Remote Method Invocation (RMI)

# Introduction

We can read/write all kinds of files in Java. We will look at such a simple example and then go over Remote Method Invocation (RMI). In RMI we will learn how to call a method that is residing on a remote server. But let us get a bit familiar with remote file-reading in Java first.

## 1. Remote I/O files

We can read all kinds of files from a remote server. To be able to input a remote file in LINUX/UNIX operating system we should have the file in the folder public html.

### Example 1.

The following program reads from an mp4 file on a remote server. The server can be any remote server that you might have access to.

**import** java.io.\*;

**import** java.net.URL;

**public** **class** Main{

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

FileOutputStream out = **null**;

InputStream in = **null**;

**try**{

URL u = **new** URL("http://blah.cis.njit.edu/~rsiddique/a.mp4");

in = u.openStream();

out = **new** FileOutputStream("b.mp4");

**int** data = in.read();

**while**(data != -1){

out.write(data);

data = in.read();

}

out.close();

in.close();

}**catch**(Exception e){

System.*out*.println(e);

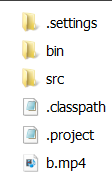
}

System.*out*.println("Program is terminated"); //Please wait until this message is displayed.

}

}

The output file: b.mp4 will be located in the eclipse project folder:



Click on b.mp4 to listen to music (mp4 is a music file).

#### Description

The class URL creates a url object to refer to the website.

u.openStream(): Opens an input stream to the URL.

FileOutputStream("b.mp4"): Creates a file output stream to write to the file named: b.mp4.

**int** data = in.read(): Reads one byte from the input file. When the end-of-file is read this method returns -1.

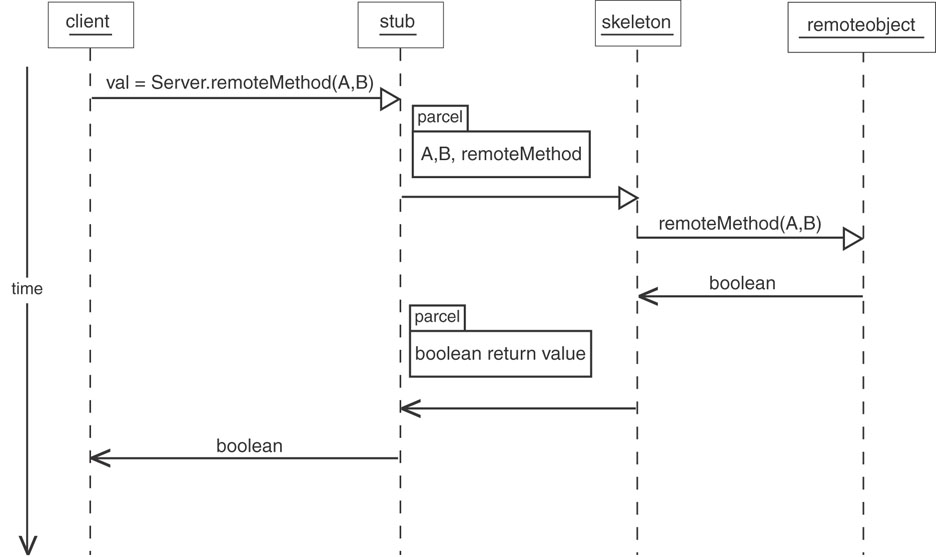
The while-loop keeps reading bytes from the file: a.mp4 and writes to the file: b.mp4.

Finally the program closes both file.

## 2. Remote Method Invocation (RMI)

Remote Method Invocation (RMI) supports Objects call. RMI allows a Java program on one machine to invoke (call) a method on a remote object.

To make remote methods transparent to both the client and the server, RMI implements the remote object using **stubs** and **skeletons**. A **stub** is a proxy (representative) for the remote object and resides in the client. When the client invokes a remote method, it is this stub that is called. The stub creates a **parcel** consisting of the name of the method and its parameter. This action is called **marshalling** the parameters. The stub sends the parcel to the server. The server has a **skeleton**. It is this skeleton that receives the parcel. The skeleton un-marshal the parameters and invokes the desired method on the server side. Then the skeleton marshals the return value of the method and sends it back to the stub on the client side. The client then un-marshals the return value.



I explain the API through a simplified version of the client server programs.

### Example 2

In the following client-server programs the client calls a method of the server.

#### The client side class and interface:

**import** java.rmi.\*;

**public** **interface** HelloInterface **extends** Remote {

**public** String sayHello(String name) **throws** java.rmi.RemoteException;

}

**public** **class** HelloClient {

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

**try**{

**int** port = 16790;

String host = "localhost";

String registryURL = "rmi://" + host + ":" + port + "/hello";

HelloInterface h = (HelloInterface)Naming.*lookup*(registryURL);

System.*out*.println("Lookup completed " );

String message = h.sayHello("Fun Java");

System.*out*.println("HelloClient: " + message);

}

**catch** (Exception e){

e.printStackTrace();

}

}

}

#### The server side classes and interface:

**import** java.rmi.\*;

**import** java.rmi.server.\*;

**import** java.rmi.registry.Registry;

**import** java.rmi.registry.LocateRegistry;

**import** java.net.\*;

**public** **class** HelloServer{

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

**try**{

**int** port = 16790;

String host = "localhost";

HelloImpl exportedObj = **new** HelloImpl();

*startRegistry*(port);

String registryURL = "rmi://" + host + ":" + port + "/hello";

LocateRegistry.*getRegistry*(port);

Naming.*rebind*(registryURL, exportedObj);

System.*out*.println("Hello Server ready.");

}**catch** (Exception e){

e.printStackTrace();

}

}

**private** **static** **void** startRegistry(**int** port)**throws** RemoteException{

**try**{

Registry registry = LocateRegistry.*getRegistry*(port);

registry.list( ); // This call will throw an exception if the registry is not already running

}**catch** (RemoteException e){

// No valid registry at that port is running.

System.*out*.println ("RMI registry cannot be located at port " + port);

Registry registry = LocateRegistry.*createRegistry*(port);

System.*out*.println("RMI registry created at port " + port);

}

}

}

**import** java.rmi.\*;

**import** java.rmi.server.\*;

**public** **class** HelloImpl **extends** UnicastRemoteObject **implements** HelloInterface {

**public** HelloImpl() **throws** RemoteException {

**super**( );

}

**public** String sayHello(String name) **throws** RemoteException {

**return** "Hello, World!" + name;

}

}

### Description:

The interface:

**public** **interface** HelloInterface **extends** Remote{

**public** String sayHello(String name) **throws** java.rmi.RemoteException;

}

This is a java interface. It informs the client that the method: sayHello is implemented remotely (on the server side). This interface is not implemented on the client side.

**Note: Since we write all the programs with eclipse you do not have to memorize which exception class should be used here and there. The eclipse tells you which exceptions you need.**

Note that this interface is in both the client and server programs. This interface is implemented by the class: HelloImpl in the server program but is not implemented in the client program.

The RMI system has a registry object. Every remote object must have a name registered in the registry.

#### The server side:

The class **HelloImpl**:

* The server creates an object of this class. This object is going to be the remote object. The method: sayHello of this remote object can be called by the client program.

The class **HelloServer**:

* Method: main:
  + String registryURL = "rmi://" + host + ":" + port + "/hello": This make the string: "rmi://localhost:16790/hello". This string is going to be registered in the registry as the name of the remote object that the variable: exportedObj refers to.
  + *startRegistry*(port): Locates an rmi registry at a specific port. This method is explained in the next paragraph.
  + Naming.*rebind*(registryURL, exportedObj): **Naming** is a class in the package rmi. The method: *rebind* records (binds) the above string (blue underlined) as the name of the remote object: exportedObj.
* Method *startRegistry*(port): There are two important method calls:
  + LocateRegistry.*getRegistry*(port): Locates an rmi registry.
  + registry.list( ): This rmi method returns an array of names (strings). Each name refers to a remote object. By a for-statement we can print all these names. If there is no registry at this port the control of execution jumps to the catch block. In the catch block we again have: LocateRegistry.*getRegistry*(port). On this second call the system starts running the registry.

#### The client side:

The class **HelloClient**:

* String registryURL = "rmi://" + host + ":" + port + "/hello": This make the string: "rmi://localhost:16790/hello". This string **must be** in the rmi registry as the name of a remote object. The variable: host is the host where the registry is located. The variable: port is the port number on which the registry accepts calls. Both host and port are optional. If host is omitted, the host defaults to the local host. If port is omitted, then the port defaults to 1099, the "well-known" port that rmi registry uses.
* HelloInterface h = (HelloInterface)Naming.*lookup*(registryURL): The method: looks up for the argument (the blue underlined). This name refers to a remote object therefore the variable: h refers to the remote object. Now h can access to any public method and variable of this remote object which is of type class: HelloImpl.
* h.sayHello("Fun Java"): passes the argument: to the following method. The method returns the string: "Hello, World! Fun Java".

**public** String sayHello(String name) **throws** RemoteException{

**return** "Hello, World!" + name;

}

## 3. Building an RMI Application

Create a folder named: **ExampleRMI**. In this folder make two folders named: **Server** and **Client**.

Click on the eclipse icon and make the folder: Server workstation and make a project named: **Server Project**. Make two classes named HelloServer.java and HelloImpl.java. Make a java user interface named: HelloInterface.java. Copy the above classes HelloServer.java, HelloImpl.java, and the interface HelloInterface.java.

Click on the eclipse icon and make the folder: Client workstation and make a project named: **Client Project**. Make a class named HelloClient.java. Make a java user interface named: HelloInterface.java. Copy the above classes HelloClient.java, and the interface. HelloInterface.java.

1. Run the server to see:

RMI registry cannot be located at port 16790

RMI registry created at port 16790

Hello Server ready.

2. Run the client to see:

Lookup completed

HelloClient: Hello, World! Fun Java

## 4. Advanced RMI programs

This section demonstrates advanced application of RMI programs. Like the prior example, this program also uses APIs.

### 4.1 Client Callback

A client program in real time does not know when the remote object is available. For example it takes some time to obtain the result (like sorting millions of names) that should be sent back to the client. In this case the client program has to **poll** (check) for the availability of the result repeatedly until it is available. This idea can be coded as follows:

CallbackServerInterface h = (CallbackServerInterface)Naming.*lookup*(registryURL);

**while**(!(h.hasBiddingStarted())**;**

Where the method: hasBiddingStarted() is a remote method that returns *true* when the remote object obtains the result to be sent to the client. As long as this remote method returns *false* the above while-loop cycles around. Obviously this is not an efficient solution. The client program is wasting CPU time sitting on this loop. A more efficient solution is the **callback** technique. In this technique we code the client to go to sleep for a period of time. In this case CPU leaves the client program to do (execute) some other useful work for other programs in the system. Let us clarify this technique through an example.

### Example 3

The following client-server program implements the callback technique.

**Interfaces**: Both sides have the same two interfaces:

**import** java.rmi.\*;

**public** **interface** CallbackClientInterface **extends** java.rmi.Remote{

**public** String notifyMe(String message) **throws** java.rmi.RemoteException;

}

**import** java.rmi.\*;

**public** **interface** CallbackServerInterface **extends** Remote {

**public** String sayHello( ) **throws** java.rmi.RemoteException;

**public** **void** registerForCallback(CallbackClientInterface callbackClientObject)

**throws** java.rmi.RemoteException;

**public** **void** unregisterForCallback(CallbackClientInterface callbackClientObject)

**throws** java.rmi.RemoteException;

}

#### The server side:

**import** java.rmi.\*;

**import** java.rmi.server.\*;

**import** java.util.Vector;

**public** **class** CallbackServerImpl **extends** UnicastRemoteObject **implements**

CallbackServerInterface{

**private** Vector clientList;

**public** CallbackServerImpl() **throws** RemoteException{

**super**( );

clientList = **new** Vector();

}

**public** String sayHello( )**throws** java.rmi.RemoteException{

**return**("hello");

}

**public** **synchronized** **void** registerForCallback(CallbackClientInterface

callbackClientObject)**throws** java.rmi.RemoteException{

**if** (!(clientList.contains(callbackClientObject))){

clientList.addElement(callbackClientObject);

System.*out*.println("Registered new client ");

doCallbacks();

}

}

**public** **synchronized** **void** unregisterForCallback(

CallbackClientInterface callbackClientObject) **throws** java.rmi.RemoteException{

**if** (clientList.removeElement(callbackClientObject)){

System.*out*.println("Unregistered client ");

}**else**{

System.*out*.println("unregister: clientwasn't registered.");

}

}

**private** **synchronized** **void** doCallbacks( ) **throws** java.rmi.RemoteException{

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n Callbacks initiated ---");

**for** (**int** i = 0; i < clientList.size(); i++){

System.*out*.println("doing "+ i +"-th callback\n");

CallbackClientInterface nextClient =

(CallbackClientInterface)clientList.elementAt(i);

nextClient.notifyMe("Number of registered clients=" + clientList.size());

}

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n Server completed callbacks ---");

}

}

**import** java.rmi.\*;

**import** java.rmi.server.\*;

**import** java.rmi.registry.Registry;

**import** java.rmi.registry.LocateRegistry;

**import** java.net.\*;

**public** **class** CallbackServer{

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

**try**{

**int** port = 1234;

String host = "localhost";

*startRegistry*(port);

CallbackServerImpl exportedObj = **new** CallbackServerImpl();

String registryURL = "rmi://" + host + ":" + port + "/callback";

Naming.*rebind*(registryURL, exportedObj);

System.*out*.println("Callback Server ready.");

}

**catch** (Exception re){

System.*out*.println("Exception in HelloServer.main: " + re);

}

}

**private** **static** **void** startRegistry(**int** RMIPortNum)**throws** RemoteException{

**try**{

Registry registry = LocateRegistry.*getRegistry*(RMIPortNum);

registry.list( );

}

**catch** (RemoteException e) {

Registry registry = LocateRegistry.*createRegistry*(RMIPortNum);

}

}

}

#### The client side:

**import** java.rmi.\*;

**import** java.rmi.server.\*;

**public** **class** CallbackClientImpl **extends** UnicastRemoteObject

**implements** CallbackClientInterface {

**public** CallbackClientImpl() **throws** RemoteException {

**super**( );

}

**public** String notifyMe(String message){

String returnMessage = "Call back received: " + message;

System.*out*.println(returnMessage);

**return** returnMessage;

}

}

**import** java.util.Scanner;

**import** java.rmi.\*;

**public** **class** CallbackClient {

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

Scanner keyboard = **new** Scanner(System.*in*);

**try**{

**int** port = 1234;

String host = "localhost";

System.*out*.println("Enter how many seconds to stay registered:");

**int** time = keyboard.nextInt();

String registryURL = "rmi://" + host + ":" + port + "/callback";

CallbackServerInterface h = (CallbackServerInterface)Naming.*lookup*(registryURL);

System.*out*.println("Lookup completed " );

System.*out*.println("Server said " + h.sayHello());

CallbackClientInterface callbackObj = **new** CallbackClientImpl();

h.registerForCallback(callbackObj);

System.*out*.println("Registered for callback.");

**try**{

Thread.*sleep*(time \* 1000);

}**catch** (InterruptedException ex){

}

h.unregisterForCallback(callbackObj);

System.*out*.println("Unregistered for callback.");

}**catch** (Exception e) {

System.*out*.println("Exception in CallbackClient: " + e);

}

}

}

We run the server side first and then the client side and enter data on the client console.

**A sample dialog**:

**Client side console**:

Enter how many seconds to stay registered:

10

Lookup completed

Server said hello

Call back received: Number of registered clients=1

Registered for callback.

**Server side console**:

Callback Server ready.

Registered new client

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Callbacks initiated ---

doing 0-th callback

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Server completed callbacks ---

Unregistered client

#### Description

Remember Vector – it’s like the ArrayList. Vector is **synchronized**, which means only one thread at a time can access the code, while arrayList is**not synchronized**, which means multiple threads can work on arrayList at the same time. For example, if one thread is performing an add operation, then there can be another thread performing a remove operation in a multithreading environment.

In this approach both sides need a remote method calls of the other side. Therefore both sides have remote objects. Obviously like the previous example the remote object on the server side has the method: *sayHello* and the client program calls this remote method.

#### The server program:

**interface** CallbackClientInterface: The method: notifyMe of this interface won’t be implemented by the server program. The stub of the server calls this remote method which is residing in a remote object on the client side.

**interface** CallbackServerInterface: The methods sayHello and registerForCallback are implemented on the server side and the client calls them remotely.

Therefore the roles of these two interfaces on the client side are switched.

#### The class CallbackServerImpl:

* Since many clients may call the remote object of this server program we need a java Vector: clientList to record them.
* The method sayHello: This method is called by the client side remotely to receive: "hello".
* The method registerForCallback: This is a method that is called by many clients remotely. Since every client is changing the vector: clientList by adding itself which is referred by the parameter: callbackClientObject, this method must be a synchronized to prevent more than one client going inside its body. Therefore any local call like: doCallbacks in the body of this method must also be a synchronized method.
* The method doCallbacks: This method is called by the above method. The major instruction in the body of this method is the remote (on the client side) method call: notifyMe(...). The server side is asking the client to notify him that has received the word: "hello".
* The method unregisterForCallback: This remote call which is called by a client removes the client from the vector and this means this client is not registered anymore.

**The class CallbackServer:**

* The method main: It creates a remote object of the type of the class: CallbackServerImpl to be used by the clients remotely.

**The client side program**:

The interfaces are doing the opposite of the ones in the server side. Their roles are explained in the description of the server program.

**The class CallbackClientImpl:**

* The method notifyMe: This is a remote method which is called by the server. This method returns a message (Call back received...) to let the server which means this client is connected and registered.

**The class CallbackClient**:

* The method main:
  + CallbackServerInterface h = (CallbackServerInterface)Naming.*lookup*(registryURL): Finds the remote object created by the server and makes h refer to it. Now we can call any remote method on the server via: h.
  + h.sayHello(): Now the client via: h is calling the remote method: sayHello which is on the server side to get the word: hello.
  + **new** CallbackClientImpl(): The main method creates a remote object that the server can call its methods.
  + h.registerForCallback(callbackObj): Calls this remote method to store itself (referred by the variable: callbackObj) in the vector which is on the server side.
  + Thread.*sleep*(time \* 1000): The call to the method: *sleep* postpones the execution of the rest of the instructions in main for a few second. Therefore we are not using the polling technique explained at the beginning of this section to waste the time. This CPU can now go and execute some of the programs of the system. After the sleeping time is over it will come back to execute the rest of the instructions.
  + h.unregisterForCallback: Calls the remote method (defined on the server) to remove itself from the vector.