**Distributed Applications:**

Distributed object system is a technology that combines networking and object oriented programming.

A distributed system contains a number of nodes that perform computations. A node may be a PC, a mainframe computer or any other device. Nodes of distributed system are scattered. The node in use is the local node and any other node is the remote node. Network makes distributed computing possible.

**Distributed Computing Technologies:**

**RMI vs. CORBA**

• RMI

– Java-only distributed object model

– relatively simple to use

• especially if already familiar with Java

– software needed is free

– new so lacks maturity

• but had knowledge of CORBA

• many RMI designers participated in the design of CORBA

• CORBA

– language independent distributed object model

• but nearly all CORBA development is done in C++

– relatively complex to use

– software needed to use is expensive

**RMI Architecture:**

RMI architecture consists of four layers:

1. Application Layer ii. Proxy layer iii. Remote Reference Layer iv. Transport Layer
2. **Application Layer:** This layer contains the actual implementation of the client and server applications. It is in this layer that high level calls are made to access and export remote objects.
3. **Proxy Layer:** This layer contains the client stub objects. The application layer communicates with this proxy layer directly. All calls to remote objects and marshalling(converting remote method arguments to a stream of bytes in a stub )of parameters and return objects are done through these proxies.
4. **Remote Reference Layer:** It handles packaging of method call and its parameters and its return values for transport over the network. The remote reference layer uses a server-side and client-side component to communicate via the network layer.
5. **Transport Layer:** The transport layer sets up connections, manages existing connections and handles remote objects residing in its address space.

**Remote Method Invocation**

Java’s *Remote Method Invocation (RMI)* facility lets you make method calls on objects that

exist outside your JVM. RMI hides many of the details that you have to take care of when you

use object streams, including knowledge of the server port number, socket and stream maintenance, and the need for a command class.

**Remote References**

The goal of RMI is to support remote calls in a way that resembles as much as possible the experience of making local calls. When you make a call on a local object—that is, on an object in the same JVM as the code making the call—you use a reference to the local object. RMI provides *remote references*, which act as references to remote objects. Remote references don’t really point to remote objects, though they do a good job of sustaining that illusion. They really point to local objects called *stubs*.

A stub is a local object that supports RMI by communicating with a remote object. You make an RMI call by making a call on a stub. The stub is in communication with the host that owns the remote object. The stub sends the method arguments to the remote object and tells it to execute the appropriate method. The remote object executes the method; if there is a return value or if an exception is thrown, the remote object transmits the return value or exception back to the stub.

Using RMI is easy: you just use a remote reference to make calls on a stub, and the stub takes care of the work. You don’t even have to write the stub code; it’s written for you by the rmic (RMIC compiler).

**RMI Step by Step**

The steps are

**1.** Create the remote interface.

**2.** Create the remote class.

**3.** Create the stub.

**4.** Create the remote server.

**5.** Create the client.

**6.** Start the programs.

**Step 1: Create the Remote Interface**

A *remote interface* is an interface that describes the remotely accessible methods of a remote object. A remote object and the stub that communicates with it both implement the remote interface. Recall that RMI clients use remote references, which point to stubs. The type of a remote reference is a remote interface. So a client makes RMI calls by calling the remote interface’s methods on the remote reference. All remote interfaces must extend java.rmi.Remote.

All methods in a remote interface must throw java.rmi.RemoteException, in addition to any other specified exceptions. The stub throws RemoteException if something goes wrong with the connection to the server. This means that in all client code, all RMI calls must catch RemoteException or deal with it by some other means.

RMI uses serialized objects to send method arguments from a client to a server and to send return values and exceptions back to a client from a server. Thus all arguments, return values, and exceptions must be serializable. RMI can deal with primitive arguments and return values, so what you really need to remember about a remote interface’s methods is that all object-type arguments and return values must be serializable.

**Following is a remote interface that describes the remote services:**

import java.rmi.\*;

public interface MathServices extends Remote {

public double add(double[] addUs)

throws RemoteException;

public double subtract(double[] subtractUs)

throws RemoteException;

}

Notice that this interface fulfills its two requirements: it extends Remote, and its methods throw RemoteException. Now let’s see how the remote interface is used.

**Step 2: Create the Remote Class**

The remote interface is implemented by the remote class and the stub. Here we’ll look at the remote

class. This class is constructed by the server and then made available for remote invocation. There are two requirements on a remote class:

\_ It must extend java.rmi.server.UnicastRemoteObject.

\_ It must implement the remote interface.

The remote class is the workhorse of an RMI system. It is the class whose methods provide services to clients. **Here is a remote class whose methods provide the services of our mathematical example:**

import java.rmi.\*;

import java.rmi.server.\*;

public class Mathematician

extends UnicastRemoteObject

implements MathServices {

public Mathematician() throws RemoteException { }

public double add(double[] addUs)

throws RemoteException {

return addUs[0] + addUs[1];

}

public double subtract(double[] subtractUs)

throws RemoteException {

return subtractUs[0] - subtractUs[1];

}

}

The only tricky part of this class is the constructor. The no-args constructor of the superclass

(java.rmi.server.UnicastRemoteObject) throws java.rmi.RemoteException, so a subclass

must explicitly provide a constructor that also throws this exception, even if the constructor’s

body does nothing.

Now let’s see how to make a stub.

**Step 3: Create the Stub**

A stub must be created for every class whose methods are to be made available for RMI calling.

The stub and the remote class both implement the remote interface. On the client side, a remote reference points to an instance of the stub. Stubs are also used on the server side.

Stubs are created by using the rmic tool, not by writing and compiling Java source code. rmic

resides in the JDK’s bin directory, so it’s already in your path if you can compile and execute. To

create a stub, type **rmic *remote\_class\_name***. In our case, the remote class name is Mathematician, so the corresponding stub is created by typing **rmic Mathematician**.The output from rmic is a classfile whose name is the remote class name with \_Stub appended at the end. So in our example the stub is called Mathematician\_Stub.

Now the three essential pieces have been created: the remote interface, the remote class, and

the stub. The next step is to create server and client code.

**Step 4: Create the Server**

An RMI server is an application that creates one or more remote objects and makes them available

to clients. Creating an RMI server is not completely straightforward, because of the need to interact

with the RMI registry. The RMI registry is a program that associates names with RMI services. A server specifies a name for every remote object it provides. A client accesses a remote object by specifying the server and the service name. In general, the names in the registry are descriptive of the services they represent. Choosing good service names is as important as choosing good variable names. The RMI registry is a process separate from any individual JVM. It must be running before any Java applications try to interact with it. To start the registry, type **rmiregistry** on a command line. The program resides in the Java bin directory. On a single machine, multiple Java applications can offer services through a single registry.

An application accesses the registry via the java.rmi.Naming class, all of whose public methods are static. To associate a name with a remote object, a server should call Naming.rebind(), which has the following signature:

public static void rebind(String name, Remote obj)

The name argument is the name by which clients will access the remote object. The obj argument

is the remote object itself; its type is Remote rather than Object because a remote object implements a remote interface, which extends Remote. You can look up the other methods of the Naming class in the API; it’s possible to bind a new object to a name and to unbind a name (that is, to tell the registry that the name no longer corresponds to the object). A server continues to run as long as at least one remote object is bound.

An RMI server performs two tasks:

**1.** Create an instance of the remote object.

**2.** Bind the remote object to a name.

**Here is a server for our example. The service name is brainiac.**

import java.io.\*;

import java.rmi.\*;

public class MathServer {

public static void main(String[] args) {

try {

Mathematician m = new Mathematician();

Naming.rebind("brainiac", m); //to associate a name with a remote object, a server should call Naming.rebind()

}

catch (RemoteException x) {

System.out.println("Remote Exception stress " + x);

}

catch (IOException x) {

System.out.println("Other IOException stress " + x);

}

}

}

Recall that remote objects extend UnicastRemoteObject, so their constructors throw

RemoteException. The Naming.rebind() call throws various subclasses of IOException. Servers must always be prepared to handle multiple clients. You saw in the object streams server example that it’s a good strategy for a server to spawn off a service thread as soon as possible after receiving a client connection on a server socket. With RMI you have no access to the server socket, but you still need to take precautions. No matter how many clients concurrently call a remote method, there is still only one object executing the method.

**Step 5: Create the Client**

A client obtains a remote reference by contacting the RMI registry on the server machine. This

is done by calling the static lookup() method of the Naming class. The signature of lookup() is public static Remote lookup(String name) Something seems to be missing from the signature. The object name is provided, but not the server hostname. How does the RMI infrastructure know which host to contact?

The name argument actually specifies both the server hostname and the object name, in a URL format. The format of this string is rmi://server\_hostname/object\_name. The object name must be a name that has been bound to an object in the server’s registry. It is optionally legal to omit everything except the object name. In this case the RMI infrastructure uses the local host as a server. This is very convenient for debugging, since you can develop code on a single machine.

The return type of Naming.lookup() is Remote, which is the superclass of the remote interface.

The returned value should be cast to the remote interface type.After the remote reference is obtained, calls made on the remote reference are automaticallysent to the server. From a programmer’s point of view the only difference between an ordinary call and an RMI call is that all RMI calls throw RemoteException. Here is a client application that uses RMI to remotely call the add() and subtract() methods of the brainiac service:

import java.io.\*;

import java.rmi.\*;

public class MathClient {

public static void main(String[] args) {

try {

// Get remote reference.

String url = “rmi://” + args[0] + “/brainiac”;

MathServices ms = (MathServices)Naming.lookup(url); //to obtain a remote reference by contacting the RMI registry on the server machine

// Add.

double[] addUs = { 12.34, 56.78 };

double sum = ms.add(addUs);

System.out.println("Sum = " + sum);

// Subtract.

double[] subtractUs = { 99.99, 76.54 };

double difference = ms.subtract(subtractUs);

System.out.println("Difference = " + difference);

}

catch (NotBoundException x) {

System.out.println("Name stress " + x);

}

catch (RemoteException x) {

System.out.println("Remote Exception stress " + x);

}

catch (IOException x) {

System.out.println("Other IOException stress " + x);

}

}

}

The user supplies the remote server’s hostname on the command line. At this point all necessary code has been written. All that remains is to deploy and execute the client and server applications.

**Step 6: Start the Programs**

Before you start the programs that constitute an RMI application, you need to make sure that

the right files are present on the right machines. Remember that the remote interface and the

stub must be deployed on both the server and the client.

The various programs must be started up in the following order:

**1.** The server’s RMI registry

**2.** The server

**3.** The client or clients

The RMI registry must be started first. If it is not running when the server application calls

Naming.rebind(), an exception will be thrown. The server should be started next, so that the

desired remote objects will be available when clients call Naming.lookup(). Clients should be

the last applications to be started.