**I.Java Remote method Invocation**

**a) Remote Method Invocation Concept.**

**b)ServerSide.**

**c)Client Side.**

**II.Java Interface Definition Language and Corba**

**a)The Concept of Object request Brokerage.**

**b)Java IDL and CORBA.**

**c)The IDL Interface.**

**d) The Client Side.**

**f) The server side.**

**III. Enterprise JavaBeans**

**a)Enterprise Java Beans.**

**b)Deployment Descriptors**

**c)Entity Java Bean.**

**d)Session Java Bean.**

**e) Message-Driven Bean.**

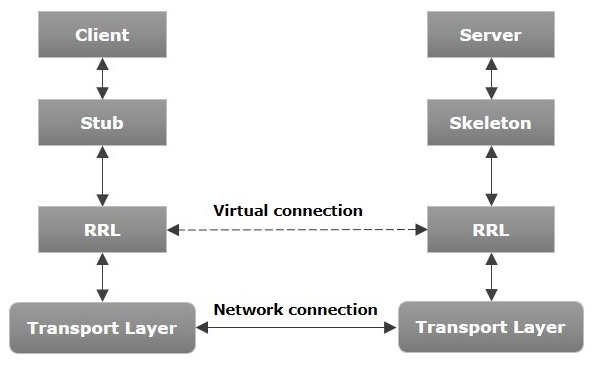
**f) The JAR File.**

**JAVAREMOTE METHOD INVOCATION**

a). REMOTE METHOD INVOCATION CONCEPT:- The RMI is used to connect together a client and server. A client is an application that requires services of object to fulfil a request. A server creates an object and makes object available to clients. A client contacts server to reference and invoke object by using RMI.

A RMI is used to build distributive applications.

Note:- The Client and Server side objects both are written in JAVA.



a.1)Working of RMI Application:-

* When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL. The stub is (Proxy)representation of object at

client side.

* The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
* The result is passed all the way back to the client.

1.2)RMI Process:- There are 3 steps necessary to make an object available to remote object.

Step1:- Design an Object.

Step2:- Compile the object.

Step3:-Make object accessible to remote clients over network.

1.2.1)Design an object:- Besides defining business logic of an object, the developer must define the remote interface for object.

Remote Interface:-

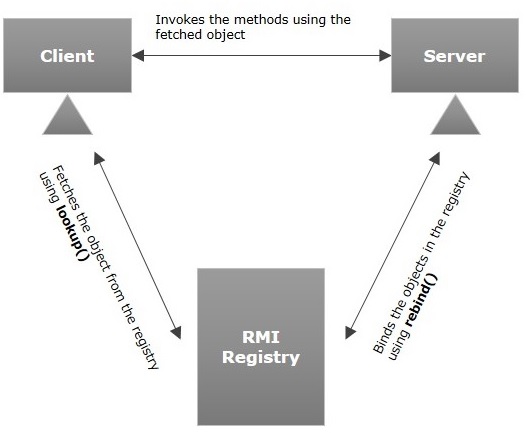
* The remote interface is used by clients to interact with the object using RMI communications.
* The remote interface extends java.rmi.Remote.
* The server-side objects that are invoked by remote clients must implement remote

interface and associated method definitions.

1.2.2)Compile the object:- Compilation of object is 2 step process that begins by compiling the object using javaccompiler. Once object is compiled, you must create stub for the object. This is done by calling RMI compiler called rmic.

1.2.3)The Compiled object is then made accessible over network by loading object to server.RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using **bind()** or **reBind()** methods). These are registered using a unique name known as **bind name**.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using **lookup()** method).



b)Server side:-

2.1) Define the remote Interface:-

Import java.rmi.Remote;

Import java.rmi.RemoteException;

Public interface myApplication extents Remote{

String myMethod() throws RemoteException;

}

2.2) Design the Object:-

Import java.rmi.\*;

Import java.rmi.server.\*;

Public class myApplicationServer extends UnicastRemoteObject implements myApplication

{

Public myApplicationServer() throws RemoteException

{ super();}

Public String myMethod()

{

Return “ I am suku\n”;

}

Public static void main(String arg[])

{

If(System.getSecutiryManager()==null)

{

System.setSecurityManager(new RMISecirityManager());

}

String app=”//localhost/myApplicaiton”;

Try{

MyApplicationServer server=new myApplicationServer();

Naming.rebind(app,server);

}

Catch(Exception error)

{

System.err.println(“Server Exception:”+ error.getMessage());

}

}

}

The server program has to create and install security manager. A security manager serves as firewall. RMI requires that server-side application install security manager.

RMISecurityManager() method creates a security manager object and setSecurityManager() method associates server-side program with security manager object.

Then rebind() method registers the remote object with RMI remote object registery.

C) Client Side:- The client–side program calls remote object which returns a string object that client side program displays.

Import java.rmi.\*;

Public class myApplicationClient{

Public static void main(String args[])

{

If(System.getSecurityManager()==null)

{

System.setSecurityManager(new RMISecurityManager());

}

Try

{

String app=”//localhost/myApplication”;

myApplication map=(myApplication)Naming.lookup(app);

System.out.println(mApp.myMethod());

}

Catch(Exception error)

{

System.err.println(“myApplicationClient exception:”+error.getMessage());

}

}

}

The lookup method is used to locate the remote object.

Execution steps to RMI Applications:-

------------------------------------------------

1.In single system, let us assume that we can open two command prompts. One command prompt is called server while the other one is called Client.

Step1: Compile the remoteInterface,serverprogram and client program at server

C:\javac myApplication.java

C:\javac myApplicationServer.java

C:\javac myApplicationClient.java

Step2: Compile the remoteInterface,serverprogram and client program at client

C:\javac myApplicationClient.java

Step3:

3.1)Compile remoteinterface using ‘rmic’ compiler at server.

3.2)After type ‘start rmiregistry’ and enter, then one blank window is open minimize that window.

3.3) Start the execution of MyApplicationServer program

C:\javac myApplicationServer.java

C:\javac myApplicationClient.java

C:\rmic myApplication

C:\ start rmiregistry

C:\java myApplicationServer

Step4:- run the myApplicationClient at client side.

C:\javac myApplicationClient.java

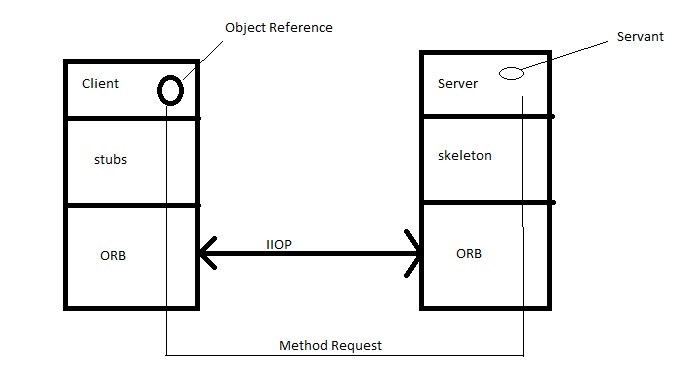
C:\java myApplicationClient

Sukumar.

**JAVA INTERFACE DEFINITION LANGUAGE AND CORBA**

a)Concept of Object Request Brokerage:-corba (common object request broker architecture) is standard distributed object architecture. The OMG-specified interface definition language(IDL) is used to define interfaces to CORBA Objects. The CORBA allows clients to use remote interface to access server-side components. The CORBA objects are always in server.

The ORB(Object Request Broker) is central component of CORBA.



A client has no knowledge of CORBA object location ,implementation details, nor which ORB is used to access the object. The client send request to ORB. The Request includes stub method of method. A stub method is alias for actual method. The ORB which is in client machine identify machine that runs the CORBA object. It asks the server machine ORB for connection. When connection is established , the ORB translates request into method call. It send it to skeleton. The skeleton code transforms method call into required implementation-specific format and calls object. Any result or exception are returned along the same path. The result arrives the client.

b)JAVA IDL and CORBA:- The clients interface to remote object is defined using (IDL) Interface definition language. The definition is placed in interface definition file(.idl), which is then compiled using idlj compiler that translates IDL interface into Java interface and generates other files that contain class code for stubs and skeleton code .

**c)IDL Interface**:-

Syntax to define module in IDL.

----------------

-------------------------

Module module\_name

{

Interface interface\_name

{

----

};

};

Example:- module myApplication

{

Interface myInteface

{

String mymethod();

}

}

Save the module as myApplication.idl & then compile by entering.

Idlj -fall myApplication.idl

The idlj compiler creates called myApplication and generates number of files.

1. myApplicationPOA.java:- This is abstract class that contains server skeleton and is used to provide CORBA functionality to server.

2. myApplicationStub.java:- This is stub used by client and implements myApplication.java interface.

3. myApplication.java:- This is java interface of IDL interface.

4. myApplicationHelper.java:- This is final class casts CORBA object references to their proper types.

5. myApplicationHolder.java:- This final class provides public instance member for out and inout arguments.

6.myApplicationOperations.java:- stubs & skeleton share this file. This file contains mapping of operations that are defined in IDL interface.

d)Client Side:-

Import myApplication.\*;

Import org.omg.CostNaming.\*;

Import org.omg.CostNaming.NamingContextPackage.\*;

Import org.omg.CORBA.\*;

Public class myClient

{

Try{

ORB

orb=ORB.init(args,null);

Org.omg.CORBA.Object nc1= orb.resolve\_initial\_references(“NameService”);

NamingContextExt nc2=NamingContextExtHelper.narrow(nc1);

myInterface myInt=myInterfaceHelper.narrow(nc2.resolve\_str(“myInterface”));

System.out.println(myInt.myMethod());

myInt.shutdown();

}

Catch(Exception error)

{

System.out.println(“ERRor:”+error.getMessage());

}

}

}

* The org.omg.CostNaming.\* package contains naming services used by class to call CORBA objects.
* The org.omg.CORBA.\* is package that contains CORBA classes.
* The first task of application is to create and initialize a local ORB object. It handles the IIOP operations.
* The init() method is passed any command line arguments. The arguments are IP address and port number for ORB service.
* The program calls orb.resolve\_initial\_references() method . This method receives the “NameService” as argument. It returns the “naming context object”. The Naming context object is reference for name service.
* This reference is CORBA object reference that must be narrowed to specific proper type by calling narrow() method, which is passed the obj reference and returns an instance of “Naming context”.
* Inorder for the program to receive a local reference to the myApplication object, the program needs to call resolve\_str() method.This method returns generic corba object and must be narrowed to myApplication object by calling narrow() method of myApplicationHelper object.

e)Server side:-

Import myApplication.\*;

Import org.omg.CostNaming.\*;

Import org.omg.CostNaming.NamingContextPackage.\*;

Import org.omg.CORBA.\*;

Import org.omg.PortableServer.\*;

Import org.omg.PortableServer.POA.\*;

Import java.util.Properties.\*;

Class MyApplication extends myApplicationPOA{

Private ORB orb;

Public void setORB(ORB v)

{ orb=v;}

Public String myMethod()

{

Return “I am here”;

}

}

Public class myServer{

Public static void main(String arg[])

{

Try{

ORB orb=ORB.init(args,null);

POA rpoa=POAHelper.narrow(orb.resolve\_initial\_references(“RootPOA”));

Rpoa.the\_POAManager().active();

MyApplication myApp=new MyApplication();

myApp.setORB(orb);

org.omg.CORBA.Object rf=rpoa.servant\_to\_reference(myApp);

myInterface hrf=myInterfaceHelper.narrow(rf);

org.omg.CORBA.Object orf=orb.resolve\_initial\_references(“NameService”);

NamingContextExt ncrf=NamingContextExtHelper.narrow(orf);

NameComponentpath[]=ncrf.to\_name(“myInterface”);

Ncrf.rebind(path,hrf);

System.out.println(“my server : Operational”);

Orb.run();

}

Catch (Exception error) {

System.err.printn( “Error:”+ error.getMessage());

}

System.out.println(“myServer:Terminated”);

}

}

1. The program creates the Servant subclass called MyApplicaiton that extends myApplicationPOA. The sub class Inherits CORBA functionality. Two methods are defined in Servant subclass. These are

setORB() :- It is used to initialize the private variable called orb.

myMethod():- It returns the simple string.

1. The program then defines the server class. The first step with in server class definition is to create a local ORB object by calling the init() method, which returns an ORB object called orb. The init() method is passed command-line arguments enabling the IP address and port to be set at runtime.
2. Next, the program obtains reference to NameService by calling the resolve\_intial\_references() method, the results of which are passed to narrow method of POAHelper to obtain reference to the root Portable Object Adapter(POA). The resolve\_initial\_references() returns a generic CORBA object that must be narrowed to its proper type by using narrow() method.
3. The activae() method is called next to change the state of the POA manager to active.
4. The program then creates servant called myApps. The servant is then passed to setORB() method of subclass.
5. The program needs to obtain an object reference to the servant. This is accomplished by callingservant\_to\_reference() method and narrow() method.
6. The program has object reference to servant, the servant is not available yet to clients. Reference to servant must be published using Common Object Services (COS) Naming Service.

To do this, the program must obtain the reference to “nameservice”. The program calls orb.resolve\_initial\_references() method . This method receives the “NameService” as argument. It returns the “naming context object”. The Naming context object is reference for name service.

1. The rebind() method requires an array of NameComponents, called path in above example. The path contains reference to the servant object. However, the Servant object is identified by a string called ‘myInterface’. The program must convert the string to reference using to\_name() method, which is then assigned to NameComponent array.

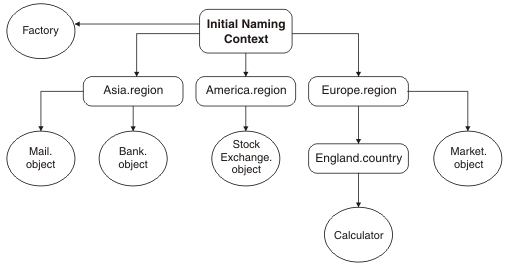
The rebind() method can then be called to make the servant available to clients.

1. Finally, program calls the run() method to start server-side program and wait for client request.

Naming Service:-A CORBA naming service holds CORBA object references. A CORBA server puts references to CORBA objects inside a naming service so that clients can query the naming service and obtain the object reference, then call operations on the CORBA objects. Typically, a client queries the naming service once, then caches the object reference. Each object in a naming service has a unique name. You must use this name when you configure the Object reference name property on the CORBARequest node.

Example to Naming Service.

---------------------------------



| **Object** | **Object reference name** |
| --- | --- |
| Factory | Factory |
| Bank | Asia.region/Bank.object |
| Mail | Asia.region/Mail.object |
| StockExchange | America.region/StrockExchange.object |
| Market | Europe.region/Market.object |
| Calculator | Europe.region/England.country/Calculator |

**Enterprise JavaBeans**

1. EnterPrise javaBeans:- Let’s say the user of j2EE application needs to display a catalog page that contains merchandise information. The j2EE application displays a web page that contains an HTML form where the user can enter the identification number of merchandise. The browser then calls JSP program. The JSP program parses product number of merchandise from query string sent by the browser**. The product number is then passed to an EJB, which interacts with database to extract information about merchandise. This information is returned to JSP program**, which formats the information into web page and sends web page to browser for display.

An Enterprise java bean is component of J2EE architecture that primarily provides business logic to JEEE application.

a.1)**EJB Container**:- The EJB container is vendor-provided entity located on the EJB server that manages system-level services for EJB. Each EJB must be installed in an EJB container. Typically, an EJB container contains many EJB’s. There can also be multiple EJB containers. However, an EJB must be installed in only one EJB container.

a.2)**EJB Classes**:- There are 3 kinds of EJB types.

a)Entity bean:- It is used to represent business data.

b)session bean:- A session bean is used to model the business process.

c)Message-Driven bean:- A message-driven bean is used to receive messages from JMS source-queue.

a.3)**EJB Interfaces**:- The EJB container handles communications between the EJB and other components in EJB environment using Home interface and Remote interface.

* The local clients that are on same JVM as EJB interact with the EJB

using Home interface.

Example:-

------------

Import javax.ejb.\*;

Import java.rmi.RemoteException;

Public interface MyEJBhome extends EJBHome

{

Public MyEJB create() throws CreateException,RemoteException;

}

In this example , home interface name is MyEJBhome. The create() method is defined inside the EJB.

* The Remote interface is used by remote clients that are capable of accessing EJB container from application that is compliant with RMI and IIOP.

Example:-

-------------

Import java.rmi.RemoteException;

Import javax.ejb.\*;

Public interface MyEJB extends EJBObject{

{

Public String myMethod() throws RemoteException;

}

In this example remote interface name is EJBObject. The mymethod signature is identical in the remote and bean class except that remote interface declares RemoteException.

->Local and LocalHome are part of EJB 2.0 specification and are used to access EJB with in same container.

b)Deployment Descriptors:- A deployment descriptor describes how EJBs are managed at runtime. The behaviour of EJB can be modified with in the deployment descriptor without having to modify the EJB class or EJB interfaces.

A deployment descriptor is written in file using XML syntax.

The deployment descriptor beigns with <!DOCTYPE> element that defines the URL for document type definition(DTD). The DTD defines structure of XML document.

<ejb-jar> element , which is root element of deployment descriptor. The <ejb-jar> contains the several sub-elements-some of which are required,while others are optional. The <enterprise-beans> element is required element.

The <enterprise-beans> contains own set of subelements, three of which are <session>,<entity> and <message-driven> sub elements. Each of these used to describe the specific type of EJB.

1.<ejb-name></ejb-name>:- This element contains descriptive name for the entity EJB.

2.<home></home>:- This element must contains fully qualified class name of remote home interface.

3.<remote></remote>:-This element describes fully qualified class name of the remote interface, which defines entity EJB’s business methods to remote clients.

4.<local-home></local-home>:- This element is used to describe the fully qualified class name for local home interface.

5.<local></local>:- This element describes the fully qualified class name of the local interface.

6.<ejb-class></ejb-class>:- This element describes fully qualified class name of EJB class, which implements the entity EJB’s business methods.

7.<persistence-type>:- This element contains value either ‘container’ or ‘entity’. It defines how entity EJB manages persistence.

8.<prim-key-class>:- This element describes fully qualified class name of entity ejb’s primary key.

Example: Deployment descriptor for EJB 2.0

<!DOCTYPE ebj-jar PUBLIC <http://java/sun/com/dtd/ejb-jar_2_0/dtd>>

<ejb-jar>

<enterprise-beans>

<entity>

<ejb-name>MyEJB</ejb-name>

<home>com.jimkeogh.ejb.myEJBHome</home>

<remote>com.jimkeogh.ejb.MyEJBRemote</home>

<local-home>com.jimkeogh.ejb.MyEJBLocal</local-home>

<local>com.jimkeogh.ejb.MyEJBLocal</local>

<ejb-class>com.jimkeogh.ejb.MyEJB</ejb-class>

<persistence-type>container</persistence-type>

<prim-key-class>java.lang.String</prim-key-class>

</entity>

</enterprise-beans>

</ejb-jar>

d) Session Java Bean:- A session bean is a java class. The session bean contains business logic. A session bean can be stateless or stateful.

d.1) Stateless Vs Stateful: Before creating a session bean, you must determine if the session bean should be stateless or stateful. Stateless session beans are generally more resource efficient than stateful session bean because stateless session bean can be used by EJB container to service another request instead of waiting for the next method call.

A stateless session bean is shared amongst clients. In contrast, a stateful session bean is dedicated to client.

Stateless session bean utilizes resources more efficiently than a stateful session bean.

d.2) Creating a Session Java bean:- A session bean is java class that implements the session bean interface. The session bean interface requires that five methods be defined in session bean class. These are

a)ejbActive():- It is called whenever the session bean is removed from the pool. You should place routines in this method that reacquire resources that were released when session bean entered the passive state.

b) ejbPassive() :- It is called before instance enters the “passive” state . This method contains routines that release resources.

c)ejbRemove():- It is called just before the bean is available for garbage collection.

d)setSessionContext(SessionContext):- It is called before any business methods are called.

e)ejbCreate():- It is used to initialize fields of EJB instances.

Example:

-------------

Import javax.ejb.\*;

Public class MyEJBBean implements SessionBean

{

Public void ejbActive()

{

S.o.p(“called ejbActive”);

}

Public void ejbRemove()

{

S.o.p(“Called ejbRemove”);

}

Public void ejbPassive()

{

S.o.p(“called ejb passive()”);

}

Public void setSessionContext(SessionContext ctx)

{

S.o.p(“called setsession context”);

}

Public void ejbCreate()

{

S.o.p(“called ejbcreate”);

}

Public String myMethod()

{

Return(“called mymethod()”);

}

}

F) The JAR File:- EjB clases and related files are packaged together into java archive(JAR) file for deployment. The JAR file is compressed file format that was originally designed to reduce size of software so it could be easily be transported.

The jar file used to package an EJB must contain the following:

EJB classes.

Dependent classes.

Remote Interface.

Home interface.

Dependent Interfaces.

Deployment Descriptor.

You can use JAR utility by entering jar cf followed by name of JAR file, and then followed by path and names of files that wil be archieved.