Prepared By: Shiba R. Tamrakar

Distributed Programming with RMI

Overview

Distributed object computing extends an object-oriented programming system by allowing objects to be distributed across a heterogeneous network, so that each of these distributed object components interoperate as a unified whole.

- These objects may be distributed on different computers throughout a network, living within their own address space outside of an application, and yet appear as though they were local to an application.
- Three of the most popular distributed object paradigms are :
    - Microsoft's **Distributed Component Object Model (DCOM)**
    - OMG's **Common Object Request Broker Architecture (CORBA)**
    - JavaSoft's **Java/Remote Method Invocation (Java/RMI)**.
    - **Enterprise Java Beans (JEB)**

CORBA

- **CORBA** relies on a protocol called the **Internet Inter-ORB Protocol (IIOP)** for remoting objects.
- Everything in the CORBA architecture depends on an **Object Request Broker (ORB)**.
- The ORB acts as a central Object Bus over which each CORBA object interacts transparently with other CORBA objects located either locally or remotely.
- Each CORBA server object has an interface and exposes a set of methods.
- To request a service, a CORBA client acquires an object reference to a CORBA server object.
- The client can now make method calls on the object reference as if the CORBA server object resided in the client's address space.
- The ORB is responsible for finding a CORBA object's implementation, preparing it to receive requests, communicate requests to it and carry the reply back to the client.
- A CORBA object interacts with the ORB either through the ORB interface or through an **Object Adapter** - either a **Basic Object Adapter (BOA)** or a **Portable Object Adapter (POA)**.
- Since CORBA is just a specification, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is an ORB implementation for that platform.
- Major ORB vendors like Inprise have CORBA ORB implementations through their VisiBroker product for Windows, UNIX and mainframe platforms and Iona through their Orbix product.

DCOM

- **DCOM** which is often called '*COM on the wire*', supports remoting objects by running on a protocol called the **Object Remote Procedure Call (ORPC)**.
- This ORPC layer is built on top of DCE's RPC and interacts with COM's run-time services.
- A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime.
- Each DCOM server object can support *multiple interfaces* each representing a different behavior of the object.
- A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces.
- The client object then starts calling the server object's exposed methods through the acquired interface

pointer as if the server object resided in the client's address space. As specified by COM, a server object's memory layout conforms to the C++ vtable layout.

- Since the COM specification is at the binary level it allows DCOM server components to be written in diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL.
- As long as a platform supports COM services, DCOM can be used on that platform.
- DCOM is now heavily used on the Windows platform. Companies like Software AG provide COM service implementations through their **EntireX** product for UNIX, Linux and mainframe platforms; Digital for the Open VMS platform and Microsoft for Windows and Solaris platforms.
- *Enterprise JavaBeans* (EJB) implements server-side, arbitrarily scalable, transactional, multi-user, secure enterprise-level applications.
- It is Java's component model for enterprise applications.

Enterprise Java Beans (EJB)

- EJB combines distributed object technologies such as CORBA and Java RMI with server-side components to simplify the task of application development.
- EJBs can be built on top of existing transaction processing systems including traditional transaction processing monitors, Web servers, database servers, and application servers.
- Sun claims that EJB is not just platform independent--it is also implementation independent.
- An EJB component can run in any application server that implements the EJB specification.
- User applications and beans are isolated from the details of the component services.
- The ability to reuse the same enterprise bean in different specific applications is one advantage of this separation.
- The bean implementation or the client application need not have the parameters that control a bean's transactional nature, persistence, resource pooling, or security management.
- These parameters can be specified in separate deployment descriptors.
- So, when a bean is deployed in a distributed application, the properties of the deployment environment can be accounted for and reflected in the setting of the bean's options.
- EJB is a distributed component model. A distributed component model defines how components are written.
- Hence different people can build systems from components with little or no customization.
- EJB defines a standard way of writing distributed components.
- The EJB client only gets a reference to an EJBObject instance and never really gets a reference to the actual EJB Bean instance itself.
- The EJBObject class is the client's view of the enterprise Bean and implements the remote interface.
- EJB is a rather specialized architecture--aimed at transaction processing and database access.
- It is not really an environment for general distributed computing in our sense.

RMI (Remote Method Invocation)

- **Java/RMI** relies on a protocol called the **Java Remote Method Protocol (JRMP)**.
- Java relies heavily on Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream.
- Since Java Object Serialization is specific to Java, both the Java/RMI server object and the client object have to be written in Java.
- Each Java/RMI Server object defines an interface which can be used to access the server object outside of the current Java Virtual Machine (JVM) and on another machine's JVM.
- The interface exposes a set of methods which are indicative of the services offered by the server object.
- For a client to locate a server object for the first time, RMI depends on a naming mechanism called an

RMIRegistry that runs on the Server machine and holds information about available Server Objects.

- A Java/RMI client acquires an object reference to a Java/RMI server object by doing a **lookup** for a Server Object reference and invokes methods on the Server Object as if the Java/RMI server object resided in the client's address space.
- Java/RMI server objects are named using URLs and for a client to acquire a server object reference, it should specify the URL of the server object as you would with the URL to a HTML page.
- Since Java/RMI relies on Java, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is a Java Virtual Machine (JVM) implementation for that platform.
- In addition to Javasoft and Microsoft, a lot of other companies have announced Java Virtual Machine ports.
- RMI allows us to execute methods on remote server. It helps us locate and execute methods of remote objects.
- It's like placing a class on Machine A and calling methods of that class from Machine B as through they were from the same machine.
- It is the easiest solution for distributed programming.
- RMI is a pure Java solution unlike CORBA where we can have objects from different programming languages interacting.

Limitation of RMI

Inherently, RMI server is not multi threaded. As a programmer, you have to take care of the variables in it. If a programmer wishes to create a nontrivial application using RMI then it faces several difficulties, which is as described below:

- Firstly, because of the requirements imposed on remote interfaces (they must implement Remote, and their methods must throw **RemoteException**), it is found that interfaces not designed with RMI in mind cannot be used remotely.
- Secondly, calls to remote interfaces prove bulky, because they must be enclosed in a try/catch block for catching the possible **RemoteException**. Therefore, throughout the application, you must scatter exception-handling code. To avoid doing so, developers usually limit remote invocation to a small portion of their programs.
- Thirdly, it is found that the bother of **RemoteExceptions** also makes it difficult to user interfaces designed for remote execution locally.
- Fourthly, no convenient approach can generically handle disconnections from a server in a single location. (An example of such handling might include looking up the remote object again and trying to re-invoke the method.)
- Since they normally extend **UnicastRemoteObject**, therefore implementations of Remote interfaces cannot easily extend arbitrary classes. (You could avoid this limitation with some effort, by re-implementing **UnicastRemoteObject**)
-

Concept

- Objects which have to be made available to other machines have to be exported to ***Remote Registry Server*** so that they can be invoked.
- If Machine A wants to call methods of some object on machine B, then Machine B would have to export that object on its Remote Registry Server (RRS).
- RRS is a service that runs on the server and helps client's search and access objects on the server remotely.
- Now, if an object has to be capable of being exported then it must implement the ***Remote Interface*** present in the RMI package.
- For example, say that we want an object Xyz on machine A to be available for remote method invocation, than it must implement the ***Remote Interface***.
- RMI uses **stub** and a ***skeleton***.

- The ***stub*** is present on the ***client side***, and the **skeleton** the ***server side***.
- There are a number of events that have to take place beforehand which help in the communication of the data.
- The stub is like a local object on the client side, which acts like a proxy of the object on the server side.
- It provides the methods to the client which can be invoked on the server.
- The stub then sends the method call to the skeleton, which is present on the server side.
- The Skeleton then implements the method on the server side.
- The stub and skeleton communicate with each other through Remote Reference Layer (RRL).
- This layer gives the stub and skeleton the capability to send data using the TCP/IP protocol.
- Whenever a client wants to make a reference to any object on the server, have we thought how he would tell the server what object he wants to create? Well, this is where the concept of ***Binding*** comes in.
- On the server end we associate a string variable with an object.
- The client tells the server what object he wants to create by passing that string to these strings and objects are stored in the RRS on the Server.

Some terminologies used in RMI coding

- **The Remote Object**
  - This interface will have only method declarations.
  - It will have a declaration for each method that we want to export.
  - This would implement the Remote interface, which is present in the Java.rmi package.
- **The Remote Object Implementation**
  - This is a class that implements the Remote Object.
  - If we implement the Remote Object, its common sense that we implementation class would actually have all of the method bodies of the methods that we want to export.
  - This method will be extended from the UnicastRemoteObject class.
- **The Remote Server**
  - This is a class that will act like a server to the client wanting to access remote methods.
  - Here's the place where we will bind any string with the object we want to export.
  - The binding process will be taken care of in this class.
- **The Remote Client**
  - This is a class that will help us access the remote method.
  - This is the end user, the client.
  - We will call the remote method from this class.
  - We will use methods to search and invoke that remote method.

RMI Architecture

There are three layers that comprise the basic remote-object communication facilities in RMI:

- The ***stub/skeleton*** layer, which provides the interface that client and server application objects use to interact with each other.
- The ***remote reference*** layer, which is the middleware between the stub/skeleton layer and the underlying transport protocol. This layer handles the creation and management of remote object references.
- The ***transport protocol*** layer, which is the binary data protocol that sends remote object requests over the wire.

- These layers interact with each other as shown in Figure 3-1.In this figure, the server is the application that provides remotely accessible objects, while the client is any remote application that communicates

with these server objects.

- In a distributed object system, the distinctions between clients and servers can get pretty blurry at times.
- Consider the case where one process registers a remote-enabled object with the RMI naming service, and a number of remote processes are accessing it.
- We might be tempted to call the first process the server and the other processes the clients.
- But what if one of the clients calls a method on the remote object, passing a reference to an RMI object that's local to the client.
- Now the server has a reference to and is using an object exported from the client, which turns the tables somewhat.
- The "server" is really the server for one object and the client of another object, and the "client" is a client and a server, too.
- For the sake of discussion, I'll refer to a process in a distributed application as a server or client if its role in the overall system is generally limited to one or the other.
- In peer-to-peer systems, where there is no clear client or server, I'll refer to elements of the system in terms of application-specific roles (e.g., chat participant, chat facilitator).

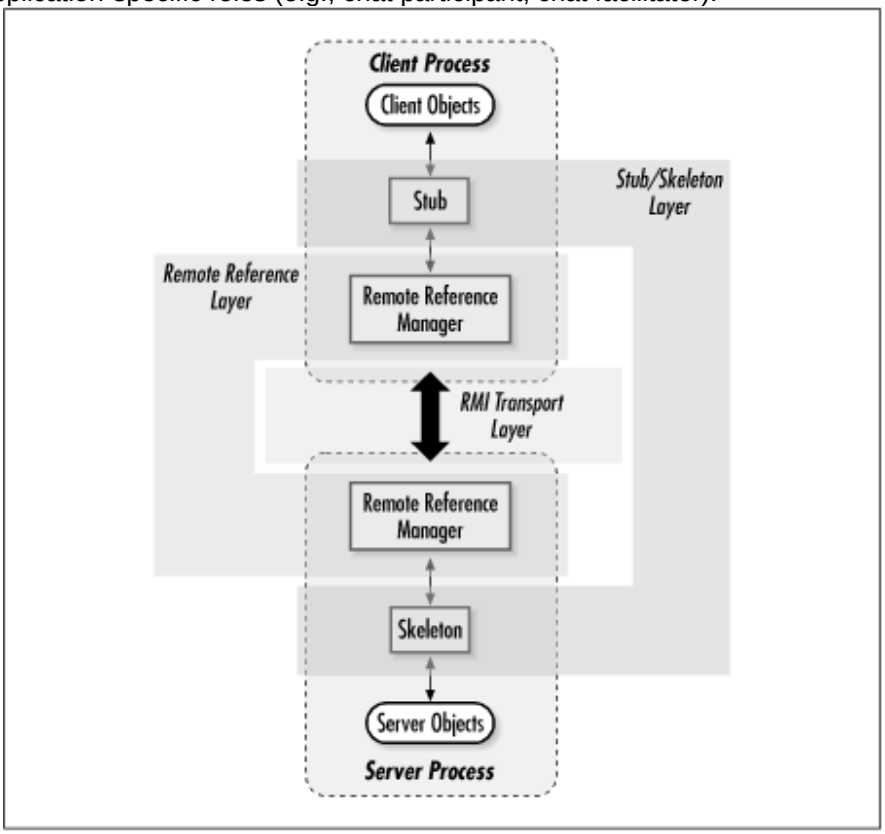

Figure 3-1. The RMI runtime architecture

- As you can see in Figure 3-1, a client makes a request of a remote object using a client-side stub; the server object receives this request from a server-side object skeleton. A client initiates a remote method 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 by *marshalling* the method arguments into serialized form and asking the remote reference layer to forward the method request and arguments to the appropriate remote object. Marshalling involves converting local objects into portable form so that they can be transmitted to a remote process. Each 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, the argument is serialized into bytes that are sent to the remote host and reconstituted into a copy of the local object. If the argument is neither Remote nor Serializable, the stub throws a java.rmi.MarshalException back to the client.

If the marshalling of method arguments succeeds, the client-side remote reference layer receives the remote reference and marshaled arguments from the stub. This layer converts the client request into low-level RMI transport requests according to the type of remote object communication being used. In RMI, remote objects can (potentially) run under several different communication styles, such as point-to-point object references, replicated objects, or multicast objects. The remote reference layer is responsible for knowing which communication style is in effect for a given remote object and generating the corresponding transport-level requests. In the current version of RMI (Version 1.2 of Java 2), the only communication style provided out of the box is point-to-point object references, so this is the only style we'll discuss in this chapter. For a point-to-point communication, the remote reference layer constructs a single network-level request and sends it over the wire to the sole remote object that corresponds to the remote reference passed along with the request.

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. The skeleton converts the remote request into the appropriate method call on the actual server object, which involves *unmarshalling* the method arguments into the server environment and passing them to the server object. As you might expect, unmarshalling is the inverse procedure to the marshalling process on the client. 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.

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. This result is sent back using the appropriate transport protocol, 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 Object Services

On top of its remote object architecture, RMI provides some basic object services you can use in your distributed application. These include an object naming/registry service, a remote object activation service, and distributed garbage collection.

Naming/registry service

- When a server process wants to export some RMI-based service to clients, it does so by registering one or more RMI-enabled objects with its local RMI registry (represented by the Registry interface).
- Each object is registered with a name clients can use to reference it.
- A client can obtain a stub reference to the remote object by asking for the object by name through the Naming interface.
- The Naming.lookup( ) method takes the fully qualified name of a remote object and locates the object on the network.
- The object's fully qualified name is in a URL-like syntax that includes the name of the object's host and the object's registered name.
- It's important to note that, although the Naming interface is a default naming service provided with RMI,

the RMI registry can be tied into other naming services by vendors.

- Once the lookup( ) method locates the object's host, it consults the RMI registry on that host and asks for the object by name.
- If the registry finds the object, it generates a remote reference to the object and delivers it to the client process, where it is converted into a stub reference that is returned to the caller.
- Once the client has a remote reference to the server object, communication between the client and the server commences as described earlier.
- We'll talk in more detail about the Naming and Registry interfaces later in this chapter.

Object activation service

- The remote object activation service is new to RMI as of Version 1.2 of the Java 2 platform.
- It provides a way for server objects to be started on an as-needed basis.
- Without remote activation, a server object has to be registered with the RMI registry service from within a running Java virtual machine.
- A remote object registered this way is only available during the lifetime of the Java VM that registered it.
- If the server VM halts or crashes for some reason, the server object becomes unavailable and any existing client references to the object become invalid.
- Any further attempts by clients to call methods through these now-invalid references result in RMI exceptions being thrown back to the client.
- The RMI activation service provides a way for a server object to be activated automatically when a client requests it.
- This involves creating the server object within a new or existing virtual machine and obtaining a reference to this newly created object for the client that caused the activation.
- A server object that wants to be activated automatically needs to register an activation method with the RMI activation daemon running on its host.
- We'll discuss the RMI activation service in more detail later in the chapter.

Distributed garbage collection

- The last of the remote object services, distributed garbage collection, is a fairly automatic process that you as an application developer should never have to worry about.
- Every server that contains RMI-exported objects automatically maintains a list of remote references to the objects it serves.
- Each client that requests and receives a reference to a remote object, either explicitly through the registry/naming service or implicitly as the result of a remote method call, is issued this remote object reference through the remote reference layer of the object's host process.
- The reference layer automatically keeps a record of this reference in the form of an expirable "lease" on the object.
- When the client is done with the reference and allows the remote stub to go out of scope, or when the lease on the object expires, the reference layer on the host automatically deletes the record of the remote reference and notifies the client's reference layer that this remote reference has expired.
- The concept of expirable leases, as opposed to strict on/off references, is used to deal with situations where a client-side failure or a network failure keeps the client from notifying the server that it is done with its reference to an object.
- When an object has no further remote references recorded in the remote reference layer, it becomes a candidate for garbage collection.
- If there are also no further local references to the object (this reference list is kept by the Java VM itself as part of its normal garbage-collection algorithm), the object is marked as garbage and picked up by the next run of the system garbage collector.

Sample Code

//AddServer.java

```
import java.rmi.*;

public interface AddServer extends Remote
{
	public int AddNumbers(int fno,int sno) throws RemoteException;
}
```

//AddServerImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject implements AddServer
{

	public AddServerImpl() throws RemoteException
	{
		super();
	}

	public int AddNumbers(int fno,int sno) throws RemoteException
	{

			return fno+sno;
	}
}
```

//RmiServer.java

```
import java.rmi.*;
import java.net.*;

public class RmiServer
{
	public static void main(String args[]) throws RemoteException, MalformedURLException
	{
		AddServerImpl add=new AddServerImpl();
		Naming.rebind("addnumbers",add);
	}
}
```

//RmiClient.java

```
import java.rmi.*;
import java.net.*;
public class RmiClient
{
	public static void main(String args[]) throws RemoteException, MalformedURLException,
NotBoundException
	{
		String url="rmi://127.0.0.1/addnumbers";
		AddServer add;
		add=(AddServer)Naming.lookup(url);
		int result =add.AddNumbers(10,5);
		System.out.println(result);
	}
```

}

Compiling RMI

Compile Remote object:

javac AddServer.Java

Compile Remote Object Implementation:

javac AddServerImpl.Java

- We end up with two Java files and two class files.
- We are now going to create our stub and skeleton.
- For creating the stub and skeleton files we have to use the rimc compiler on the remote object implementation file.

  rmic AddServerImpl
- After following the above step you will see that two newer class files have been created.
- They are addServerImpl_Stub.class (the stub which will reside on the client side and AddServerImpl_skel.class (the skeleton which will reside on the server side).

Compile RMIServer.java as well as RMIClient.java

Compile RMIServer.java

javac RMIServer.Java

Compile RMIClient:

javac RMIClient.Java

- Start RMI Registry server
  start rmiregistry
- Run RMIServer First and then RMIClient.
  Run->Cmd <enter>
  Go to the folder where you store java files
  java RmiServer
  Run->cmd <enter>
  Go to the folder where you store java files
  Java RmiClient