UNIVERSITY OF PORTSMOUTH

# OS Week 6 - Lab

# Remote Method Invocation (RMI)

## Introduction

In the Week 6 lecture on Interprocess Communication, we discussed *Remote Procedure Call* as an important paradigm for communication between processes on different hosts (or on the same host).

*Java RMI* is a more or less contemporary implementation of this general paradigm. "RMI" stands for *Remote Method Invocation* - "method invocation" being the object-oriented generalization of "procedure call".

In this lab we will use Java RMI to gain practical experience with the basics of Remote Procedure Call, which (together with *Message Passing*) remains one of the dominant models for interprocess communication, especially in distributed systems.

This week there is only one relatively complex experiment to record in your lab book. We expect that you record in some level of detail the procedures you followed to set up the client and server end of the protocol, and any problems you encountered. Then there is a reflective question followed exercises that progressively develop the main experiment into a simple kind of chat service between two computers.

## Java Interfaces and Remote Interfaces

Before embarking on Java RMI we need to remind ourselves about an important feature of the Java language. In Java, an *interface* is something like a stripped down *class*. But where a class contains method *implementations*, as well as data fields, an interface typically only defines *method interfaces*.

Here is a simple example of an interface definition, which we will develop further below:

```
public interface MyInterface {

    void printMessage(String message);
}
```

Notice that although this interface contains the *declaration* of a method called `printMessage`, it does not contain the implementation of that method - there is just a semicolon in place of the method body.

For now, don't try to create this interface in NetBeans or your preferred development environment - just study it. We will be using similar interfaces in the practice in the next section.

To be useful, an interface must be *implemented* by a class, for example:

```
public class MyImplementation implements MyInterface {

	public void printMessage(String message) {
		System.out.println(message);
	}
}
```

Used properly, interfaces help to improve the modularity of code, by maintaining a separation between the *externally visible contract* met by a piece of code, and its particular, internal, implementation. In fact, several different classes may implement *the same* interface, using quite different approaches.

Remote Procedure Call is a particularly literal example of the separation of the external interface and its implementation. In this case, the application that *uses* an interface may be on a completely different processor to the implementation code. Java RMI builds on the basic idea of a Java interface, developing it further to the idea of a *remote interface*. Here is an example of a remote interface:

```
import java.rmi.* ;

public interface MyRemoteInterface extends Remote {
	void printMessage(String message) throws RemoteException ;
}
```

Any remote interface has two characteristics - it *extends* a special interface called `Remote` (found in the standard package `java.rmi`), and all its methods must be declared to throw an exception called `RemoteException`. Apart from this, a remote interface behaves like - in fact is - an ordinary Java interface.

This is as much as you need to know about interfaces and remote interfaces to use

them. If you are not very familiar with the Java exception system, don't get unduly distracted by the possibility of "remote exceptions". They just allow for the fact that something *might* go unexpectedly wrong in the complicated process of offloading a computation to a remote host. But for the moment we don't need to worry about the details of how that works.

## Experiment 1 - Using Java RMI

To run the examples in this lab you will need at least two computers - a client machine and a server machine. I suggest you organize yourselves into pairs, with one student initially responsible for the server side and the other initially responsible for the client side.

### The Remote Interface

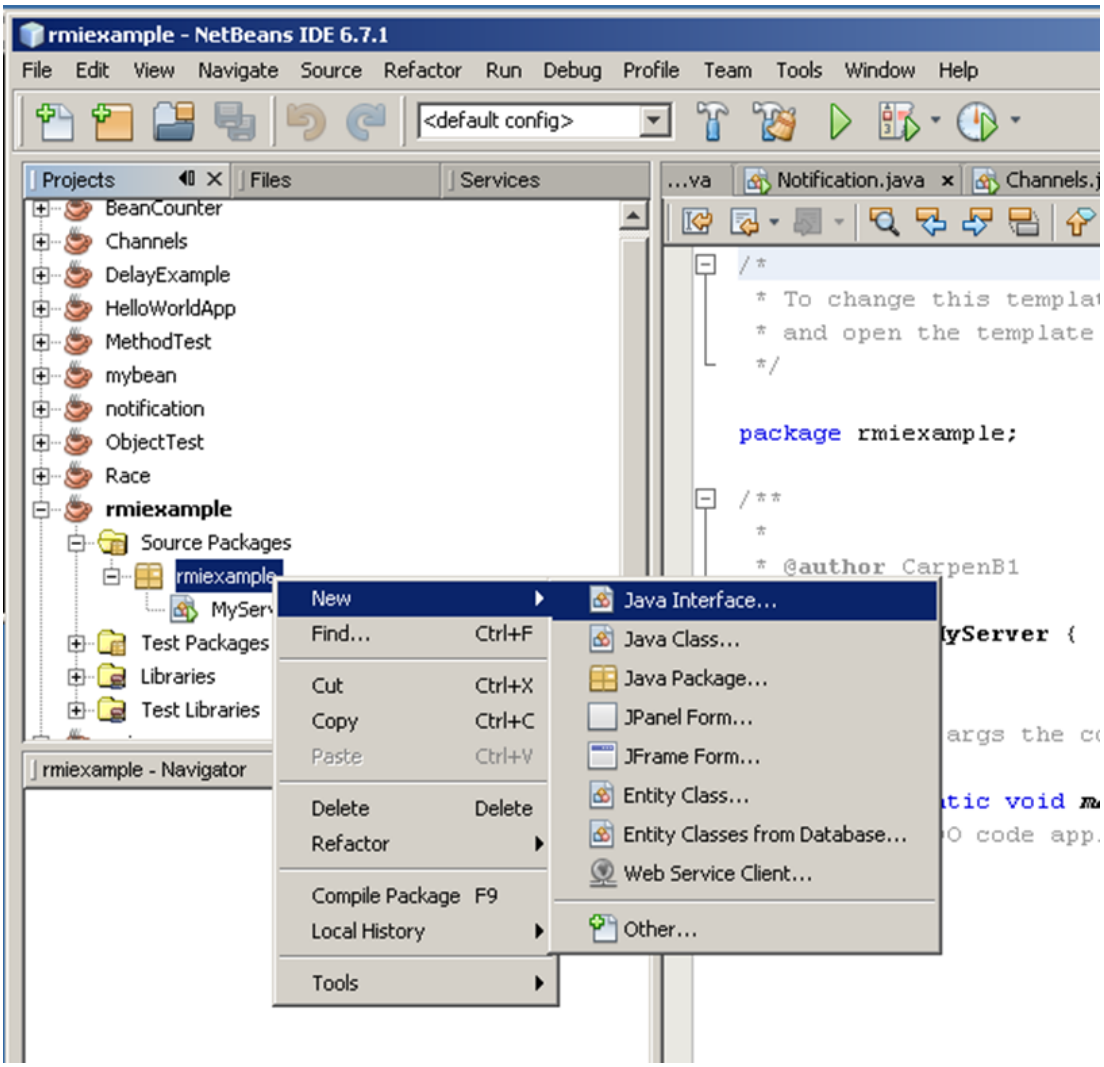
As usual client and server will each start by creating a new NetBeans project. Client and server machines should both create a project with the same Java *package name*. For example, the main class on the server might be `rmiexample.MyServer`, and on the client it might be `rmiexample.MyClient`.

The reason it is important to use the same package name on both computers is that the remote interface should have exactly the same fully qualified name - e.g. `rmiexample.MyRemoteInterface`. If the interface names on the two computers differ even slightly - for example in their package name - the examples below will fail!

Both client and server should create an *identical* remote interface in the same package as the main class. So for example you may right-click on the package folder under "Source Packages", and follow:

```
New -> Java Interface
```

as illustrated below:




Create an interface in the same package called `MyRemoteInterface`. We will use the definition of the remote interface given in the previous section:

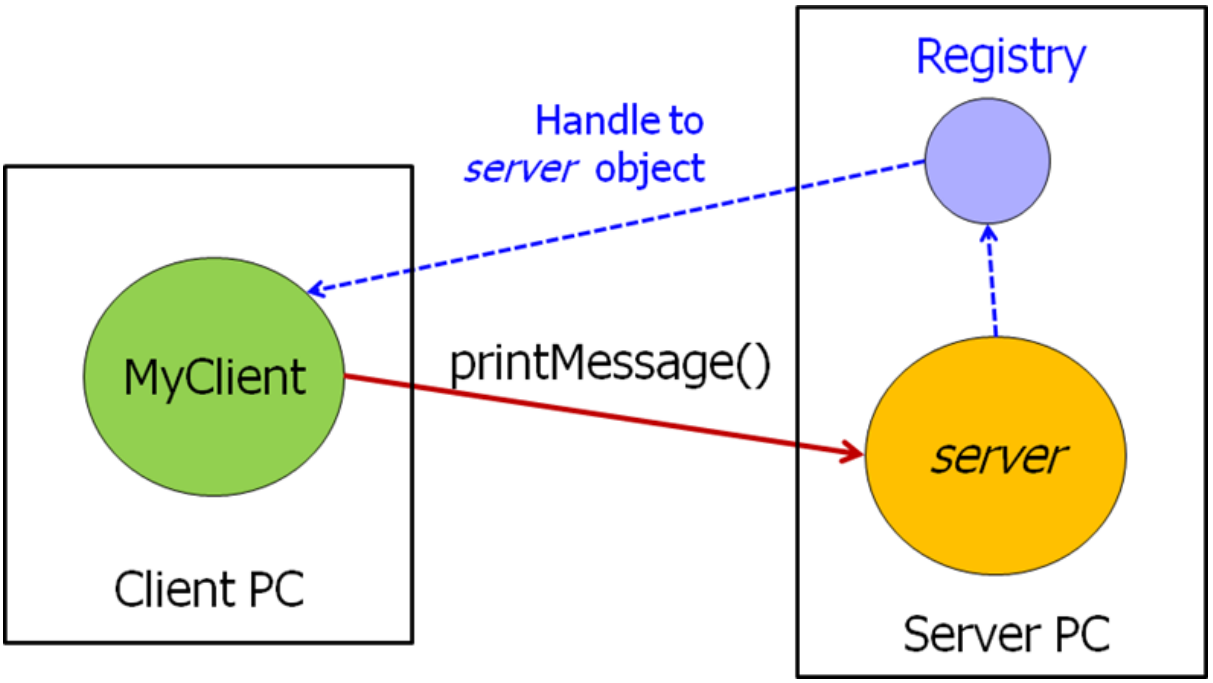
```
import java.rmi.* ;

public interface MyRemoteInterface extends Remote {

      void printMessage(String message) throws RemoteException ;
}
```

### Server Code

For reference, the interactions between the various components described in this subsection and the next are illustrated below:




Now the main class on the server machine can look like this:

```
   import java.rmi.*;

   import java.rmi.server.*;

   import java.rmi.registry.*;


public class MyServer extends UnicastRemoteObject
                  implements MyRemoteInterface {


       public void printMessage(String message) throws RemoteException {
              System.out.println(message);
       }


       public static void main(String args []) throws Exception {

              MyServer server = new MyServer();
              Registry reg = LocateRegistry.createRegistry(1234);
              reg.bind("myrmiserver", server);
       }


       public MyServer() throws RemoteException {}
}
```

The most important part of this class is the definition of `printMessage`, which is the method that will be invoked remotely. It just prints out the message passed to it as an argument. Before we can invoke this method we have to deal with a couple of

administrative issues.

The first thing, the main method does is creating an object of type `MyServer`. The client, on another host, somehow needs to be able to "find" that object before it can call a method on it. The easiest way to enable that in Java RMI is to use a "registry" for remote objects.

The penultimate line of the main method creates a local registry running on port 1234. You can choose any port number (above 1024), but the client will need to know this number. The final line adds the object `server` to the registry, giving it the name "myrmiserver". Again, you can choose any name you like, but the client will need to know it.

There is one more declaration in the `MyServer` class. This is a *constructor declaration*, needed because `MyServer` extends `UnicastRemoteObject`. This is the simplest way of creating remote implementation classes. Similar constructor declarations will be needed for any remote implementation class implemented this way - just accept this for now.

Now you simply run the main class, and wait for the client to do its business.

(If you have to restart your server program at any stage, it may be necessary to restart NetBeans. In particular, you may need to do this if you get a message saying "Port 1234 is already in use.")

## Client Code

Meanwhile, *on the client machine,* the main class may look (something) like this:

```
import java.rmi.registry.*;
public class MyClient {

	public static void main(String [] args) throws Exception {

		Registry reg = LocateRegistry.getRegistry("148.197.55.175", 1234);
		MyRemoteInterface handle =
			(MyRemoteInterface) reg.lookup("myrmiserver");

			handle.printMessage("Wassup!");
	}
}
```

The main business is the last line the `main` method, where it calls the `printMessage` method on... something. We hope that something is a local "handle" to the remote object, `server`, created on the server.

The first two lines acquire this handle. The way they do that is by going to the registry running on the server machine. We recognize "1234" as the port number for that registry. But to locate that registry we need to identify the host where it is running - i.e. on the server machine. *Change the first argument of* `getRegistry` *to contain the IP address of the server host.*

One way to find a workstation's IP address is by starting a Command Prompt (`cmd`) and within that running the command:

```
ipconfig
```

Of course, you must run this on the *server* workstation. Another approach is just to Google something like "what is my IP" on the server machine.

Now run the client program. You can run it several times. Each time a message should be printed by the application running on the server.

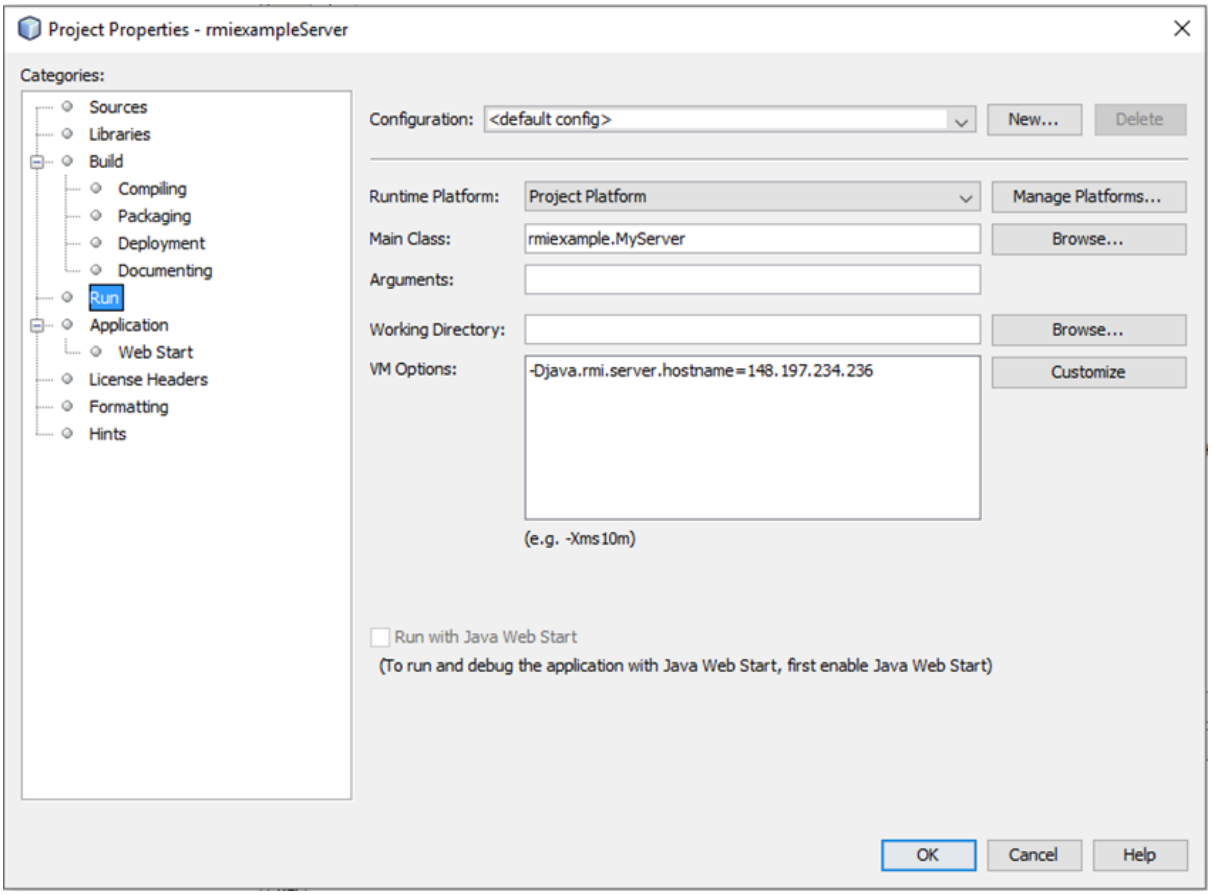
(There are a few ways this could fail. But *if* the client fails with an exception starting like this:

```
java.rmi.ConnectException: Connection refused to host: 192.168.56.1
```

It probably reflects an unusual network setup in some of our teaching labs, which means the server fails to pick up the correct value of its *own* IP address.

For a workaround, right-click on the NetBeans project that is being used on the server side, and select Properties. Go to the Run screen and edit the VM Options field so it may look similar to the following picture, with the IP address below replaced with the proper IP address of this machine - the server:




You of course still need to put the same IP address in the client program.

When you have this working, swap roles between the two computers. For example add a server class to the project on the client machine and vice versa, and change the main class for the projects. (You may do this by right-clicking on the project folder, selecting "Properties", then "Run", and editting "Main Class".)

## Reflective Questions

*Discuss what happened here and relate it to the general description of remote procedure call in the week 6 lecture on Inter-Process Communication; discuss the handling of procedure (or method) arguments.*

### Exercises

1. Adapt your class `MyClient` to read a line typed at the keyboard, and send that to the server in place of the fixed string above.

Example code for reading a line from standard input in Java:

```
Import java.io.*;

[...]
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String message = in.readLine(); [...]
```

2. Put the code in the client for reading and sending message in the client in a while loop, so you can send a whole stream of messages to the server.

3. Try to extend your program for *two-way* chat between two workstations.

   The simplest approach is just to run a copy of `MyServer` and `MyClient` on both workstations.

   If you are more ambitious, you may wish to merge the functionality of both into a single program. Take the code from the `main` method of your extended client, and append it at the end of the main method of `MyServer`.

   A problem you will probably encounter is that now starting the registry on one PC is immediately followed by connecting to the registry on the other. Unless the programs start at exactly the same time at both ends, one of the client connections will fail because the service hasn't started yet on the other.

   A crude solution is to put a delay in the program between `bind` and `getRegistry`, and start the programs on the two hosts at approximately the same time - something like.

```
Thread.currentThread().sleep(30*1000);  //30seconds
System.out.println("attempting connection");
```

   Can you think of better solutions?