RMI[Remote Method Invocation]
—----------------------------
If we want to prepare distributed applications by using Socket Programming then the Developers must provide the distributed applications infrastructure like ServerSocket, Sockets, InputStreams and OutputStreams,.... , it will increase burden to the developers.

In the above context, to prepare Distributed applications without thinking about the Distributed application's infrastructure by the developers we have to use RMI.

In RMI, we will define a method at Remote machine and we will access that method from the local machine.

In RMI, the distributed application's infrastructure is provided internally in the form of the following components.

1. STUB
2. SKELETON

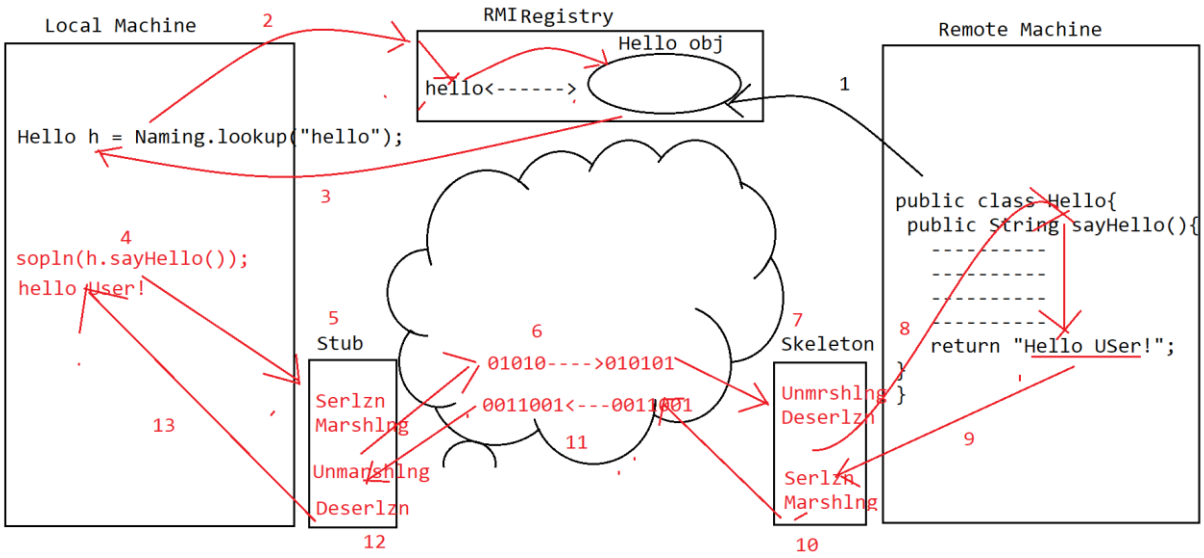
Where STUB is available at Local Machine, it is able to manage the client side socket, the required InputStreams and OutputStreams , the required Serialization and Deserialization and the required Marshaling and Unmarshalling at local machines.

Where SKELETON is available at Remote Machine, it is able to manage the Server side socket, the required InputStreams and OutputStreams , the required Serialization and Deserialization and the required Marshaling and Unmarshalling at Remote machines.

When SKELETON receives a remote method call from STUB , SKELETON is able to access the remote method, SKELETON executes the remote method, SKELETON sends the generated results from remote method to the STUB.

RMI uses a separate Registry Software in the form of RMIRegistry to expose all the Remote objects to all the clients in order to get Remote objects and in order to access remote methods from their local machines.

RMI Arch:



1. Create a Remote Object and keep the generated Remote Object in the RMIRegistry.
2. Client Application has to search for the Remote object in RMIRegistry on the basis of the logical name by using the Naming.lookup() method.
3. If the required Remote object is identified then RMIRegistry will send the Remote object to the Client Application.
4. After getting the Remote Object from RMIRegistry , the Client application will access the Remote method.
5. When we access the Remote method, Stub will take a remote method call and Stub will perform the required serialization and Marshalling over the method call.
6. Stub will send the remote method call to the Network, where Network will carry the remote method call to the SKELETON.
7. When the Remote method call is identified at SKELETON, SKELETON will perform Unmarshalling and Deserialization over the remote method call.
8. Skeleton must access the remote method.
9. Skeleton will execute the remote method and Skeleton will get the return value from the remote method.
10. Skeleton will perform Serialization and Marshalling over the return value and send that return value to the network.
11. Where Network will send the return values to the STUB.
12. Where STUB is able to perform Unmarshalling and Deserialization over the return value.
13. Stub will send the return value to the Client Application.

Steps to prepare RMI Applications:
—---------------------------------
1. Create Remote interface:
The main purpose of Remote interface is to declare the remote methods.
Steps:
   a. Declare an user defined interface.
   b. Extend user defined interface from java.rmi.Remote marker interface.
   c. Declare all the remote methods as per the requirement.
   d. Every Remote method must throws java.rmi.RemoteException

```
public interface HelloRemote extends Remote{
      public String sayHello(String name)throws RemoteException;
}
```

2. Create an implementation class for the Remote interface:
The main purpose of the Remote interface's implementation class is to provide implementation for all the remote methods.

Steps:
   a. Declare an user defined class.
   b. Extend java.rmi.server.UnicastRemoteObject to the user defined class in order to give eligibility to expose in the network.
   c. Implement User defined Remote interface.
   d. Provide a 0-arg constructor with throws RemoteException.
   e. Provide implementation for all the remote methods.

```
public class HelloRemoteImpl extends UnicastRemoteObject implements
HelloRemote{
      public HelloRemoteImpl()throws RemoteException{
      }
      public String sayHello(String name)throws RemoteException{
            return “Hello User!”;
      }
}
```

3. Create Registry program:
The main purpose of the Registry program is to create a Remote Object and to keep the generated Remote Object in the RMIRegistry along with a particular logical name.

Steps
   a. Declare an user defined class with main() method.
   b. Create Remote object

c. Bind the remote object in RMIRegistry by using the following method from the Naming class.

public static void bind(String name, Remote object)throws RemoteException

EX:

```
public class HelloRegistry{
      public static void main(String[] args)throws Exception{
            HelloRemote helloRemote = new HelloRemoteImpl();
            Naming.bind("hello", helloRemote);
      }
}
```

4. Create Test or Client Application:

The main purpose of the Client application is to get the Remote Object from RMIRegistry and to access remote methods.

Steps:

a. Declare an user defined class with the main() method.
b. Search for the Remote Object in RM<IRegistry by using the following method from naming class.
   public Remote lookup(String name)throws RemoteException
c. Access remote methods.

EX:

```
public class Test{
      public static void main(String[] args)throws Exception{
            HelloRemote hr = (HelloRemote)Naming.lookup("hello");
            System.out.println(hr.sayHello("Durga"));
      }
}
```

5. Execute Client Application:

a. Compile all the Java files
   javac *.java
b. Start RMIRegistry
   start rmiRegistry
c. Execute Registry program
   start java HelloRegistry

d. Execute Test Application
   java Test

EX:

CalculatorRemote.java

```
import java.rmi.*;
public interface CalculatorRemote extends Remote{
      public int add(int i, int j)throws RemoteException;
      public int sub(int i, int j)throws RemoteException;
      public int mul(int i, int j)throws RemoteException;
}
```

CalculatorRemoteImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
public class CalculatorRemoteImpl extends UnicastRemoteObject implements
CalculatorRemote{
      public CalculatorRemoteImpl()throws RemoteException{

      }
      public int add(int i, int j)throws RemoteException{
            return i+j;
      }
      public int sub(int i, int j)throws RemoteException{
            return i-j;
      }
      public int mul(int i, int j)throws RemoteException{
            return i*j;
      }
}
```

CalculatorRegistry.java

```
import java.rmi.*;
public class CalculatorRegistry{
      public static void main(String[] args)throws Exception {
            CalculatorRemote calRemote = new CalculatorRemoteImpl();
            Naming.bind("calRemote", calRemote);
            System.out.println("CalculatorRemote Object is binded with the
logical name 'calRemote' in RMIRegistry");
      }
}
```

ClientApp.java

```
import java.rmi.*;
public class ClientApp{
      public static void main(String[] args)throws Exception {
```

```
            CalculatorRemote calRemote = (CalculatorRemote)
Naming.lookup("calRemote");
            System.out.println("ADD     : "+calRemote.add(10,5));
            System.out.println("SUB     : "+calRemote.sub(10,5));
            System.out.println("MUL     : "+calRemote.mul(10,5));
      }
}
```

COmmand Prompt1:

```
D:\java6\rmi\app02>javac *.java
```

COmmand Prompt2:

```
D:\java6\rmi\app02>rmiregistry
```

COmmand Prompt3:

```
D:\java6\rmi\app02>java CalculatorRegistry
CalculatorRemote Object is binded with the logical name 'calRemote' in
RMIRegistry
```

Command Prompt1:

```
D:\java6\rmi\app02>javac *.java

D:\java6\rmi\app02>java ClientApp
ADD     : 15
SUB     : 5
MUL     : 50

D:\java6\rmi\app02>
```