

Remote Method Invocation (RMI)

The Remote Method Invocation (RMI) is an API that provides a mechanism to create distributed application in java. The RMI system allows an object running in one Java Virtual Machine (VM) to invoke methods of an object running in another Java VM. RMI is used for client and server models.

RMI is the object oriented equivalent to Remote procedure call (RPC). RMI provides for remote communication between programs written in the Java programming language. The RMI provides remote communication between the applications using two objects stub and skeleton.

Distributed applications are applications or software that runs on multiple computers within a network at the same time and can be stored on servers or with cloud computing. Unlike traditional applications that run on a single system, distributed applications run on multiple systems simultaneously for a single task or job. Distributed apps can communicate with multiple servers or devices on the same network from any geographical location. The distributed nature of the applications refers to data being spread out over more than one computer in a network.

Remote versus Local Objects

Although we would like remote objects to behave exactly the same as local object, that is impossible for several reasons:

- ✓ Networks can be unreliable.
- ✓ References on one machine (memory addresses) have no meaning on another machine.

Consequences

- Special exceptions will indicate network failures.
- Objects passed as parameters and returned as results are passed by copy mode (pass by value).

Reasons for Using RMI

- ✓ Some operations can be executed significantly faster on the remote system than the local client computer.
- ✓ Specialized data or operations are available on the remote system that cannot be replicated easily on the local system, for example, database access.
- ✓ Simpler than programming our own TCP sockets and protocols

Stub:

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

- ✓ It initiates a connection with remote Virtual Machine (JVM),
- ✓ It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
- ✓ It waits for the result
- ✓ It reads (unmarshals) the return value or exception, and
- ✓ It finally, returns the value to the caller.

Skeleton:

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

- ✓ It reads the parameter for the remote method.
- ✓ It invokes the method on the actual remote object.
- ✓ It writes and transmits (marshals) the result to the caller.

RMI Architecture:

The server is the application that provides remotely accessible objects, while the client is any remote application that communicates with these server objects. Description of the architecture:

1. The client uses the client-side stub to make a request of the remote object. The server object receives this request from a server-side object skeleton.
2. A client initiates an RMI invocation by calling a method on a stub object. The stub maintains an internal reference to the remote object it represents and forwards the method invocation request through the remote reference layer(RRL) by *marshaling* the method arguments into serialized form and asking the remote reference layer to forward the method request and arguments to the appropriate remote object.
3. Marshaling involves converting local objects into portable form so that they can be transmitted to a remote process. Each object (e.g. a String object, an array object, or a user defined object) is checked as it is marshaled, to determine whether it implements the *java.rmi.Remote* interface. If it does, its remote reference is used as its marshaled

data. If it isn't a *Remote* object but is rather a *Serializable* object, the object is serialized into bytes that are sent to the remote host and reconstructed into a copy of the local object. If the object is neither *Remote* nor *Serializable*, the stub throws a *java.rmi.MarshalException* back to the client.

(Object serialization is a scheme that converts objects into a byte stream that is passed to other machines; these rebuild the original object from the bytes)

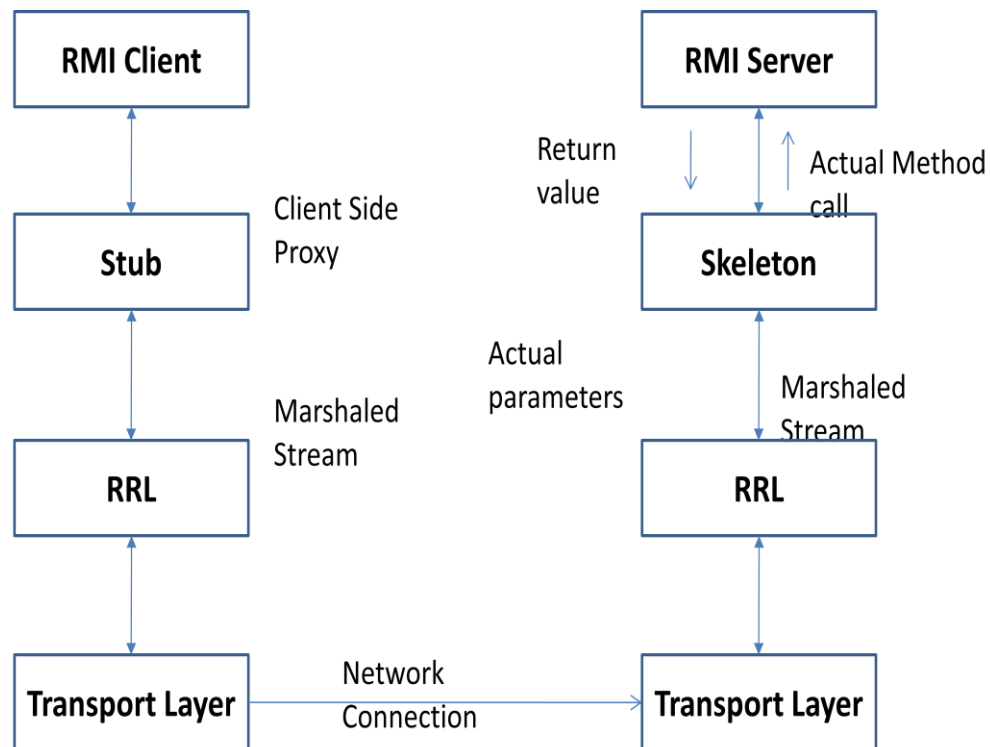


Figure: 1 Remote Method Invocation (RMI) Architecture

4. If the marshaling of method arguments succeeds, the client-side remote reference layer receives the remote reference and marshaled arguments from the stub.
5. The remote reference layer converts the client request into low-level RMI transport requests, i.e., into a single network-level request and sends it over the wire to the sole remote object (since in Java 2 the communication style supported is the point-to-point object references) that corresponds to the remote reference passed along with the request.
6. On the server, the server-side remote reference layer receives the transport-level request and converts it into a request for the server skeleton that matches the referenced object.

7. The skeleton converts the remote request into the appropriate method call on the actual server object. This involves *unmarshaling* the method arguments into the server environment and passing them to the server object. Arguments sent as remote references are converted into local stubs on the server, and arguments sent as serialized objects are converted into local copies of the originals.
8. If the method call generates a return value or an exception, the skeleton marshals the object for transport back to the client and forwards it through the server reference layer.
9. The result is sent back using the appropriate transport protocol (e.g. Socket API using TCP/IP), where it passes through the client reference layer and stub, is unmarshaled by the stub, and is finally handed back to the client thread that invoked the remote method.

RMI Development Steps:

1. Create the remote interface
2. Implementation of Remote Interface
3. Compile the Implementation Class & create Stub and skeleton objects using rmic tool.
4. Start the registry service by rmiregistry tool.
5. Create and Start the remote Application.
6. Create and Start the client Application.

1) Create Remote interface:

A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. If any of these interfaces or classes do not yet exist, you need to define them as well.

Remote interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine. A remote interface must satisfy the following requirements:

- ✓ Create Remote interface by extending **java.rmi.Remote** interface.
- ✓ Declare the **RemoteException** with all the methods of the Remote interface.

Example:

```
// Create remote interface
import java.rmi.*;

public interface Concat extends Remote
{
    public String concatenation(String x, String y) throws RemoteException;
}
```

2) Implementation of Remote Interface

Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to

- ✓ Either extend the UnicastRemoteObject class

(OR)

- ✓ Use the exportObject() method of the UnicastRemoteObject Class

NOTE: If you extend the UnicastRemoteObject class, you must define Constructor that declares Remote Exception

Example:

// Implement the remote interface Concat in this class (extends UnicastRemoteObject class)

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

public class ConcatRemote extends UnicastRemoteObject implements Concat
{
    ConcatRemote()throws RemoteException
    {
        super();
    }
}
```

```

public String concatenation(String x,String y)    // Remote Method Implementation
{
    return x+y;
}
}

```

3) Compile the Implementation Class & create Stub and skeleton objects using rmic tool.

Compile these two java files (Concat.java & ConcatRemote) java compiler generates .class files. Next step is to create stub and skeleton objects using the rmi compiler. The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

```
rmic ConcatRemote
```

Now stub is created with the name called **ConcatRemote_Stub.class**

4) Start the registry service by rmiregistry tool.

Now start the registry service by using the **rmiregistry** tool. If you don't specify the port number, it uses a default port number (1099).

```
Start rmiregistry
```

5) Create and Start the remote Application.

Now rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object. The Naming class provides 5 methods.

- **public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException;** it returns the reference of the remote object.
- **public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException;** it binds the remote object with the given name.
- **public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException;**

it destroys the remote object which is bound with the given name.

- **public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException;**
it binds the remote object to the new name.
- **public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException;**
it returns an array of the names of the remote objects bound in the registry.

Example:**//Creating Server Object**

```
import java.rmi.*;
import java.rmi.registry.*;

public class RemoteServer
{
    public static void main(String args[])
    {
        try{
            Concat stub=new ConcatRemote();
            Naming.rebind("rmi://localhost/phani",stub);
            System.out.println("Remote Object is Ready.....");
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

6) Create and Start the client Application.

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

Example:**// Creating Client application to access remote Object**

```

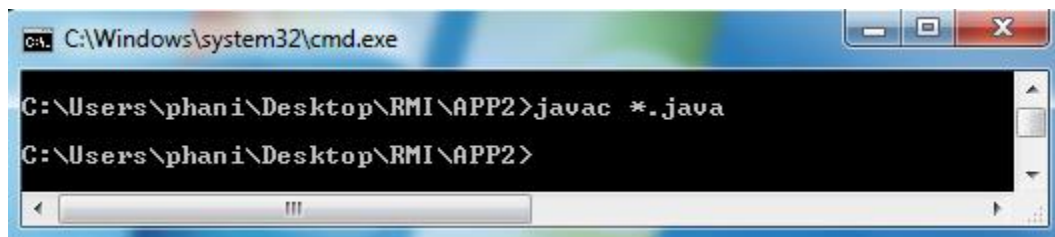
import java.rmi.*;
import java.util.Scanner;
public class LocalClient
{
    public static void main(String args[]){
        try{
            Concat stub=(Concat)Naming.lookup("rmi://localhost/phani");
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter Your First Name: ");
            String fname=sc.next();
            System.out.print("Enter Your Last Name: ");
            String lname=sc.next();

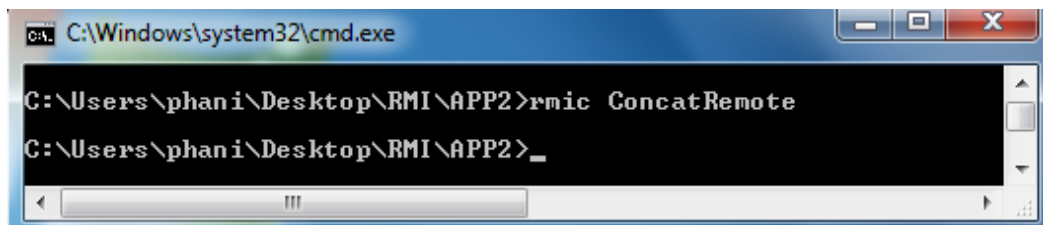
            //calling remote method
            System.out.println(stub.concatination(fname, lname););
        }
        catch(Exception e)
        {System.out.println(e);}
    }
}

```

Compile & Run RMI Application

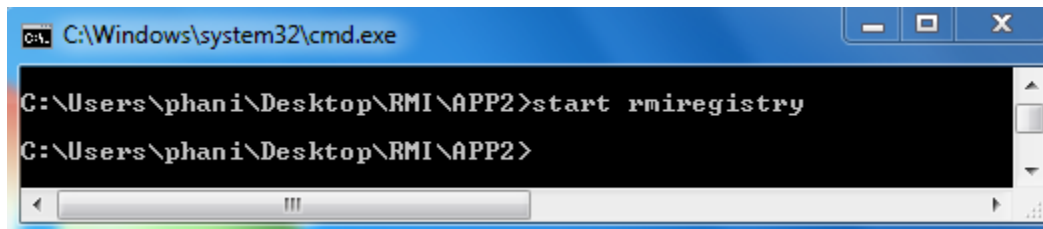
1. Compile all the java files **javac *.java**
2. Create Stub and Skeleton object by **rmic** tool
3. **rmic** ConcatRemote
4. Start **rmiregistry** in one command prompt start rmiregistry
5. Start the server in another command prompt java **RemoteServer**
6. Start the client application in another command prompt java **LocalClient**

OUTPUT:



```
C:\Windows\system32\cmd.exe

C:\Users\phani\Desktop\RMI\APP2>rmic ConcatRemote
C:\Users\phani\Desktop\RMI\APP2>_
```

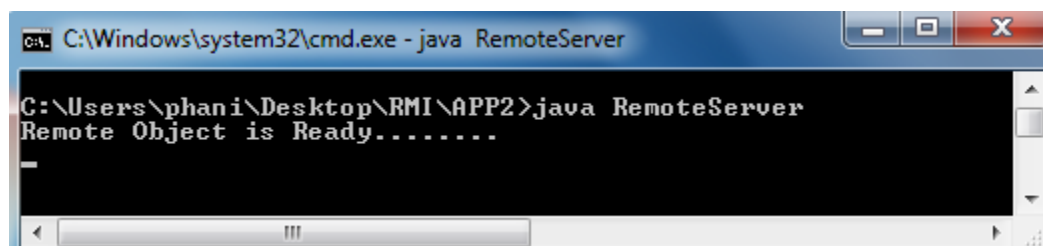


```
C:\Windows\system32\cmd.exe

C:\Users\phani\Desktop\RMI\APP2>start rmiregistry
C:\Users\phani\Desktop\RMI\APP2>
```

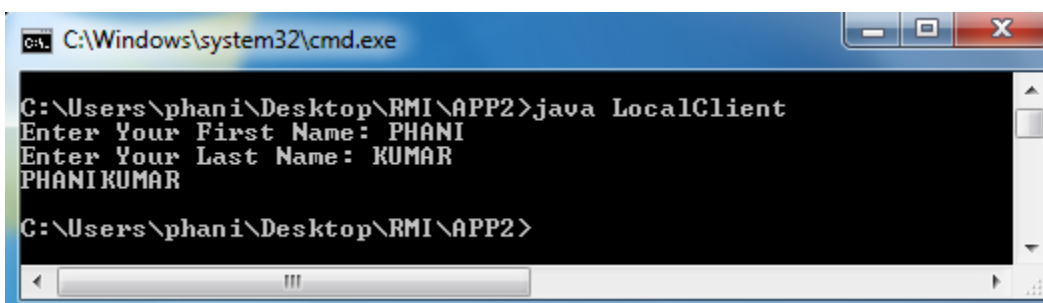


```
C:\Program Files\Java\jdk1.7.0_75\bin\rmiregistry.exe
```



```
C:\Windows\system32\cmd.exe - java RemoteServer

C:\Users\phani\Desktop\RMI\APP2>java RemoteServer
Remote Object is Ready.....
```



```
C:\Windows\system32\cmd.exe

C:\Users\phani\Desktop\RMI\APP2>java LocalClient
Enter Your First Name: PHANI
Enter Your Last Name: KUMAR
PHANIKUMAR
C:\Users\phani\Desktop\RMI\APP2>
```

Advantages of RMI:

- **Object Oriented:** RMI can pass full objects as arguments and return values, not just predefined data types. This means that you can pass complex types, such as a standard Java hashtable object, as a single argument. RMI lets you ship objects directly across the wire with no extra client code.
- **Safe and Secure:** RMI uses built-in Java security mechanisms that allow your system to be safe when users downloading implementations. RMI uses the security manager defined to protect systems from hostile applets to protect your systems and network from potentially hostile downloaded code. In severe cases, a server can refuse to download any implementations at all.
- **Easy to Write/Easy to Use:** RMI makes it simple to write remote Java servers and Java clients that access those servers. A remote interface is an actual Java interface. A server has roughly three lines of code to declare itself a server, and otherwise is like any other Java object. This simplicity makes it easy to write servers for full-scale distributed object systems quickly, and to rapidly bring up prototypes and early versions of software for testing and evaluation
- **Write Once, Run Anywhere:** RMI is part of Java's "Write Once, Run Anywhere" approach. Any RMI based system is 100% portable to any Java Virtual Machine *, as is an RMI/JDBC system.
- **Distributed Garbage Collection:** RMI uses its distributed garbage collection feature to collect remote server objects that are no longer referenced by any clients in the network. Analogous to garbage collection inside a Java Virtual Machine, distributed garbage collection lets you define server objects as needed, knowing that they will be removed when they no longer need to be accessible by clients
- **Parallel Computing:** RMI is multi-threaded, allowing your servers to exploit Java threads for better concurrent processing of client requests.
- **The Java Distributed Computing Solution:** RMI is part of the core Java platform starting with JDK?? 1.1, so it exists on every 1.1 Java Virtual Machine. All RMI systems talk the same public protocol, so all Java systems can talk to each other directly, without any protocol translation overhead.

Disadvantages of RMI:

- ✓ Language - centric. RMI code must be written in Java
- ✓ Overhead of marshaling and unmarshaling.
- ✓ Overhead of object serialization.
- ✓ Security issues need to be monitored more closely.
- ✓ Can be slower than other alternatives (esp. CORBA)

Example #2 Addition of two integer using RMI**1. Create the remote interface**

```
import java.rmi.*;

public interface Adder extends Remote{

    public int add(int x,int y)throws RemoteException;

}
```

2. Implementation of Remote Interface

```
import java.rmi.*;

import java.rmi.server.*;

public class AdderRemote extends UnicastRemoteObject implements Adder{

    AdderRemote()throws RemoteException{

        super();

    }

    public int add(int x,int y) {           // Remote Method Implementation

        return x+y;

    }

}
```

3. Create Stub and skeleton objects using rmic tool.

rmic AdderRemote

4. Start the registry service by rmiregistry tool.

Start rmiregistry

5. Create and Start the remote Application.

```

import java.rmi.*;
import java.rmi.registry.*;

public class MyServer{

public static void main(String args[]){
    try{
        Adder stub=new AdderRemote();
        Naming.rebind("rmi://localhost:5000/phani",stub);
        System.out.println("Remote Object is Ready.....");
    }
    catch(Exception e)
        {System.out.println(e);}
}

}

```

6. Create and Start the client Application

```

import java.rmi.*;

public class MyClient{

public static void main(String args[]){

    try{

        Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/phani");

        System.out.pritln(stub.add(34,4));

    }

    catch(Exception e)

        {System.out.println(e);}

}

}

```