

# Understanding Java RMI Internals

By [Ahamed Aslam.K](#)

January 6, 2005

## Motivation for this Article

The motivation to write this article is my own experience. I came to know about Java RMI only recently. Instantly, I liked the concept very much, especially that of stubs and skeletons. It seemed very interesting to me how RMI designers have designed the whole framework such that the RMI client feels as if the method is invoked locally while it is actually executed in a remote object. As I learned more, I came to know about the RMIRRegistry and more. There began the train of problems. I had many doubts, such as what is the actual role of RMIRRegistry and is it absolutely necessary? Where is the object of the stub created (in the server, client, or RMIRRegistry)? How will the client know to which port the server is listening? Which port is the server listening to? Are all the servers using port 1099 for listening to clients and so on.

This article is really an attempt to answer my own questions that I had previously. I referred to a lot of books and articles on RMI on the net, but I was unable to get any clue on the matter. Nobody told me how things are happening. Everyone was telling me how to program—only that, and nothing about how things really work in the lower levels. I got answers to my questions after a long time after doing a lot of research and experimenting. So, I decided I should share my knowledge to others because many guys might be having the same doubts.

## Introduction

This article tries to answer the questions about RMI Internals, such as the following:

1. Who actually creates an object of stubs? The server, Registry, or client?
2. Which port is the server listening to?
3. Does the server listens to port 1099 (default port of RMI Registry)?
4. How will the client know to which port the server is listening
5. Is a Registry necessary to run the RMI system?
6. Can you use RMI without rmiregistry?

I will answer all these questions in detail in the following sections. If you want quick answers to these questions, you can directly go to the last part. Here, I will go in a step-by-step manner to give you an idea about what is happening in RMI.

## Forget About RMIRRegistry (for now)!!!

Yes, just forget about the RMIRRegistry for now. Just assume such a thing does not exist as of now.

The initial scenario is like this: We have a server and a client. The server extends from `java.rmi.server.UnicastRemoteObject`. The client and server are running on different machines.

Now, our requirement is this: The client wants to execute a function of the server that is on a remote machine.

How that can be done? The Java RMI framework deals with this question. The solution certainly involves network programming using sockets because the server is running on a remote machine. The point is that solution to this problem should focus on a framework in which the client is decoupled from the networking programming code. Also, you should build the framework in such a way that the client should not be aware that the the function it is calling is actually run in a remote machine. It should appear as if the function is run locally. So, the RMI framework developers introduced the Stub and skeleton model (I assume you already know about stubs and skeletons).

What happens is that all the network-related code is put in the stub and skeleton so that the client and server won't have to deal with the network and sockets in their code. The stub implements the same Remote interface (as in an interface that extends `java.rmi.Remote`) that the server implements. So, the client can call the same methods in the stub that it wants to call in this server. But, the functions in the stub are filled with network-related code, not the actual implementation of the required function. For example, if a server implements an `add(int,int)` function, the stub also will have an `add(int,int)` function, but it won't contain the actual implementation of the addition function; instead, it will contain the code to connect to the remote skeleton, to send details about the function to be invoked, to send the parameters, and to get the results back.

So, this will be the situation:

```
Client<--->stub<--->[NETWORK]<--->skeleton<--->Server
```

The client speaks to the stub, the stub speaks to the skeleton through the network, the skeleton speaks to the server, the server executes the required function, and the results are also passed in the same way. So, you will have four separate programs corresponding to each of these entities (this means four class files).

Later, in JDK 1.2, the skeletons were incorporated into the server itself so that there are no separate entities as skeletons. In other words, the skeletons's functionality was incorporated into the server itself. So, the scenario became like this:

```
Client<--->stub<--->[NETWORK]<--->Server_with_skeleton_functionality
```

## Socket-Level Details

Now, you can examine how the communication is achieved in the socket level. This is *very* important. This is where the concept usually gets twisted. So, take special note about this section.

1. The server listens to a port on the server machine. This port is usually an anonymous port that is chosen at runtime by the jvm or the underlying operating system. Or, you can say that the server exports itself to a port on the server machine.
2. The client has *NO* idea on which machine and to which port the server is listening. But, it has a stub object that knows all these. So, the client can invoke the desired method of the stub.
3. The client invokes the function of the stub.
4. The stub connects to the server's listening port and sends the parameters. The details of this are given below. You can skip it if you don't want to know about the intricate details about TCP/IP connection semantics. For those who are interested, this is how it is done.

1. The client connects to the server's listening port.
2. The server accepts the incoming connection and creates a new socket just to handle this *single* connection.
3. The old listening port is still there; it waits for the incoming requests from the clients.
4. The communication between client and server takes place using the newly created socket on the server.
5. With an agreed-upon protocol, they communicate and exchange parameters and results.
6. The protocol can be JRMP (Java Remote Method protocol) or CORBA-compatible RMI-IIOP (Internet Inter-ORB Protocol).
5. The method is executed on the server and the result is sent back to the stub.
6. The stub returns the results back to the client as if the stub had executed the function locally.

So, that is how it is done. Wait a second. Take a look at Point 2. The stub knows which host the server is running and to which port the server is listening. How is that possible?? If the client does not know about the server host and port, how can it create an object of the stub that knows all these? Especially when the server port is chosen *arbitrarily* or randomly when the server is instantiated, it will change for each object of the same server class. Once the client knows these details, it can proceed. Also, it should be noted that even if the client has CalcImpl\_Stub.class in its machine, it cannot simply create an object of the stub because its constructor takes a RemoteRef reference as a parameter and you can get that only from an object of the remote server—exactly what we are trying to access! This is called a *Bootstrap Problem*.

## Bootstrap Problem!!!

This is one of the biggest problems in many real-life systems. The solution depends on how we can inform the client about the details of the server. RMI designers have a work-around for the bootstrap problem. That is where the RMIRRegistry comes in. The Registry can be thought of as a service that keeps (public\_name, Stub\_object) pairs in a hashmap. For example, if I have a Remote Server object called Scientific\_Calculator, I can make it available by a public name, "calc". For this, I will create a Stub Object at the server machine and register it with the RMIRRegistry so that clients who want to access the services of the Remote Server object can get the Stub Object from the Registry. For doing these things, you use a class called java.rmi.Naming.

Let me illustrate this with an example.

Consider a calculator application. It has an add(int,int) function. You want to make this function available to remote clients on other machines. We have the Remote interface called Calc.

### Source code of Calc.java:

```
public interface Calc extends Remote
{
    public int add(int i,int j)throws RemoteException;
}
```

We have the server class CalcImpl.

## Source code of CalcImpl.java:

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class CalcImpl extends UnicastRemoteObject implements Calc
{
    public CalcImpl()throws RemoteException
    {
        super();
    }

    public int add(int i,int j)throws RemoteException
    {
        return i+j;
    }

    public static void main(String[] args)throws Exception
    {
        CalcImpl c=new CalcImpl();
        Naming.rebind("rmi://localhost:1099/calc",c);
    }
}
```

We have the client class CalcClient.

```
import java.rmi.Naming;

public class CalcClient
{
    public static void main(String[] args)throws Exception
    {
        Calc c=(Calc)Naming.lookup("rmi://remotehost:1099/calc");
        System.out.println(c.add(3,4));
    }
}
```

**Note:** You have to replace *remotehost* with the host name or IP address of the server machine.

These are the classes. Compile all the java files to get the class files. Now, use the RMI compiler to generate the stub. Use the .keep option if you need the source file of the stub.

```
C:\test>rmic .keep CalcImpl
```

This will create two files: CalcImpl\_Stub.class and CalcImpl\_Stub.java.

## Source code for CalcImpl\_Stub.java:

```
public final class CalcImpl_Stub
    extends java.rmi.server.RemoteStub
    implements Calc, java.rmi.Remote
{
    private static final long serialVersionUID = 2;
    private static java.lang.reflect.Method $method_add_0;
    static {
        try {
            $method_add_0 = Calc.class.getMethod("add",
                new java.lang.Class[] {int.class, int.class});
        } catch (java.lang.NoSuchMethodException e) {
            throw new java.lang.NoSuchMethodError(
                "stub class initialization failed");
        }
    }

    // constructors
    public CalcImpl_Stub(java.rmi.server.RemoteRef ref) {
        super(ref);
    }

    // methods from remote interfaces

    // implementation of add(int, int)
    public int add(int $param_int_1, int $param_int_2)
        throws java.rmi.RemoteException

    {
        try {
            Object $result = ref.invoke(this, $method_add_0,
                new java.lang.Object[]
                {new java.lang.Integer($param_int_1),
                 new java.lang.Integer($param_int_2)},
                -7734458262622125146L);
            return ((java.lang.Integer) $result).intValue();
        } catch (java.lang.RuntimeException e) {
            throw e;
        } catch (java.rmi.RemoteException e) {
            throw e;
        } catch (java.lang.Exception e) {
            throw new java.rmi.UnexpectedException("undeclared checked
                exception", e);
        }
    }
}
```

Right now, just don't worry much about what it does; just have a look at the constructor, which you might need later.

Now to run this code on the server machine, we need two consoles.

On console 1, run

```
C:\test>rmiregistry
```

On console 2, run

```
C:\test>java CalcImpl
```

On the client machine, run the following on a console,

```
C:\test>java CalcClient
```

You will get an output of 7 on your screen.

<http://www.developer.com/java/ent/print.php/3455311>

## **What Is Happening Here**

I'll show you what is really happening behind the scenes.

1. First, RMIRRegistry is run on the server machine. RMIRRegistry itself is a remote object. The point to note here is this: All remote objects (in other words, all objects that extend UnicastRemoteObject) export themselves to an arbitrary port on the server machine.

Because RMIRegistry is also a remote object, it exports itself into a port. A difference is that the the port is a well-known port. By default, it is 1099. The term well-known means that the port is known to all the clients.

2. Now, the server is run on the server machine. In UnicastRemoteObject's constructor, it exports itself to an anonymous port on the server machine. This port is unknown to the clients; only the server knows about this port.
3. When you call Naming.rebind(), you are passing a reference of CalcImpl (here c) as the second parameter to the Naming class. The Naming class constructs an object of the Stub and binds it past the stub object (not the actual object) to the Registry remote object. How is the stub object created? Here it is:
  1. The naming class uses the getClass() method to get the name of the class (here, CalcImpl).
  2. Then, it adds \_Stub to the name of the class to get the stub's name (here, CalcImpl\_Stub).
  3. It loads the CalcImpl\_Stub class to the JVM.
  4. It gets a RemoteRef obj from c.

```
RemoteRef ref=c.getRef();
```

It is this ref that encapsulate all the details about the server such as server hostname, server's listening port, address, and so on.

5. It uses this RemoteRef object to construct the stub:

```
CalcImpl_Stub stub=new CalcImpl_Stub(ref);
```

If you look at CalcImpl\_Stub.java, you will find that the stub's constructor takes RemoteRef reference as the parameter.

6. It passes this stub object to RMIRegistry for binding, along with the public name as (calc,stub).
7. RMIRegistry stores this public name and stub object in a hashmap internally.
4. When the client executes Naming.lookup(), it passes the public name as the parameter. The RMIRegistry (which itself is a remote object) returns the stored stub object back to the client. Now, the client gets a stub object that knows about the server host name and port to which the server listens. The client can invoke the stub's method to call the remote object's method.

## **Is RMIRegistry Absolutely Necessary? Can I Do Away With RMIRegistry and Bind/Rebind/Lookup Methods?**

That is how it works. Is RMIRegistry an absolute necessity? From the above discussion, it is evident that an RMIRegistry is *not* a must. RMIRegistry is used just for the bootstrapping purpose, nothing else. Once the client gets the stub object from the Registry, the Registry is not used anymore. The communication takes place directly between client and server by using sockets. So, if you can provide the bootstrapping in some other way, you can eliminate the use of Registry itself. In other words, if the client

can get an object of the stub from the server somehow, that is just enough. That means if you have some facility to transport the stub object created in the server to the client machine, you don't have to use the RMIRegistry or bin/rebind/lookup methods. That sounds great. eh? Also, it should be noted that even if the client has CalcImpl\_Stub.class on its machine, it cannot simply create an object of the stub because its constructor takes a RemoteRef reference as a parameter; you can get only from an object of the remote server, which is exactly what we are trying to access! To facilitate the transportation of the stub object from server to client, you can build a transport facility of your own that is based on sockets that serialize the Stub object and sends it to the clients. That will eliminate the need of the Registry at all!

I will give you an example that does not use the RMIRegistry. But, in this example, the client and server run on the same machine. You can use socket programming as I stated earlier if the client and server are on different machines, but the program will make the basic concepts clear.

### Source code of CalcImpl.java:

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class CalcImpl extends UnicastRemoteObject implements Calc
{
    public CalcImpl()throws RemoteException
    {
        super();
    }

    public int add(int i,int j)throws RemoteException
    {
        return i+j;
    }

    public static void main(String[] args)throws Exception
    {
        CalcImpl ci=new CalcImpl();
        RemoteRef ref=ci.getRef();
        CalcClient cc=new CalcClient(ref);
    }
}
```

### Source code of CalcClient.java:

```
import java.rmi.server.*;
import java.rmi.*;

public class CalcClient
{
    public CalcClient(RemoteRef ref)throws RemoteException
    {
        CalcImpl_Stub stub=new CalcImpl_Stub(ref);
        System.out.println(stub.add(3,4));
    }
}
```



compile the classes and run CalcImpl and CalcClient on two different consoles.

On console 1:

```
C:\test2>java CalcImpl
```

On console 2:

```
C:\test2>java CalcClient
```

You will get an output of 7 on your screen.

That is it!!!

## Questions Revisited

Question	Answer
Who actually creates an object of stubs? The server, Registry, or client?	A stub is created by the Naming class at the server. For getting the stub class, it uses the Java Reflection feature (see above).
Which port is the server listening to?	The port that the remote server is listening to is usually chosen arbitrarily at runtime and is decided by the JVM or underlying operating system.
Does the server listen to port 1099 (the default port of the RMI Registry)?	The server usually does not listen to port 1099. Port 1099 is usually used by a special Remote object called RMIRRegistry. The port that is the server's listening port is chosen arbitrarily at runtime (see above).
How will the client know to which port the server is listening?	The client does <i>*not*</i> know which port the server is listening to, but it has a reference to an object of the stub that <i>does</i> know which port the server is listening to.
Is a Registry necessary for running the RMI system?	The Registry is used for bootstrapping purposes only. If you have some other means for transporting the stub object created at the server machine to the client, you can do away without the Registry (see above).
Can we use RMI without rmiregistry?	Same as the answer to the third question.