

Remote Method Invocation (RMI) in Java

- For true concurrency, programs need to be able to execute across multiple processors.
- One way of addressing this is through *remote procedure calls* or RPCs in Ada.
- Java uses a similar idea, remote method invocation or RMI.

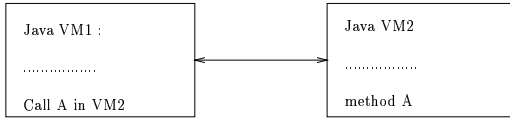


Figure 1: The scenario for an RMI

When you execute a Java program :

- It runs in the context of a Virtual Machine (the Java interpreter),
- Objects created by the program are accessed using references.
- References are meaningful only in the context of the local address space of the JVM executing the program.

1

RMI Issues

- So how can you call **method A** of an object in JVM2 from a program executing in JVM1?
- The address spaces are different and unrelated.

There are really two issues here :

- How can you expose the services a (server) program provides (e.g., JVM2 above) so that they can be used from a different JVM (e.g., JVM1)?
- Then how do you handle the problem of accessing object references across different JVM address spaces?

What has RMI to do with concurrent programming?

- Well, with RMI, you now have the opportunity to execute methods on remote processors and so implement true concurrency.
- Of course, your thread on the remote processor is subject to the workload of that processor. But your threads on each processor will be executing in parallel.

2

RMI Issues

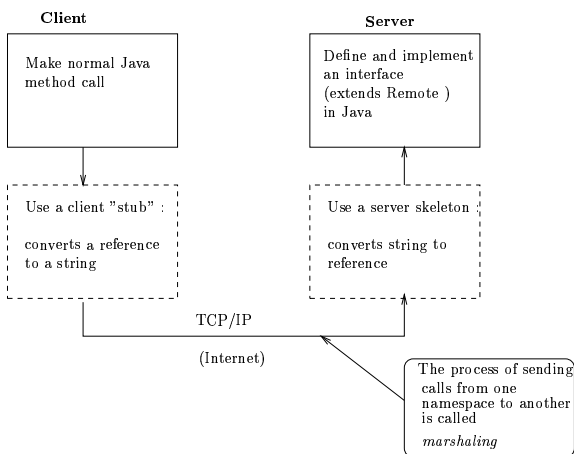


Figure 2: The scenario for an RMI

Steps :

1. Compile the Java client and server programs in the normal way.
2. Generate the necessary client *stub* (or *proxy*) and server *skeleton* using the Java RMI compiler (*rmic*).

3

RMI Issues

- **rmic** takes the client class files as arguments and uses these to find out which methods are defined to be remote (in the server interface) and constructs *stub* and *skeleton* code for them for remote marshaling.
- Java RMI exposes objects whose methods can be remotely accessed.
- Objects are class *instantiations* and the methods that are available for remote access are defined in a Java *interface* which extends Java's **Remote** class.
- The class *implements* the interface and extends the Java RMI class. The **UnicastRemoteObject** class provides the methods necessary for supporting remote services.

4

Architecture of RMI

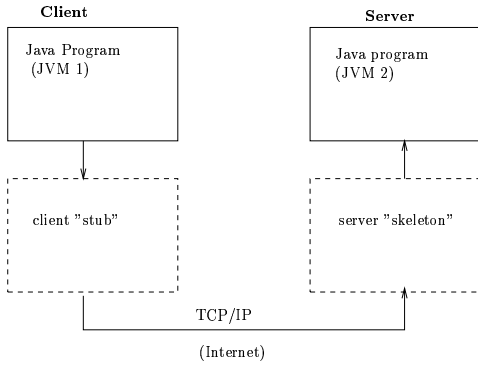


Figure 3: Architecture of RMI

5

Code Outline for the Server

First, the interface :

```
public interface Hello extends java.rmi.Remote{
    String sayHello() throws java.rmi.RemoteException;
}
```

Then the server code :

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class HelloImplementation
    extends UnicastRemoteObject // required for RMI
    implements Hello {
    private String name;
    public HelloImplementation(String s)
        throws RemoteException{
        super(); name=s;
    }
    public String sayHello()
        throws RemoteException {
        return ("This is a distributed Hello remote
            method invocation test");
    }
}
//code for main()-constructs and registers server object
}
```

6

Code Outline for the Client

The client finds the remote object using a look-up service provided by RMI. It can then access server methods defined by the server interface.

```
import java.rmi.*;

public class RemoteHelloClient{

void InvokeRemoteMethod(){
    String message="";
    try{

        //locate registry; get remote object reference
        //associated with name

Hello obj =
    (Hello)Naming.lookup("//pc-114.cs.uwa.edu.au/HelloServer");
        //you can use URL here
        //You have to pc-47 by the
        //name of the machine you
        //are currently working
        //Invoke the remote method on this object
        message = obj.sayHello();
    } catch (Exception e) {
```

7

```
        System.out.println("RemoteHelloClient exception" +
            e.getMessage());
        e.printStackTrace();
    }
    System.out.println(message);
}
```

8

Deploying the Implementation

- Compile Java source files,
- Generate stubs (client proxies) and skeletons (server interface) using the `rmic` compiler on the class files that contain remote object implementations.
- Start the RMI bootstrap registry - this provides support for constructing a "catalogue" of object names *vs* references that are available on the server.
- Start the server. The server registers itself in the registry.
- Start the client. The client locates the registry on the server and uses it to get an object reference which can then be used to call the remote method(s).

9

Start the Client

The client :

- Locates the registry on the server,
- uses it to get an object reference,
- uses this object reference to call the remote method

```
public static void main(String args[]){
    System.setSecurityManager(new RMISecurityManager());
    RemoteHelloClient rhc = new RemoteHelloClient();
    rhc.InvokeRemoteMethod();
}
```

11

The Server Registers Itself in the Registry

The `main()` method for the server :

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

//Create and install a security manager
System.setSecurityManager(new RMISecurityManager());

try{
    //Construct a server object and give it a name
    HelloImplementation obj =
        new HelloImplementation("HelloServer");
    //Register the server object in the registry
    Naming.bind("HelloServer", obj);
    System.out.println("HelloServer bound in registry");
} catch(Exception e) {
    System.out.println("HelloImplementation error"
        + e.getMessage());
    e.printStackTrace();
}
}
```

10

RMI Summary

- RMI allows a Java program executing in one JVM environment to invoke methods in a different JVM.
- Objects that have methods that can be remotely invoked must be registered in a server Remote Object registry.
- The client consults this registry to obtain remote object references.
- The basic steps are :
 - define a remote interface
 - write an implementation class
 - write a client that uses the remote interface
 - compile the files
 - start the registry, server and client

12