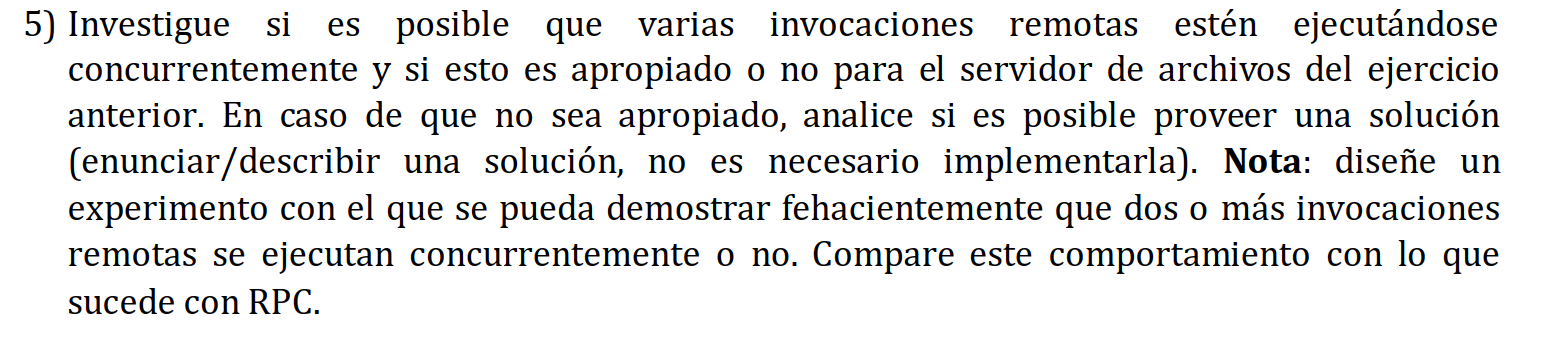
Se creó el método leer en el AskRemote.java para poder hacer la copia del archivo original desde el servidor al cliente, como también se creó el método escribir para poder hacer la copia desde la copia cliente al copia del servidor.

El método leer guarda en un array de bytes lo que se lee con el “leer” creado en el punto anterior de la RemoteClass (por medio de la interface). Transforma eso a buffer con el wrap de ByteBuffer, se leen los 4 primeros bytes para obtener el tamaño del archivo y ahora se procede a hacer la lectura y envio completo del archivo en segmentos, algo muy parecido a lo realizado en el tp1, donde hasta no enviar el archivo completo no termina el loop. (Para la lectura del buffer se utilizó FileOutputStream como en el punto anterior).



El método escribir primero eliminar algún archivo con el mismo nombre y crea uno nuevo, luego mientras el total de bytes escritos sea menor a el tamaño del archivo a crear se procede. El tamaño de bytes leídos se obtiene del método escribir creado en el punto anterior. A la hora de escribir hay un límite de bytes por segmento, entonces si este límite es mayor a lo que escribiremos se toma el total de bytes que tenemos para escribir, en caso contrario se escribirá la misma cantidad de bytes que la del límite, y así hasta que se complete la escritura del mismo.

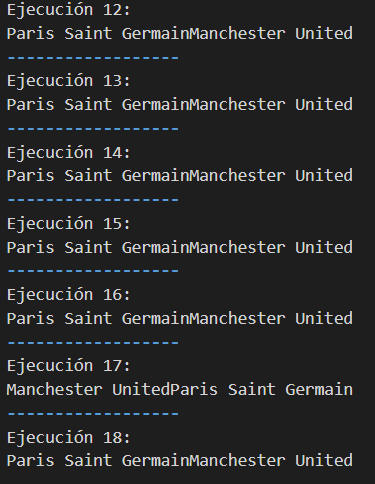
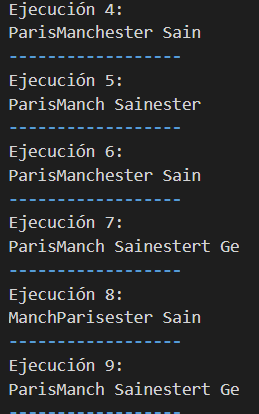




## 5)

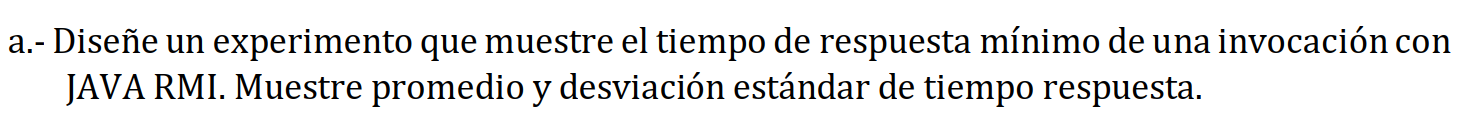
Sí, es posible que varias invocaciones remotas se estén ejecutando concurrentemente, pero  
no es apropiado para el servidor, debido a que puede haber interferencia entre las múltiples  
comunicaciones simultáneas que tenga el servidor con los clientes. Esta interferencia puede  
provocar que por ejemplo un archivo del servidor al ser escrito por varios clientes  
concurrentemente, tenga información errónea (es decir que los bytes enviados por los  
clientes se mezclen).

En mi caso en el experimento se ejecutó con bytes que no alcanzaran a abarcar ambos txt, lo cual entiendo que “simulo” una situación similar a tener un límite como el planteado en el punto anterior, es decir, que se envié en segmentos el mensaje porque no entraba todo y lo que sucedió es que se mezclaron ambos mensajes. En el caso de introducir bytes de sobra solo se cambiaba el orden de impresión de los mensajes pero no se mezclaban.

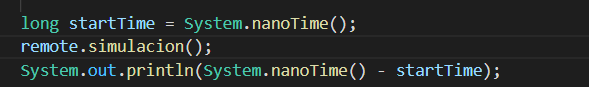
  
Como nuestra implementación escribe el archivo en pequeñas porciones de una ventana de  
bytes no alcanza con que sea atómica sólo la operación de lectura o escritura, ya que  
puede haber interferencias incluso si son atómicas. Por lo tanto, debe ser atómica la sesión  
del cliente hasta que se termine de leer/escribir el archivo deseado.  
Entonces llegamos a que la ​**solución**​ es crear una sesión de lectura o escritura para cada  
cliente y hacer que dicha sesión sea atómica

## 

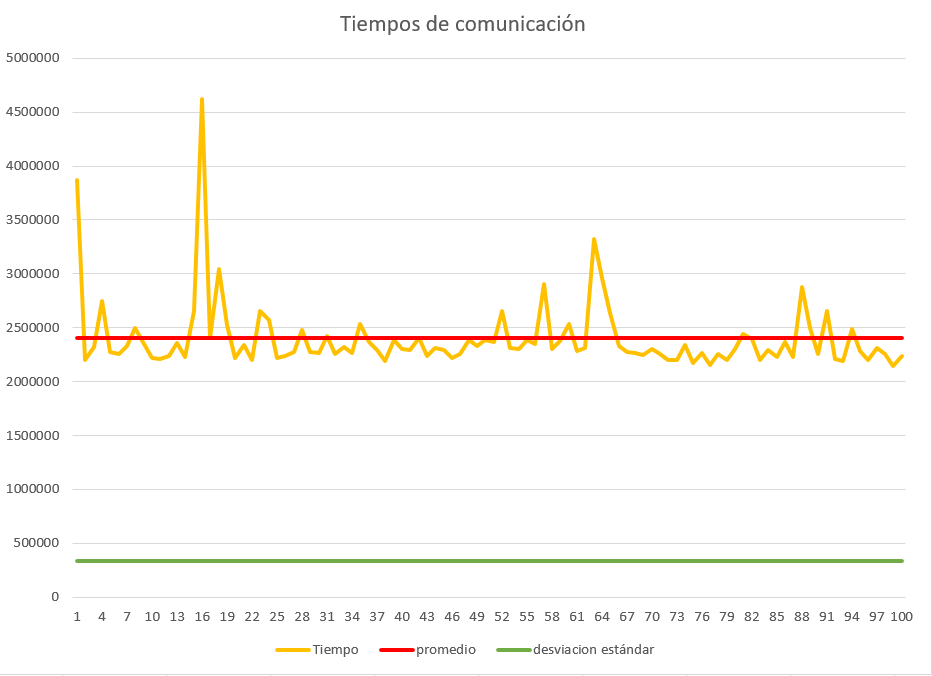
## 6)

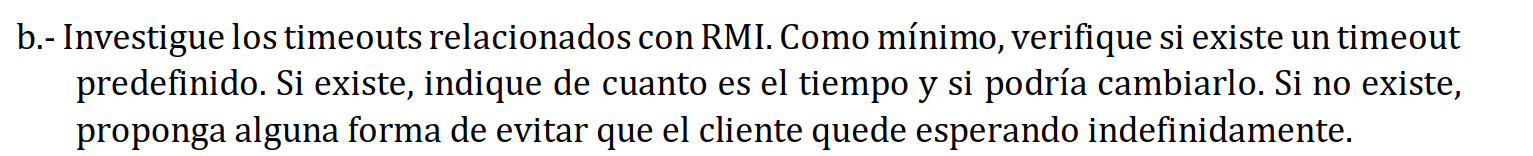


### a-

Se declaró en la interfaz e implemento en el servidor un método “simulación” que no hace nada, solo sirve para poder evitar el tiempo de procesamiento. Por último se creo en el cliente lo necesario para obtener el tiempo de respuesta de cada invocación de la misma forma que en el tp anterior, solo que utilizamos nanosegundos esta vez

Se realizó un script para poder obtener los tiempos 100 veces y crear este grafico.



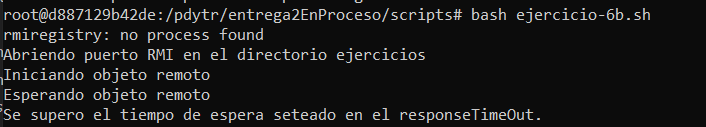


### b-

No existe un timeout predefinido pero existe el paquete de RMI llamado ​sun.rmi, que nos permite setear tiempos con la propiedad sun.rmi.transport.tcp.responseTimeout.

​Esta propiedad nos da la posibilidad de configurar el tiempo de espera de respuesta de un objeto remoto que tiene el cliente. Si paso más tiempo que el seteado en la espera entonces se  
lanza una excepción de la clase ​RemoteException.

Para simular lo pedido se implementó un método que realiza un loop infinito, así el cliente quedaría esperando indefinidamente, y ahora para evitar que esto suceda se cambió el valor del responseTimeOut, así corta después de “x” segundos en espera.



Bibliografia

<https://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html>

<https://docs.oracle.com/javase/7/docs/api/java/io/FileOutputStream.html>

<https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html>

<https://stackoverflow.com/questions/1822695/java-rmi-client-timeout>

<https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/sunrmiproperties.html>