Práctica 3
Programación Distribuida y Tiempo Real

Lucas Di Cunzolo
Santiago Tettamanti

Abstract—Remote Method Invocation

Index Terms—Programación Distribuida y Tiempo Real, java, LaTeX, RMI

1 Punto 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*

Colouris: Access transparency. Enables local and remote resources to be accessed using identical operations They both offer a similar level of transparency that is, local and remote calls employ the same syntax but remote interfaces typically expose the distributed nature of the underlying call, for example by supporting remote exceptions

Vemos que sintácticamente las operaciones son idénticas y no se podría saber si la operación es local o remota dado que el los nombres de las operaciones son exactamente los mismos. sin embargo a la hora de llamar a la clase hay que hacer un lookup del nombre de esta, al no estar en el espacio de nombres local, teniendo el cuenta el hostname remoto. Por lo que si bien a nivel invocación de métodos nunca no enteramos si es local o remoto y no se hace ninguna diferencia en la invocación de ambos, a la hora de obtener la clase remota si. Todo esto sin tener en cuenta el manejo de posibles errores en la comunicación ya que el método remoto tiende a fallar mucho más seguido que el local mayormente por fallas en la comunicación que son inexistentes en el otro. En conclusión, teniendo en cuenta lo definido en el libro, RMI si ofrece una tranparencia de acceso al permitir invocar tanto un procedimiento local como remoto de la misma manera. Pero esa transparencia no es total ya que hay que tener en cuenta otras cuestiones como el lookup de la clase remota y la posible falla en la comunicación.

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

- **AskRemote.class:** Presente en el cliente, identifica a la clase cliente que realiza el llamado al metodo remoto. Contiene, entre otras cosas, información de conexión al servidor tal como su hostname, invocación a la clase y al método remoto.

- **StartRemoteObject.class:** Presente en el servidor. Es la clase encargada de instanciar a la clase contenedora de los métodos remotos a invocar y la registra en el espacio de nombres asegurándose que pueda ser llamada por el cliente.

- **IfaceRemoteClass.class:** Presente en el servidor; es la interface que define los métodos que pueden ser invocados y que la clase del servidor debe implementar.

- **RemoteClass.class:** Presente en el servidor; solo implementa los métodos que serán invocados remotamente presentes en la interfáz mencionada anteriormente. Extiende de UnicastRemoteObject, lo que hace toda la comunicación totalmente transparente al programador desde este lado.

2 PUNTO 2

Identifique similitudes y diferencias entre RPC y RMI.

La principal diferencia entre RPC y RMI es que RMI involucra objetos. En lugar de llamar procedimientos de forma remota llama a métodos de objetos remotos. Se podría decir que RMI es la version orientada a objetos de RPC. En RMI cada objeto tiene una referencia única en todo el sistema, tanto si los objetos son remotos como locales, permitiendo así el pasó de objetos como parámetros, herencia y relaciones entre objetos remotos, ofreciendo un pasaje de parámetros con una mayor riqueza semántica y un mayor nivel de abstracción que en RPC, donde solo se pueden pasar parámetros por valor.

En ambos modelos los detalles de implementación están ocultos al usuario y los métodos o procesos disponibles se conocen a través de una interfaz que provee el servidor. Ambos ofrecen un alto nivel de transparencia de acceso, llamadas remotas y locales poseen la misma sintáxis pero las interfaces remotas diferencian la naturaleza distribuida de las llamadas remotas al, por ejemplo, soportar excepciones remotas o tener en cuenta posibles fallas en la comunicación con tiempos de timeout.

3 PUNTO 3

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

En rmi puede ser que los métodos de un objeto remoto admitan como parámetros a otras clases y devuelvan clases. Por lo que un metodo remoto podría devolver un objeto o admitir un parámetro de una clase desconocida para el cliente. En ese caso se podría generar una ClassNotFoundException Por ejemplo, puede ser un método así

public InterfaceResultado dameResultado (InterfaceParametro parametro) throws java.rmi.RemoteException ...

Una solucion sería que tanto el servidor como el cliente tengan sus propias copias de las clases que implementen estas interfaces. Es decir, si ClaseResultado implementa InterfaceResultado y ClaseParametro implementa InterfaceParametro, tanto el cliente como el servidor deben tener en su CLASSPATH los ficheros ClaseResultado.class y ClaseParametro.class. Esto no sería una solucion deseada, ya que se deben saber todas las clases que utiliza el servidor y copiarlas del lado del cliente, quitando dinamismo y escalabilidad a rmi. La solución provista por RMI es instalar RMISecurityManager, el cual habilita la carga dinámica de clases. De esta forma el servidor no necesita tener una copia de ClaseParametro.class ni el cliente una copia de ClaseResultado.class.

4 PUNTO 4

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

a - *Defina e implemente con RMI un servidor.*
Para esto se utilizó la estructura de archivos del ejemplo. Se definió una interfaz con 3 métodos, *read*, *write* y *list* Tanto el cliente como el servidor, se alimentan de la interfaz definida en la clase IfaceRemoteClass. (Ver apéndice - Subsección IfaceRemoteClass)
Donde la clase que los implementa es la RemoteClass (Ver apéndice - Subsección RemoteClass)

Luego se implementó un cliente que interactúe contra el servidor, con los comandos basicos, nombrados anteriormente. Esto se hizo en la clase AskRemote (Ver apéndice - Subsección AskRemote)
El servidor, utiliza la implementación de la clase RemoteClass, y corre instanciando un objeto de la clase StartRemoteObject. (Ver apéndice - Subsección StartRemoteObject)

5 PUNTO 5

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: diseñar un experimento con el que se pueda demostrar fehacientemente que dos o más invocaciones remotas se ejecutan concurrentemente o no. Compare este comportamiento con lo que sucede con RPC.

6 PUNTO 6

Tiempos de respuesta de una invocación:

a - Diseñe experimento que muestre el tiempo de respuesta mínimo de una invocacióncon JAVA RMI. Muestre promedio y desviaciónestándarde tiempo respuesta.

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

APPENDIX

Código Java

.1 IfaceRemoteClass

```
import java.rmi.Remote;
import java.rmi.RemoteException;
/* This interface will need an implementing class */
public interface IfaceRemoteClass extends Remote
{
    /* It will be possible to invoke this method from an application
       in other JVM */
    public byte[] read(String path, int position) throws
       RemoteException;
    public int write(String path,byte[] data) throws RemoteException;
    public String list(String data) throws RemoteException;
}
```

.2 AskRemote

```
import java.rmi.Naming; /* lookup */
import java.rmi.registry.Registry; /* REGISTRY_PORT */
import java.util.HashMap;
import java.util.Map;
import java.io.File;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Arrays;
import java.nio.file.StandardOpenOption;
import java.io.IOException;


public class AskRemote{

    public static void read(IfaceRemoteClass remote, String localPath,
       String remotePath){
        try {
            byte endFile=0;
            int position=0;
            while (endFile != 1){
                byte[] fileContent = remote.read(remotePath,position);
                endFile=fileContent[fileContent.length-1];
                fileContent=Arrays.copyOf(fileContent,fileContent.length-1);
                position+=fileContent.length;
                try{
                    Files.write(Paths.get(localPath),
                       fileContent,StandardOpenOption.APPEND);
                }
                catch (IOException e) {
```

```
                Files.createFile(Paths.get(localPath));
                Files.write(Paths.get(localPath),
                   fileContent,StandardOpenOption.APPEND);
            }
        }
        System.out.printf("%s\n","El archivo se leyo
           correctamente");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void write(IfaceRemoteClass remote, String
   localPath, String remotePath){
    try {
        byte[]
           fileContent=Files.readAllBytes(Paths.get(localPath));
        byte[] partialContent;
        int fileSize=fileContent.length;
        int bytesReaded=0;

        while (bytesReaded<fileSize){
            partialContent=
               Arrays.copyOfRange(fileContent,bytesReaded,fileSize);
            int writeReturn =
               remote.write(remotePath,partialContent);
            bytesReaded+=writeReturn;
        }
        System.out.printf("%s\n", "The file was correctly
           written");

    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void list(IfaceRemoteClass remote, String path){

     try {
        String listReturn = remote.list(path);
        System.out.printf("%s\n", listReturn);
    } catch (Exception e) {
        e.printStackTrace();
    }
}


public static void main(String[] args)
{
```

```
/* Look for hostname and msg length in the command line */
if (args.length < 2) {
    System.out.printf("Usage: AskRemote\n"
    + "\t- <host> write <local> <remote>: Add a file from
       <local> to <remote> \n"
    + "\t- <host> add <local> <remote>: Add a file from
       <local> to <remote>\n"
    + "\t- <host> read <local> <remote>: Store a file from
       <local> to <remote>\n"
    + "\t- <host> get <local> <remote>: Store a file from
       <local> to <remote>\n"
    + "\t- <host> list <remote directory>: List files from
       <remote directory>\n"
    + "\t- <host> ls <remote directory>: List files from
       <remote directory>\n"
    );
    System.exit(1);
}

try {
    String rname = "//" + args[0] + ":" +
       Registry.REGISTRY_PORT + "/remote";
    IfaceRemoteClass remote = (IfaceRemoteClass)
       Naming.lookup(rname);

    switch (args[1]) {
            case "read":
            case "get":
                    if (args.length != 4)
                    {
                        System.out.println("4 argument needed:
                           (remote) hostname, command , local
                           directory and remote directory");
                        System.exit(1);
                    }
                    else
                        read(remote,args[2],args[3]);
                    break;
            case "write":
            case "add":
                    if (args.length != 4)
                    {
                        System.out.println("4 argument needed:
                           (remote) hostname, command , local
                           directory and remote directory");
                        System.exit(1);
                    }
                    else
                        write(remote,args[2],args[3]);
```

```
                        break;
                    case "list":
                    case "ls":
                        if (args.length != 3)
                        {
                            System.out.println("3 argument needed:
                               (remote) hostname, command and
                               directory");
                            System.exit(1);
                        }
                        else
                            list(remote,args[2]);
                        break;
                    default: System.out.println("Command unavailable");
                        break;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

.3 RemoteClass

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.nio.file.DirectoryStream;
import java.util.Arrays;
import java.nio.file.StandardOpenOption;
import java.io.IOException;

/* This class implements the interface with remote methods */
public class RemoteClass extends UnicastRemoteObject implements
   IfaceRemoteClass
{
    static final long serialVersionUID = 1;

    protected RemoteClass() throws RemoteException
    {
        super();
    }
    /* Remote method implementation */
    public byte[] read(String path, int position) throws
       RemoteException
    {
        try {
```

```
            byte[] contents = Files.readAllBytes(Paths.get(path));
            int fileSize=contents.length;
            contents=Arrays.copyOfRange(contents,position,fileSize);
            byte fileEnd=1;
            if (contents.length>1024){
                contents=Arrays.copyOf(contents,1024);
                fileEnd=0;
            }
            byte[]
               sizeAndContent=Arrays.copyOf(contents,contents.length+1);
            sizeAndContent[contents.length]=fileEnd;
            return sizeAndContent;
        } catch(Exception e) {
            System.out.println(e);
            return new byte[0];
        }
    }

    public int write(String path,byte[] data) throws RemoteException
    {
        try {
            if (data.length>1024)
                data=Arrays.copyOf(data,1024);
            try{
                Files.write(Paths.get(path),
                   data,StandardOpenOption.APPEND);
            }
            catch (IOException e) {
                Files.createFile(Paths.get(path));
                Files.write(Paths.get(path),
                   data,StandardOpenOption.APPEND);
            }
            System.out.println(data.length);
            return data.length;
        } catch (Exception e) {
            System.out.println(e.toString());
            return -1;
        }
    }

    public String list(String path) throws RemoteException
    {
        try (DirectoryStream<Path> paths =
           Files.newDirectoryStream(Paths.get(path))) {
            String directoryPaths = "";
            System.out.printf("Listing files in %s\n", path);
            for (Path p : paths) {
                directoryPaths += p.toString();
                directoryPaths += "\n";
```

```
            }
            return directoryPaths;
        } catch(Exception e) {
            System.out.println(e);
            return e.toString();
        }
    }
}
```

.4 StartRemoteObject

```
import java.rmi.registry.Registry; /* REGISTRY_PORT */
import java.rmi.Naming; /* rebind */
public class StartRemoteObject
{
    public static void main (String args[])
    {
        try{
        /* Create ("start") the object which has the remote method */
            RemoteClass robject = new RemoteClass();
            /* Register the object using Naming.rebind(...) */
            String rname = "//localhost:" + Registry.REGISTRY_PORT +
               "/remote";
            Naming.rebind(rname, robject);
        } catch (Exception e) {
            System.out.println("Hey, an error occurred at
               Naming.rebind");
            e.printStackTrace();
            System.out.println(e.getMessage());
        }
    }
}
```