Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Programación distribuida y tiempo real

Entrega 2

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Ejercicio 1

Inciso A

El **acceso transparente** permite que los recursos locales y los remotos sean accesibles usando operadores idénticos [Coulouris-Dollimore-Kindberg].
Esto se da ***parcialmente*** en los ejemplos provistos ya que tanto el cliente como el servidor acceden a los recursos mediante las mismas operaciones definidas en la interfaz **IfaceRemoteClass** y el llamado a los métodos de la clase remota se hace igual que los llamados a un método de una clase local.
Decimos ***parcialmente*** porque hay que tener en cuenta que para acceder a los recursos remotos tenemos que previamente tener acceso a la clase remota (es decir tener una instancia de la clase remota), y cumplir con cada una de las condiciones que nos exija (como por ejemplo capturar la excepción RemoteException en caso de que ocurra un error).

Inciso B

La interfaz **IfaceRemoteClass** contiene los métodos remotos que luego deben ser definidos tanto por el cliente como por el servidor**.**

La clase **RemoteClass** es la que va a implementar los métodos remotos del lado del servidor que fueron definidos en la interfaz. En otras palabras, será la clase en la cual se responderá a los servicios definidos por la interfaz.

La clase **StartRemoteObject** crea y registra los objetos remotos. Es decir, en simples palabras se encarga de instanciar un objeto de la clase Remote Class y lo asocia con un nombre/puerto.

La clase **AskRemote** se encarga de buscar el objeto remoto y usar sus servicios a través de las operaciones definidas en la interfaz, la misma actuará como Cliente.

Del lado del servidor deberían estar los archivos *.class* de las clases **IfaceRemoteClass**, **RemoteClass y StartRemoteObject.** Mientras que por el lado cliente deberían estar los archivos *.class* de las clases **IfaceRemoteClass** y **AskRemote.**

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Ejercicio 2

El problema que surge con *RMI* y el *desconocimiento de clases* es que al ser esquemas distribuidos, el server puede estar en un lugar totalmente diferente al cliente. Si por ejemplo definimos una operación en la interfaz que retorna un objeto de una clase que solo conoce el Server y compilamos el Cliente, Java lanzará una excepción **`ClassNotFoundException`** en tiempo de compilación porque no tiene el código de la clase que retorna el método de la interfaz.

En caso de que el cliente y el servidor se encuentren en el mismo lugar (***localmente***) RMI lo resolverá automáticamente por nosotros especificando el classpath al ejecutar. En caso de no especificarlo tomará como classpath el directorio actual para buscar las clases desconocidas.
Pero en caso de que se encuentren en **espacios remotos** hay que seguir los siguientes pasos según la documentación oficial:

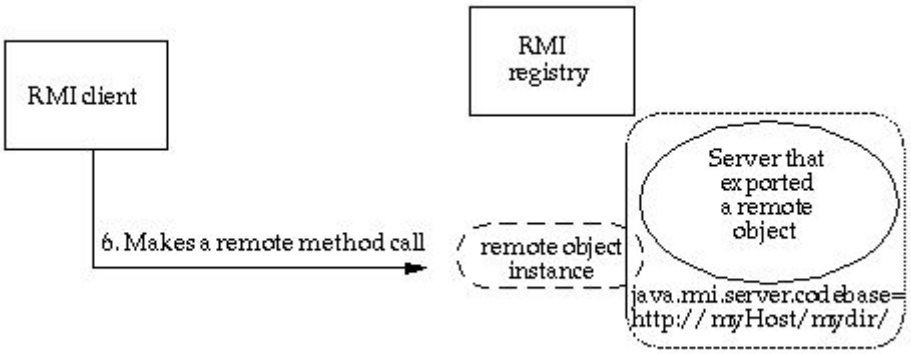
1. Toda clase que se quiera enviar debe implementar la interfaz *Serializable*. La cual le permitirá al compilador de Java determinar que el objeto se debe serializar para enviar a través de la red. Esto es necesario tanto para espacios locales como remotos.

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 que

implemente el objeto que se quiere enviar como parámetro.



Fuentes:
https://docs.oracle.com/javase/tutorial/rmi/overview.html
https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/codebase.html

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Ejercicio 3

La implementación de estos ejercicios se encuentran en el directorio *ejercicios/ftp/.*

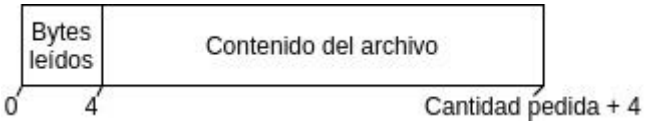
Inciso A

Para este inciso lo primero que hicimos fue definir las funciones **leer** y **escribir** dentro de la interfaz **IfaceRemoteClass.**

La operación **leer** recibe 3 argumentos:

- Un **String** que representa el **nombre del archivo** que se quiere leer del servidor.
- Un **Int** que representa desde qué **posición** del archivo se quiere empezar a leer.
- Un **Int** que representa la **cantidad de bytes** que se quiere leer del archivo.

Retorna un **arreglo de bytes** que representa el contenido leído del archivo. Además dentro de este arreglo, en los primeros 4 bytes, se retorna la cantidad de bytes del archivo que efectivamente fueron leídos:



El método **escribir** recibe 3 argumentos:

- Un **String** que representa el **nombre del archivo** que se quiere escribir en el servidor.
- Un **Int** que representa la **cantidad de bytes** que se quiere escribir en el archivo.
- Un **arreglo de bytes** que representa el contenido que se quiere escribir en el archivo.

Retorna un long int que representa la cantidad total de bytes que se escribieron en el archivo.

La implementación de ambas operaciones se encuentran dentro de la clase **RemoteClass**:
- Para la operación **leer** escribimos el código dentro de un bloque **try catch** con el fin de atrapar las excepciones que pudiesen ocurrir en el proceso de abrir y leer información de un archivo (por ejemplo que no exista en el servidor ningún archivo que coincida con el nombre solicitado).
Lo primero que hacemos es saltar hasta la posición del archivo desde donde se quiere empezar a leer el archivo. Luego, para leer el archivo usamos la función **read** de la clase **FileInputStream**, la cual retorna el contenido del archivo y la cantidad de bytes leídos. Esta información la almacenamos en un buffer de bytes:

1. En los primeros 4 bytes la cantidad de bytes leídos
2. En el resto del buffer el contenido del archivo.

Finalmente retornamos el buffer de bytes y finaliza el método.

- Para la operación **escribir** nuevamente escribimos el código dentro de un bloque **try catch** con el fin de atrapar las excepciones que pudiesen ocurrir.
Si se envió -1 como cantidad de bytes entonces se crea un nuevo archivo de forma destructiva (es decir, eliminando todo el contenido existente), en caso contrario se escribe al final del archivo. Para esto utilizamos la función **write** de la clase **FileOutputStream** sobre el archivo, datos y cantidad de bytes que recibimos**.**
Por último cerramos el archivo mediante la función **close** y retornamos la cantidad de bytes escritos hasta el momento (es decir el tamaño actual del archivo).

Inciso B

La implementación del inciso está en el archivo *ejercicios/ftp/AskRemote.java*. El mismo recibe 4 parámetros:

1. Ejercicio, en este caso *-ejercicio-3*.
2. El path del archivo original que reside en el servidor.
3. El path del archivo que será la copia del lado del cliente.
4. El path del archivo que será la copia del lado del servidor.

Por ejemplo:

```
java AskRemote.java -ejercicio3 archivo_original.jpg copia_cliente.jpg copia_sv.jpg
```

→ Para implementar la copia del archivo original hacia el lado del cliente utilizamos el método **leer**. Para ello vamos leyendo una ventana fija de bytes del archivo original hasta terminar de leer del archivo, por cada lectura realizamos lo siguiente:

Invocamos a la operación leer con los parámetros:

1. El nombre del archivo original.
2. La posición que va a estar dada por la cantidad de bytes leídos del archivo original.
3. La cantidad de bytes a leer será la ventana de 1024 bytes (*siempre fija*).

Los bytes retornados de la llamada son divididos en:

1. Los primeros 4 bytes para obtener la cantidad de bytes leídos por el servidor.
2. Los bytes restantes para obtener el contenido leído por el servidor.

En todos los casos los bytes leídos serán igual a la ventana, excepto en el último caso en donde puede que el último fragmento de bytes del archivo original sea menor a la ventana, por lo tanto para evitar escribir bytes “basura” escribimos la **cantidad de bytes leídos** (que obtuvimos en los primeros 4 bytes de la respuesta recibida) al final del archivo copia.

Los bytes leídos también servirán para determinar si terminamos de leer el archivo original, es decir, si los bytes leídos son menores o iguales a 0 entonces la lectura del archivo original terminó.

→ Luego para implementar la copia del archivo en el cliente hacia el lado del servidor utilizamos el método **escribir**. Utilizamos la misma técnica que en el anterior, vamos escribiendo la copia con una ventana de 1024 bytes. Excepto en el último fragmento ya que pueden faltar enviar una cantidad de bytes menor a la ventana, por lo tanto en el caso de que la cantidad de bytes faltantes sean menores a la ventana se enviará la cantidad de bytes faltantes. En caso contrario se envía la ventana.

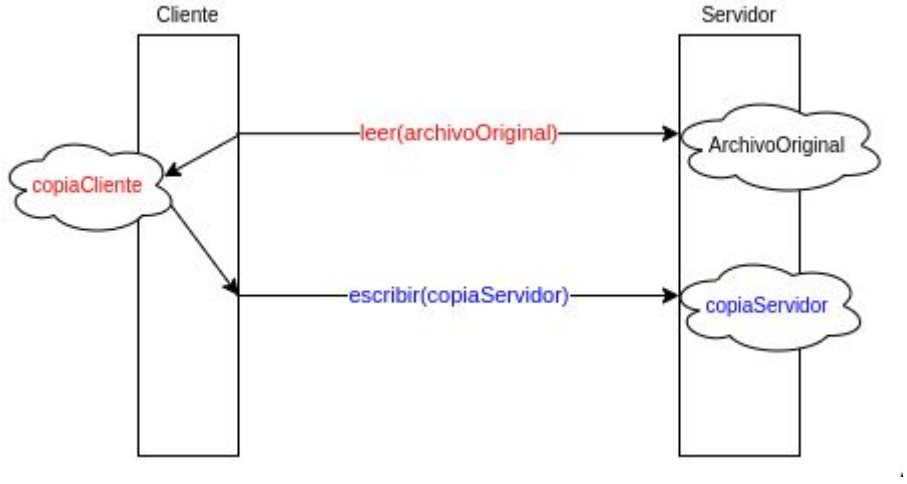
En el primer llamado del método **escribir** el cliente envía como cantidad de bytes **-1** para hacer una escritura destructiva, es decir, eliminará todo el contenido existente del archivo. Los siguientes llamados al método escribir se comportan de la siguiente manera:

A medida que vamos leyendo la copia del cliente invocamos al método **escribir** con los parámetros:

1. El nombre de la copia del servidor.
2. El contenido del archivo copia del cliente con una ventana de 1024 bytes.
3. La cantidad de bytes a enviar:
    a. Si la cantidad de bytes faltantes es menor a la ventana entonces se envía la cantidad de bytes faltantes.
    b. Caso contrario se envía la ventana.

Para terminar la escritura aprovechamos que la llamada remota **escribir** nos retorna la cantidad de bytes del archivo escrito hasta el momento, por lo tanto se escribirá hasta que esta cantidad sea igual al tamaño del archivo.

→ El siguiente gráfico resume el procedimiento:



El script que se encuentra en el archivo: *ejercicios/scripts/ejercicio-3.sh* realiza este ejercicio automáticamente, recibe como argumento el nombre (**no path**) de algún archivo que esté en la carpeta */ejercicios/ftp/database/server/originales/*. Escribe la copia en el lado del cliente en la carpeta: */ejercicios/ftp/database/cliente/copias/* y la copia del servidor en la carpeta: */ejercicios/ftp/database/server/copias/*.

Ejemplo de uso con el archivo test.png:

```
root@ac1e56547eb3:/pdytr/practica-2/ejercicios/scripts# ./ejercicio-3.sh test.png
INFO - Iniciando entorno
INFO - Archivos compilados
INFO - Lanzando objeto remoto
INFO - Esperando objeto remoto
INFO - Lanzando cliente
TEST - Comparación de contenido entre ../ftp/database/server/originales/test.png ../ftp/database/client/copias/test.png ../ftp/database/server/copias/test.png
OK - Los archivos coinciden
INFO - Finalizando entorno
```

* *Notas*:

- *No es necesario ejecutar previamente el comando `rmiregistry`, ni tampoco `java StartRemoteObject` ya que el script lo hace.*
- *El script corre un diff entre cada par para testear que los 3 archivos tengan el mismo contenido.*
- *Si se quiere probar con otros archivos se recomienda copiarlos en la carpeta /ejercicios/ftp/database/server/originales/ para poder seguir usando el script.*

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Ejercicio 4

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).

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.

Para probar la concurrencia, realizamos un experimento en donde 2 clientes escriben un archivo con el mismo nombre en el servidor. Es decir, ambos clientes llaman al método **escribir** con diferente contenido de origen y el mismo nombre del archivo destinatario. Para ello desarrollamos el script *ejercicio-4.sh*, el cual hace lo siguiente:

Dado los clientes:

- **Cliente 1** → Tiene el archivo "*database/client/experimentos/experimento1.txt*" el cual tiene el contenido "Ramiro Lopes Canadell".
- **Cliente 2** → Tiene el archivo "*database/client/experimentos/experimento2.txt*" el cual tiene el contenido "Andres Milla".

Se ejecutan en paralelo los clientes mediante un script de bash con el mismo archivo destino:

```
# Ejecutamos los 2 clientes
java AskRemote -ejercicio4 $ARCHIVO_CLIENTE1 $ARCHIVO_SERVER &
CLIENT_1_PID=$!
java AskRemote -ejercicio4 $ARCHIVO_CLIENTE2 $ARCHIVO_SERVER &
CLIENT_2_PID=$!
```

Al realizar 10 ejecuciones de este experimento con una ventana de **2 bytes** encontramos que el contenido del archivo en el servidor difiere en varias salidas por efecto de la concurrencia:

---

*El siguiente output se encuentra en el archivo: **ejercicios/outputs/ejercicio-4**.*

```
------------------
Ejecución 0:
Andres Milla Ramiro
Lo
------------------
Ejecución 1:
Ramiro LopAnesd
Canade
------------------
Ejecución 2:
Andres Milla Ramiro
Lo
------------------
Ejecución 3:
Andres Milla Ramiro
Lo
------------------
Ejecución 4:
AnRamidresro Lopes
Ca
------------------
Ejecución 5:
Ramidrro Lopes
Canadeles
------------------
Ejecución 6:
Ramiro LAndreopes
Cana
------------------
Ejecución 7:
Ramiro Lopes
Canadell An
------------------
Ejecución 8:
Andres Milla Ramiro
Lo
------------------
Ejecución 9:
Ramiro LopAnesd
Canade
```

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Ejercicio 5

Inciso A

Para tomar el tiempo se declaró en la interfaz el siguiente método:

```
public void invocacion() throws RemoteException;
```

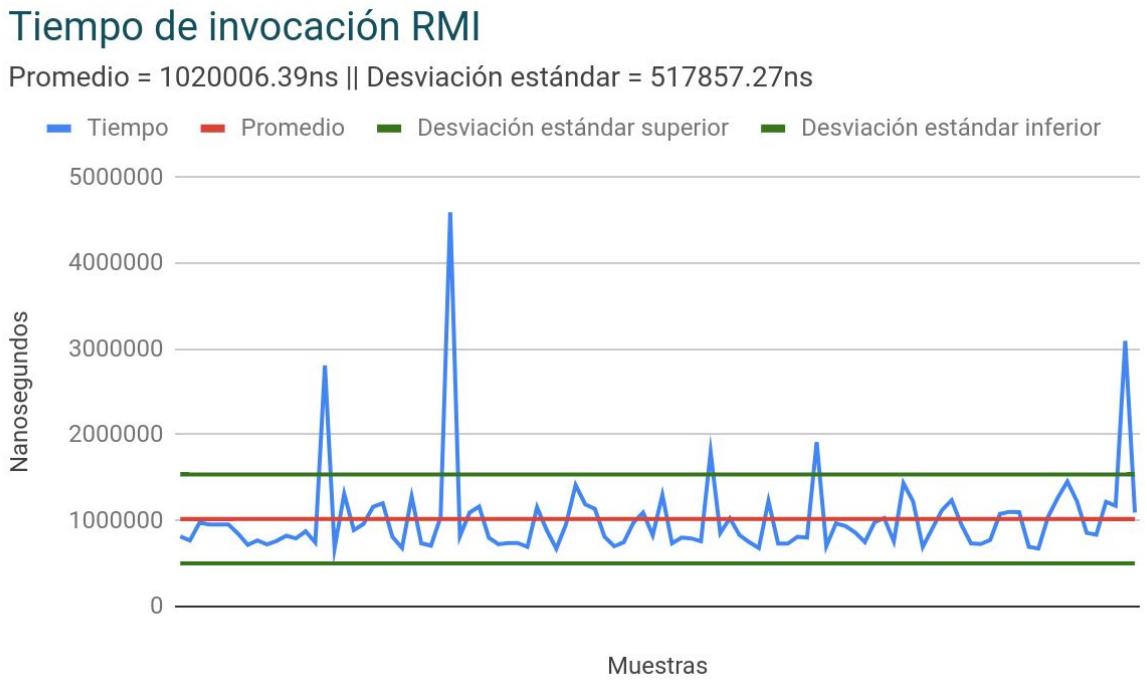
Y se lo implementó sin ninguna línea de código en el servidor, para no tener tiempo de procesamiento en la medición:

```
public void invocacion() throws RemoteException {}
```

Luego el cliente lo único que realiza es la medición en **nanosegundos** del tiempo que tomó la invocación:

```
long startTime = System.nanoTime();
remote.invocacion();
System.out.println(System.nanoTime() - startTime);
```

Realizamos el script ubicado en *ejercicios/scripts/ejercicio-5a.sh* que ejecuta al cliente 100 veces para medir el tiempo, guardamos las salidas en *ejercicios/outputs/ejercicio-5a.csv* y obtuvimos el siguiente resultado:



Inciso B

No existe un timeout predefinido, ni tampoco configurable en el paquete **java.rmi**. Pero existe un paquete de RMI llamado **sun.rmi** con el cual podemos setear varios tipos de timeouts. Estas propiedades se setean cuando se ejecutan los algoritmos, por ejemplo:

- **`sun.rmi.transport.tcp.responseTimeout:`** El valor de esta propiedad nos permite configurar el tiempo que un cliente RMI esperará la respuesta de un objeto remoto. Si el tiempo transcurrido supera al tiempo de espera seteado entonces se lanzará una excepción de la clase **RemoteException.** El valor predefinido de esta propiedad es 0 (es decir que no hay timeout de respuesta).

- Fuente: https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/sunrmiproperties.html

Para probar esto implementamos una operación remota que realiza un loop infinito, entonces el cliente al utilizarla nunca va a recibir una respuesta y se va a lanzar una excepción de **RemoteException**.
Realizamos el script ubicado en el archivo *ejercicios/scripts/ejercicio-5b.sh*. Dentro de este script seteamos el valor de la propiedad **responseTimeOut** con un valor razonable **(**3 segundos por ejemplo**)**:

```
# Inicia el entorno
./entorno.sh -start

java -cp "../ftp"  -Dsun.rmi.transport.tcp.responseTimeout=30000 AskRemote -ejercicio5b

#finaliza el entorno
./entorno.sh -stop
```

- Entonces al ejecutar el cliente va a lanzar el timeout luego de 3 segundos:

```
root@95ad1885ffb6:/pdytr/practica-2/ejercicios/scripts# ./ejercicio-5b.sh
INFO - Iniciando entorno
INFO - Abriendo puerto RMI en el directorio ftp
INFO - Archivos compilados
INFO - Lanzando objeto remoto
INFO - Esperando objeto remoto
ERROR - Tiempo de espera de respuesta agotado
INFO - Finalizando entorno
```