© 2014 IJIRT | Volume 1 Issue 5 | ISSN : 2349-6002

Java Remote Method Invocation (RMI)

Sheena Batra, Rakesh Sondal
Dronacharya College of Engineering, Guragon

***Abstract*- RMI helps access the objects residing on one machine to access the methods of that object. When we invoke the methods of that object, method gets executed and the result is sent to the method caller. In RMI, the object whose method make remote call is termed as the client object and the remote object itself is called as server object. The computer running the code that calls remote method is client for that object and the hosting computer of object is server for that call. A remote lives on a server. Each remote object implements a remote interface that specifies which of its methods can be invoked by the clients. Clients invoke methods of remote objects as that of its local methods. This concept of remote hosts invoking raises various security issues. Thus remote objects can perform limited functions. The main aim of this paper is to find an ultimate solution of these security issues and enhance the concept of RMI.**

I. INTRODUCTION

RMI is the action of invoking a method of a remote interface on a remote object. If you have an access to an object on a different machine, you can call methods of the remote object. of course, the method parameters must somehow be shipped to the other machine, the server must be informed to execute the method & the return value must be shipped back. In RMI, the object whose methods makes the remote call is called the **client object.** The remote object is called the **server object**. The computer running the java code that calls the remote method is the **client** for that call. The computer hosting the object that processes the call is the **server** for that call. In this way, the object-oriented paradigm is preserved in distributed computing. There are several implementation architectures for the remote method invocation. The most well-known one is CORBA which has several commercial implementations available. However, with the success of the Java language, Java RMI is earning more and more attention. In contrast to CORBA that is programming language independent, Java RMI only works between Java programs. On the other hand, Java RMI is far more flexible than CORBA. Java RMI is gaining popularity, it is only a matter of time before the performance of Java RMI over wireless links becomes important. We have analyzed and measured performance characteristics of Java RMI over GSM Data Service. The results are not encouraging. Java RMI works poorly in slow wireless environments.

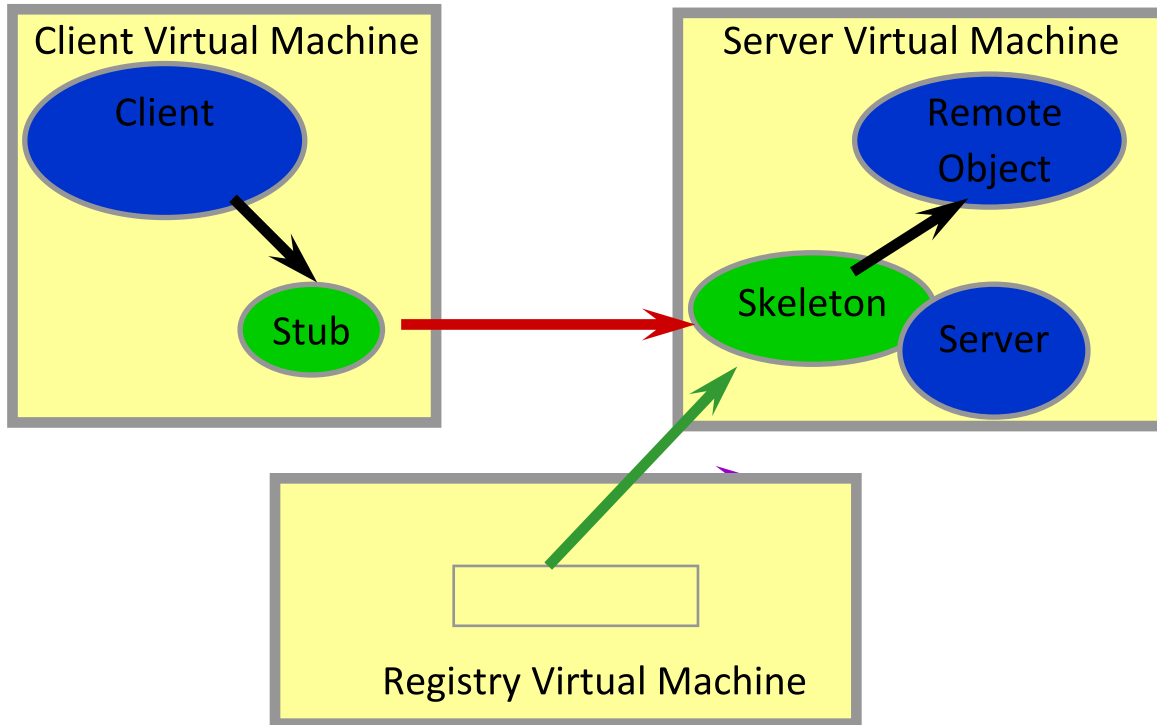
Registration (binding)- A server can register its remote objects with a naming service – the rmiregistry. Once registered, each remote object has a unique URL.

Obtaining a remote object reference - A client can use the naming service to lookup a URL to obtain a remote object reference. The application can pass and return remote object references as part of its operation.

Stubs & Parameter Marshalling- When client code invokes a remote method on a remote object, it actually calls an ordinary method on a proxy object called a **stub**. The stub resides on the client machine. The stub packages the parameters used in the remote method into a block of bytes. This packaging uses a device independent encoding for each parameter. The process of encoding the parameters is called **parameter marshalling**. The Purpose of parameter marshalling is to convert the parameters into a format suitable for transport from one m/c to another.

Skeleton - A proxy object on server side is called Skeleton.

Dynamic Class Loading- When u pass a remote object to another program, either as a parameter or return value of a remote method, then that program must have the class file for that object.

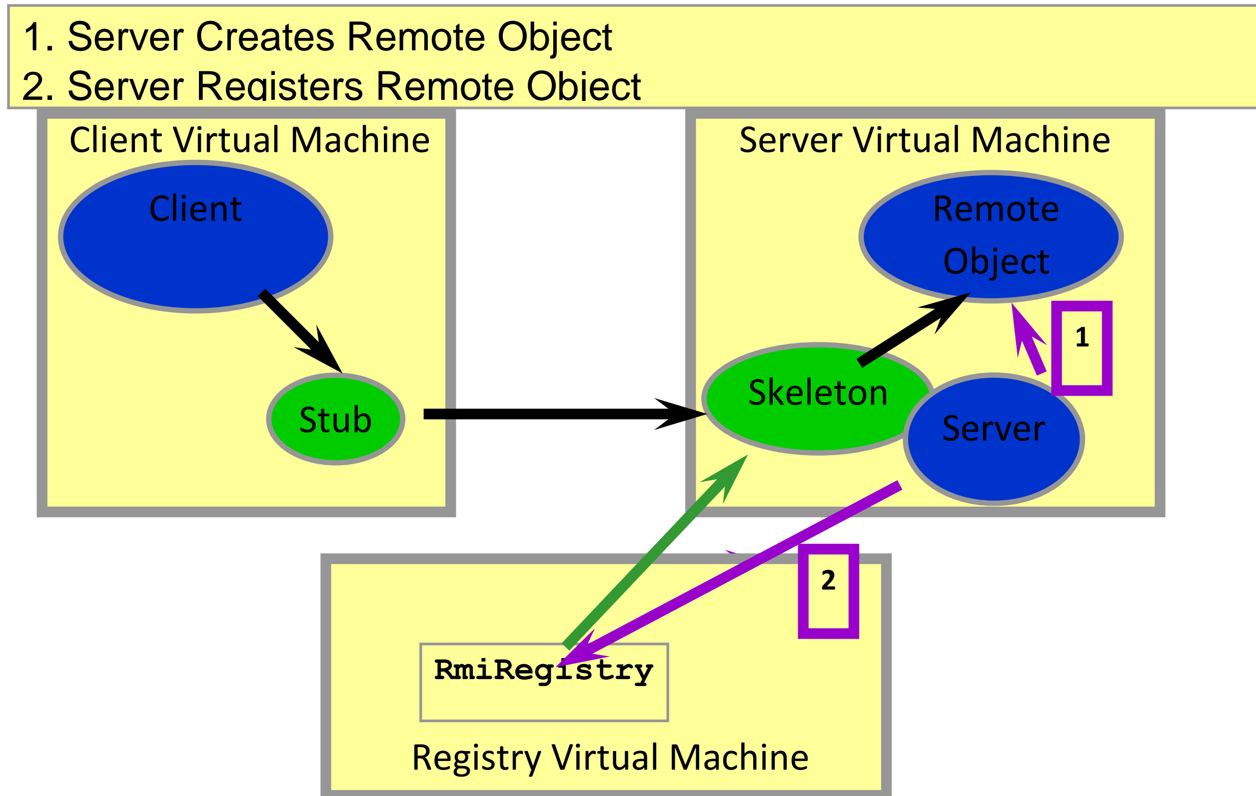


II. RMI DETAILED KNOWLEDGE

Java RMI was designed to simplify the communication between two objects in different virtual machines by allowing transparent calls to methods in remote virtual machines.Once a reference of a remote object is obtained,it is possible to call methods of that object in the same way as methods of local objects. Since the remote object resides in a different virtual machine, an RMI *Registry* is needed to manage remote references. When an RMI server wants to make its local methods available to remote objects, it registers the objects to a local registry. A remote object connects to the remote registry, which listens to a well-known socket,and obtains a remote reference. Java RMI is built on top of a *transport layer*, which provides abstract RMI connections built on top of TCP connections. When an RMI connection is opened, the transport layer either opens a new TCP connection, or reuses an existing one if a free one is available. If the reused connection has been idle for more than the time of a round-trip, the transport layer first sends a ping packet to make sure the connection is still working. Once an acknowledgment for the ping packet is received, the new RMI connection is established.

If a TCP connection has not been used by any RMI connections for a while, it is closed.

The general Java RMI architecture is depicted in Figure 1. First a server creates a remote object and registers it to a local Registry . The client then connects to the remote Registry and obtains the remote reference. At this point, a stub of the remote object is transferred from the remote virtual machine to the client virtual machine, if the stub is not yet present. When the client invokes a method at a remote object, the method is actually invoked at the local stub. The stub marshals the parameters and sends a message to the associated skeleton on the server side. The skeleton unmarshals the parameters and invokes the appropriate method. The remote object executes the method and passes the return value back to the skeleton, which marshals it and sends a message to the associated stub on the client side. Finally the stub unmarshals the return value and passes it to the client.

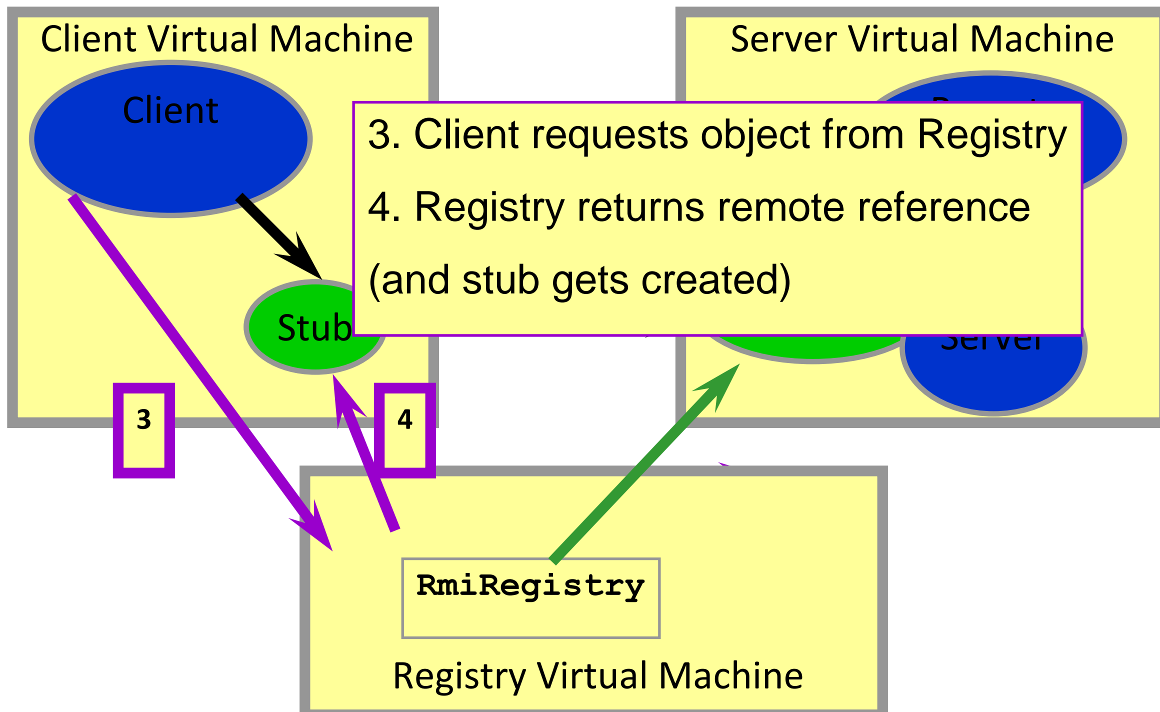


1. **Define the Remote Interface –**
Your client program needs to manipulate server objects,but it doesnot actually have copies of all of them.The objects themselves reside on the server.The client code must still know what it can do with those objects.Their capabilities are expressed in an interface that is shared between the client & server & so resides simulateneously on both machines.To define the remote service, we write a remote interface.All interfaces for remote objects must extend Remote interface defined in java.rmi package.

```
import java.rmi.*;

public interface Hello extends Remote

{

    public String sayHello() throws
RemoteException;

}
```

2. Implementing the Remote Interface- On the server side,you must implement the class that actually carries out the methods advertised in the remote

interface.Declare the remote interface being implemented. You can tell that the class is a server for remote methods because it extends UnicastRemoteObject,which is a concrete java class that makes objects remotely accessible. A UnicastRemoteObject object resides on a server.This is the class that we extend for all the server classes.



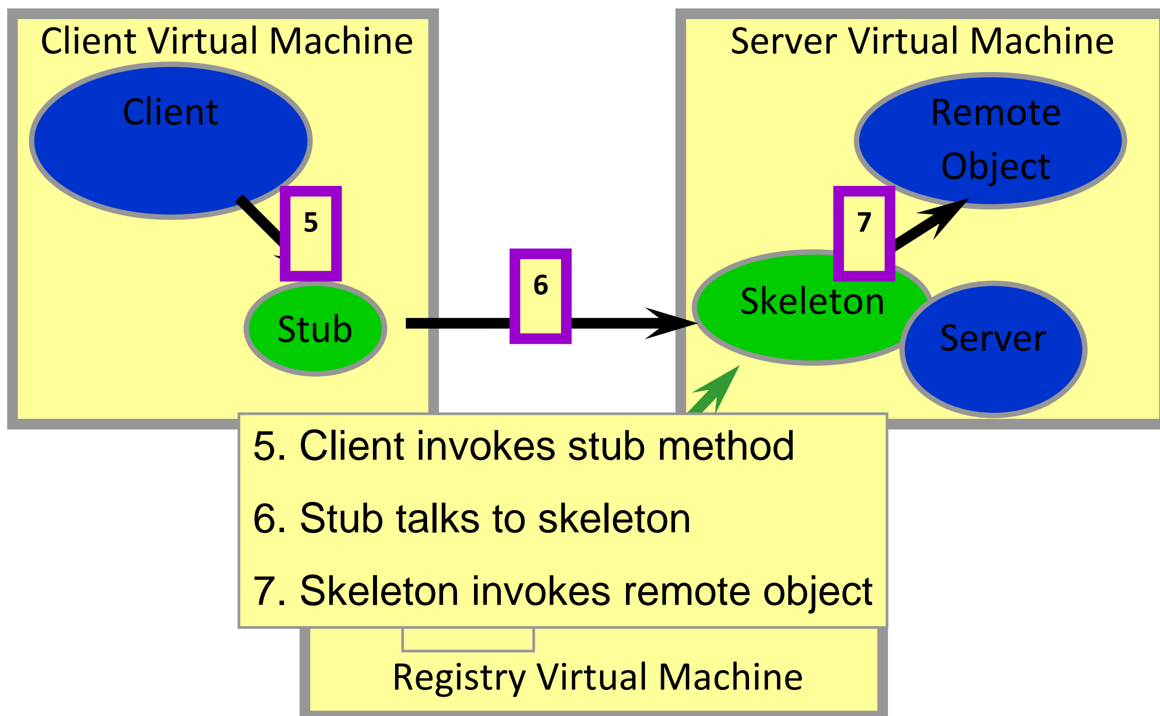
3 Locating Server Objects-To access a remote object that exists on the server,the client needs a local stub object.Then how can a client request such a stub? The most common method is to call a remote method of another server object & get a stub object as a return value. A server programs registers objects with the bootstrap registry service & the client retrieves stubs to those objects.You register a server object by giving the bootstrap registry service a reference to the object & a name.

RMI URLs start with rmi:// & are followed by a server, an optional port number, another slash, & the name of the remote object. Eg: rmi://localhost:99:port/central_warehouseFor security reasons,an application can bind,unbind or rebind registry object refrences only if it runs on the same host as the registry. Registry used for client bootstraping to get the initial reference.

4. Client program

```
import java.rmi.*;

public class HelloClient

{

public static void main(String args[])

{

try

{
```

```
Hello
h=(Hello)Naming.lookup("rmi://localhost/server");

System.out.println("client: Hello!");

System.out.println("server:" +h.sayHello());

}

catch(Exception e)

{

System.out.println("Error:"+e);

}

}

}
```



III. CONCLUSION

Due to its high protocol overhead, Java RMI is poorly suited for wireless communication. However, it can be optimized without breaking compatibility with Java RMI specification, and with minimal changes to existing software. New software is necessary only at the mobile terminal and at its access point to the fixed network. The results are encouraging. The new features will greatly improve the usability of Java RMI in a mobile environment.

REFRENCES

[1] A. Bakre and B. R. Badrinath. M-RPC: A Remote Procedure Call Service for Mobile Clients. In *Proc. of the ACM MobiCom'95*, pages 97–110, Berkeley, Calif., Nov. 1995.
[2] S. Microsystems. Java Remote Method Invocation – Distributed Computing for Java. White Paper, 1998.
[3] http://en.wikipedia.org/wiki