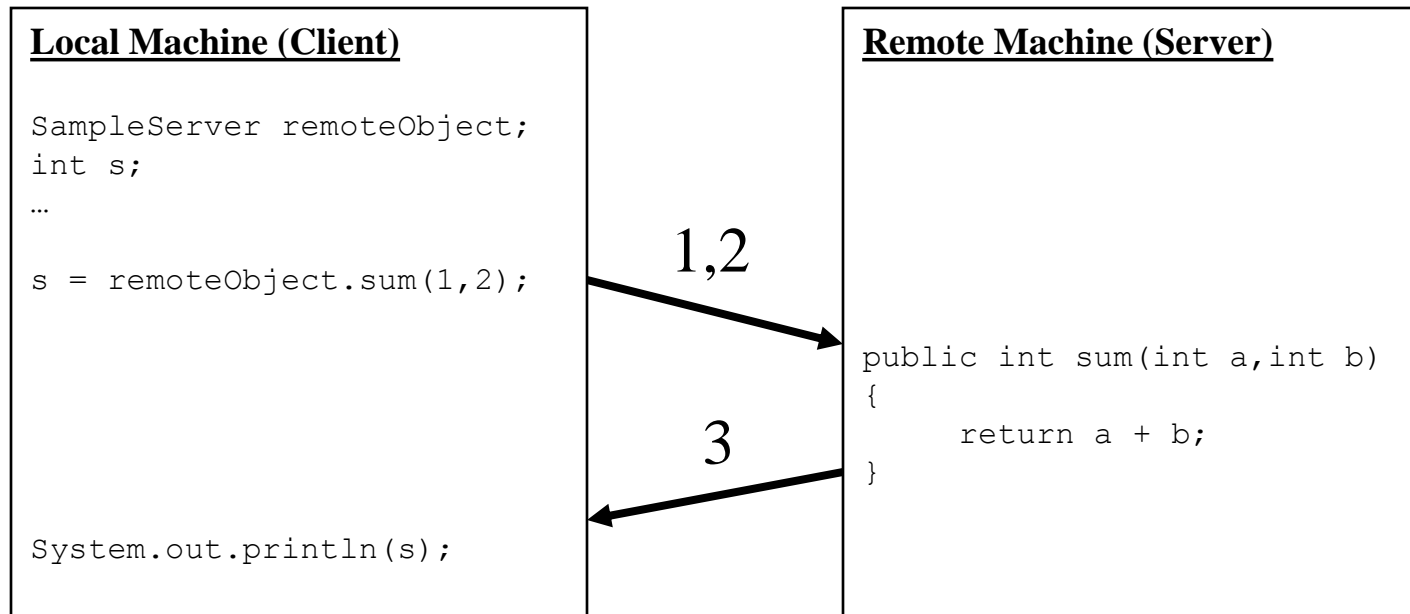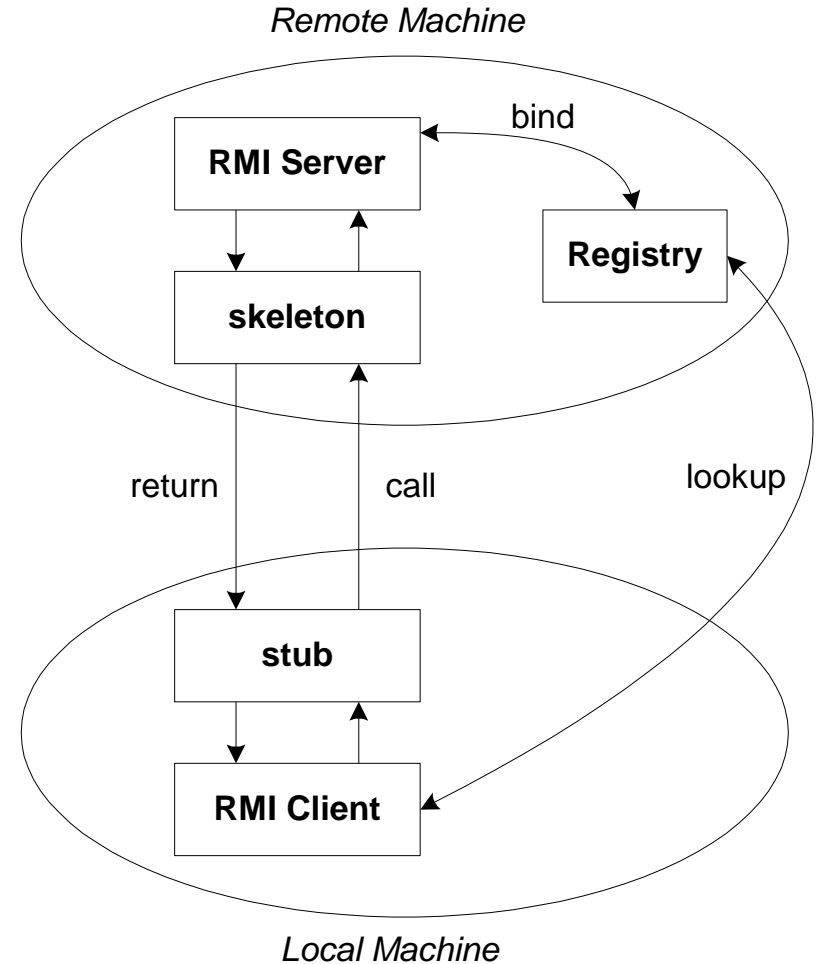# Java Remote Object Invocation (RMI)

- Overview of RMI
- Java RMI allowed programmer to execute remote function class using the same semantics as local functions calls.

Local Machine (Client)

```
SampleServer remoteObject;
int s;
…

s = remoteObject.sum(1,2);




System.out.println(s);
```

1,2

3

Remote Machine (Server)

```
public int sum(int a,int b)
{
      return a + b;
}
```
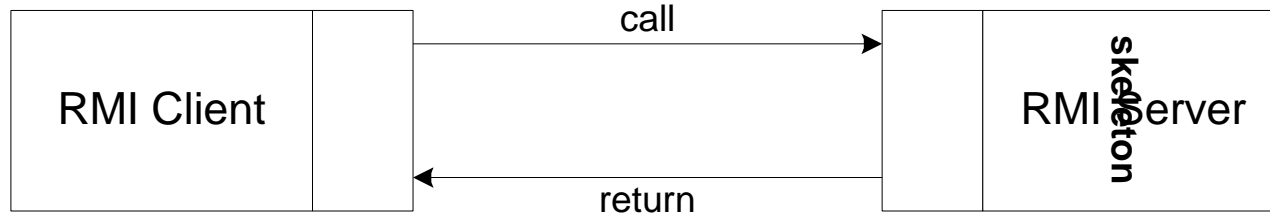
The General RMI Architecture

- The server must first bind its name to the registry

- The client lookup the server name in the registry to establish remote references.

- The Stub serializing the parameters to skeleton, the skeleton invoking the remote method and serializing the result back to the stub.

Remote Machine

RMI Server

bind

Registry

skeleton

return   call   lookup

stub

RMI Client

Local Machine

The Stub and Skeleton

```
┌─────────────────┬──┐        call        ┌──┬─────────────────┐
│                 │  │ ─────────────────► │  │      skeleton   │
│   RMI Client    │  │                    │  │   RMI Server    │
│                 │  │ ◄───────────────── │  │                 │
└─────────────────┴──┘       return       └──┴─────────────────┘
```

- A client invokes a remote method, the call is first forwarded to stub.

- The stub is responsible for sending the remote call over to the server-side skeleton

- The stub opening a socket to the remote server, marshaling the object parameters and forwarding the data stream to the skeleton.

- A skeleton contains a method that receives the remote calls, unmarshals the parameters, and invokes the actual remote object implementation.

Steps for Developing an RMI System

1. Define the remote interface

2. Develop the remote object by implementing the remote interface.

3. Develop the client program.

4. Compile the Java source files.

5. Generate the client stubs and server skeletons.

6. Start the RMI registry.

7. Start the remote server objects.

8. Run the client

Step 1: Defining the Remote Interface

- To create an RMI application, the first step is the defining of a remote interface between the client and server objects.

```
/* SampleServer.java */
import java.rmi.*;

public interface SampleServer extends Remote
{
  public int sum(int a,int b) throws RemoteException;
}
```

Step 2: Develop the remote object and its interface

- The server is a simple unicast remote server.
- Create server by extending `java.rmi.server.UnicastRemoteObject`.
- The server uses the RMISecurityManager to protect its resources while engaging in remote communication.

```java
/* SampleServerImpl.java */
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;


public class SampleServerImpl extends UnicastRemoteObject
                              implements SampleServer
{
  SampleServerImpl() throws RemoteException
  {
     super();
  }
```

Step 2: Develop the remote object and its interface

- Implement the remote methods

```
/* SampleServerImpl.java */
 public int sum(int a,int b) throws RemoteException
 {
    return a + b;
 }
}
```

- The server must bind its name to the registry, the client will look up the server name.

- Use `java.rmi.Naming` class to bind the server name to registry. In this example the name call "SAMPLE-SERVER".

- In the main method of your server object, the RMI security manager is created and installed.

Step 2: Develop the remote object and its interface

```java
/* SampleServerImpl.java */
  public static void main(String args[])
  {
      try
      {
        System.setSecurityManager(new RMISecurityManager());
        //set the security manager

        //create a local instance of the object
        SampleServerImpl Server = new SampleServerImpl();

        //put the local instance in the registry
        Naming.rebind("SAMPLE-SERVER" , Server);

        System.out.println("Server waiting.....");
      }
      catch (java.net.MalformedURLException me)        {
        System.out.println("Malformed URL: " + me.toString());    }
      catch (RemoteException re)  {
         System.out.println("Remote exception: " + re.toString());  }
  }
```

Step 3: Develop the client program

- In order for the client object to invoke methods on the server, it must first look up the name of server in the registry. You use the `java.rmi.Naming` class to lookup the server name.

- The server name is specified as URL in the from (rmi://host:port/name)

- Default RMI port is 1099.

- The name specified in the URL must exactly match the name that the server has bound to the registry. In this example, the name is "SAMPLE-SERVER"

- The remote method invocation is programmed using the remote interface name (`remoteObject`) as prefix and the remote method name (`sum`) as suffix.

Step 3: Develop the client program

```java
import java.rmi.*;
import java.rmi.server.*;
public class SampleClient
{
    public static void main(String[]  args)
    {
        // set the security manager for the client
        System.setSecurityManager(new RMISecurityManager());
        //get the remote object from the registry
        try
          {
            System.out.println("Security Manager loaded");
            String url = "//localhost/SAMPLE-SERVER";
            SampleServer remoteObject = (SampleServer)Naming.lookup(url);
            System.out.println("Got remote object");
            System.out.println(" 1 + 2 = " + remoteObject.sum(1,2) );
          }
        catch (RemoteException exc) {
            System.out.println("Error in lookup: " + exc.toString()); }
        catch (java.net.MalformedURLException exc) {
            System.out.println("Malformed URL: " + exc.toString());    }
        catch (java.rmi.NotBoundException exc)  {
            System.out.println("NotBound: " + exc.toString());
          }
    }
}
```

Step 4 & 5: Compile the Java source files & Generate the client stubs and server skeletons

- Assume the program compile and executing at elpis on ~/rmi

- Once the interface is completed, you need to generate stubs and skeleton code. The RMI system provides an RMI compiler (`rmic`) that takes your generated interface class and procedures stub code on its self.

```
elpis:~/rmi> set CLASSPATH="~/rmi"
elpis:~/rmi> javac SampleServer.java
elpis:~/rmi> javac SampleServerImpl.java
elpis:~/rmi> rmic SampleServerImpl

elpis:~/rmi> javac SampleClient.java
```

Step 6: Start the RMI registry

- The RMI applications need install to Registry. And the Registry must start manual by call `rmiregisty`.

- The `rmiregistry` us uses port 1099 by default. You can also bind rmiregistry to a different port by indicating the new port number as : `rmiregistry <new port>`

  `elpis:~/rmi> rmiregistry`

- *Remark: On Windows, you have to type in from the command line:*
  `> start rmiregistry`

Steps 7 & 8: Start the remote server objects & Run the client

- Once the Registry is started, the server can be started and will be able to store itself in the Registry.

- Because of the grained security model in Java 2.0, you must setup a security policy for RMI by set `java.security.policy` to the file `policy.all`

```
elpis:~/rmi> java -Djava.security.policy=policy.all
    SampleServerImpl

elpis:~/rmi> java -Djava.security.policy=policy.all
    SampleClient
```

Java Policy File

- In Java 2, the java application must first obtain information regarding its privileges. It can obtain the security policy through a policy file. In above example, we allow Java code to have all permissions, the contains of the policy file policy.all is:

```
grant {
permission java.security.AllPermission;
};
```

- Now, we given an example for assigning resource permissions:

```
grant {
permission java.io.filePermission "/tmp/*", "read",
"write";
permission java.net.SocketPermission
"somehost.somedomain.com:999","connect";
permission java.net.SocketPermission "*:1024-
65535","connect,request";
permission java.net.SocketPermission "*:80","connect";
};
```

Comment for the Java Policy File

1. allow the Java code to read/write any files only under the /tmp directory, includes any subdirectories

2. allow all java classes to establish a network connection with the host "somehost.somedomain.com" on port 999

3. allows classes to connection to or accept connections on unprivileged ports greater than 1024 , on any host

4. allows all classes to connect to the HTTP port 80 on any host.

- *You can obtain complete details by following links:*

```
http://java.sun.com/products//jdk/1.2/docs/guide/security/spec
   /security-spec.doc3.html
```