Understanding Java RMI Internals

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 RMIRegistry and more. There began the train of problems. I had many doubts, such as what is the actual role of RMIRegistry and is it absolutely necessary? Where is the object of the stub created (in the server, client, or RMIRegistry)? 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 RMIRegistry (for now)!!!

Yes, just forget about the RMIRegistry 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
```

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 RMIRegistry 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 RMIRegistry 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.