Practica 2 - RMI

Alumno: Cazorla, Ignacio Nicolás
N° de alumno: 13926/1

Contenidos

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.

Primero es necesario definir los conceptos de transparencia y transparencia de acceso.

- Transparencia: es definida como “un ocultamiento tanto al usuario como al programador de la separación de los componentes en un sistema distribuido, para que el sistema sea percibido como un todo, en vez de como una colección de componentes.”
- Transparencia de acceso: permite que recursos locales y remotos sean accedidos usando operaciones idénticas.

El punto clave, se encuentra en la situación en la cual las interfaces exponen la naturaleza distribuida de la llamada, por ejemplo admitiendo excepciones remotas, como es el caso de RMI en Java. Es necesario, tener en cuenta estas excepciones a la hora de usar RMI y lidiar con ellas durante la invocación remota, lo que a simple vista denota que la transparencia de acceso no es total, por lo que podemos decir que RMI no es completamente transparente.

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

Los archivos que deberían estar del servidor:

- IFaceRemoteClass.class
- StartRemoteClass.class
- RemoteClass.class

Los archivos que deberían estar en el cliente:

- IFaceRemoteClass.class
- AskRemote.class

Respecto del contenido, **IFaceRemoteClass** define las interfaces que deben ser implementadas por el objeto remoto, y las cuales luego serán invocadas por un cliente. En este caso define el método **sendThisBack()**.

El cliente es la clase **AskRemote** que invoca al método y **RemoteClass** es la que lo implementa, es decir esta última es el objeto remoto.

StartRemoteClass crea y registra al objeto remoto en un nombre/puerto. Esta clase instanciará un objeto de la clase RemoteClass.

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

El problema del desconocimiento de clases, proviene del mecanismo que usan las JVM para cargar clases dinámicamente. Las máquinas virtuales de java, tienen la capacidad de cargar clases en demanda. Puntualmente para RMI, se puede aprovechar esta característica, para descargar archivos de clases Java, incluyendo clases stubs, que son las que permiten ejecutar métodos remotos en el servidor desde un cliente.

Es importante tener en cuenta el concepto de “codebase”. Este concepto se refiere a la ubicación en la que están ubicadas las clases a cargar.

A partir de lo dicho anteriormente, dos problemas comunes que suelen presentarse son:

1. **En el servidor:** obtenemos error ClassNotFoundException intentando ligar o desligar un objeto remoto a un nombre en el registry. Esto es debido a que la petición realizada no es correcta, haciendo que el registry no pueda encontrar lo requerido por el stub remoto.
   Como el stub remoto implementa la misma interfaz que el objeto remoto, deben estar disponibles en el respectivo codebase.
   Frecuentemente esta excepción se debe a: omitir ‘/’ al final de la propiedad en la URL, la URL no es correcta, el path al codebase es incorrecto o las clases solicitadas en determinada URL no están disponibles.
2. **En el cliente:** podemos recibir la excepción ClassNotFoundException intentando buscar un objeto remoto en el registry. En ese caso, el problema se encuentra en el Classpath con el cual arrancó el registry.

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 :
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.

Las operaciones requeridas para este inciso, fueron agregadas en la clase RemoteClass provista, realizando las modificaciones necesarias para lograr lo solicitado. La interfaz IfaceRemoteClass también es modificada estableciendo los nuevos métodos que debe implementar RemoteClass.

Por un lado, se provee el método leer, que permite a un cliente obtener un archivo del servidor. El primer parámetro que recibe el servidor es el archivo requerido, el segundo la posición desde dónde leer dentro del archivo, y el tercero es la cantidad de datos que está dispuesto a recibir el cliente en esa invocación.

Por el otro, se provee el método escribir, que recibirá datos de un cliente para escribirlos en un archivo. Este método realiza una escritura no destructiva, por lo que si es necesario correr varias veces el programa, por lo que es necesario limpiar el directorio de copias si no queremos que las próximas ejecuciones concatenen datos al final de estas.

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 item anterior, sin cambios específicos del servidor para este item 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.

Por contrapartida al inciso anterior, se modificó la clase AskRemote de la siguiente manera:

- Se estableció la capacidad de datos que va a recibir/enviar el cliente en bufferLength
- Se agregaron las llamadas a los métodos remotos **leer()** y **escribir()**, provistos por la interfaz remota IfaceRemoteClass

Lo que hace el método main para este inciso es leer desde el servidor un archivo, con la operación respectiva, e inmediatamente escribir el mismo archivo en el servidor nuevamente.

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 ejecutar más de una invocación remota a la vez. Para realizar la demostración se utilizó el script **ejercicio4.sh**, el cual permite ejecutar dos clientes en paralelo. Este script repite 10 veces el proceso de ejecutar ambos clientes. La operación que se lleva a cabo es

escribir(), con los archivos de entrada archivo_proc_1.txt y archivo_proc_2.txt, cada uno representando a un cliente, con los contenidos “hola” y “chau” respectivamente. Adicionalmente, se modificó el tamaño del buffer de envio a 2 bytes. La salida obtenida durante la ejecución fue:

Ejecucion 1 chauho Ejecucion 2 holach Ejecucion 3 holach Ejecucion 4 chauho	Ejecucion 5 chauho Ejecucion 6 holach Ejecucion 7 chauho	Ejecucion 8 holach Ejecucion 9 holach Ejecucion 10 chauho

Se puede apreciar entonces, que la escritura en paralelo de los clientes hacia el servidor, produce interferencias.

Para manejar esto la operación de escritura deberá llevarse a cabo de manera atómica por cada invocación. Es decir, cuando un cliente solicita una escritura, el mismo debe terminar de escribir para que el servidor pueda atender una nueva invocación remota.

Nota: antes de ejecutar el script es necesario tener corriendo rmiregistry.

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 calcular los tiempos de comunicación, la clase cliente **AskRemote** invoca al método **invocación()** del objeto remoto. Dicho método, no efectúa ninguna operación para mantener el tiempo de respuesta lo más cercano posible al tiempo que toma la invocación en sí.

Para probar este experimento, es necesario ejecutar a la clase AskRemote con los argumentos “-5” y el número de veces que se va a repetir el proceso.

El número de repeticiones elegido para tomar la muestra fue el 100. La salida se ve representada en la terminal luego de la ejecución, mostrando cada tiempo, el promedio y la desviación estándar de la comunicación. El comando ejecutado fue el siguiente:

```
$ java AskRemote -5a 100
```

Los resultados obtenidos fueron representados en el gráfico. La muestra está expresada en microsegundos, donde el promedio fue de 530 y la desviación estándar de 309.

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.

No existe ningún timeout predefinido para RMI.

La verificación se realizó ejecutando el método remoto **waitTimeoutInfiniteLoop()**, el cual contiene un loop infinito. No se recibe ninguna respuesta por parte del servidor, sino más bien, el cliente queda a la espera.

La alternativa es utilizar las propiedades indicadas en el enlace para setear un timeout.

```
$ java -Dsun.rmi.transport.tcp.responseTimeout=15000 AskRemote -5b
```

Con el argumento -D estamos seteando la propiedad indicada (del paquete sun), para que se produzca un timeout pasados los 15000 milisegundos. De esta manera podemos evitar

que el cliente quede a la espera de alguna respuesta indefinidamente. Con esta propiedad establecida, el cliente recibe una respuesta pasado el tiempo indicado.

Bibliografía

Capítulos 1 y 5, Distributed systems - Concepts and design.

Java Remote Method Invocation Specification.

Desconocimiento de clases,
https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/codebase.html ,
https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/faq.html#wrongcodebase.

RMI Timeouts,
https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/sunrmiproperties.html