This will cover RMI and CORBA + chapters of "Advanced Java Programming" –Unit: 8(BSc. CSIT, TU)

Remote Method Invocation (RMI)

RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

Remote Method: A method of a Java program placed on any other system (not in your own system) in the world.

Invocation: Sending a request to the other system (say, a server) to execute the method (say, to call as remote method) available on it by passing sufficient parameters required for the execution of the method and asking to return the return value of the method.

Finally, RMI is nothing but communication between two JVMs loaded on two different systems.

RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

Let's take an example of a **Bank ATM** operation. Suppose you are to ATM to know the account balance. You know the balance is available on the bank server and not on ATM machine (say, ATM client). When you swipe your card your account number goes into the machine and inform the machine that you need to know the balance by clicking the button. Now there are two values to be send to the server (we call these as parameters), one is **account number** and the other is **what the operation you need**.

Now imagine, there is a method on the server (we call remote method as it is not available on your local ATM system) which will take care of your need, say "**public double getBalance(int accountNumber, double retrieveBalance)**". Now RMI job is to take your information and send it as parameters to the remote method, execute the method and send back the account balance as return value to the ATM client to print the label.

This is exactly what RMI application does.

What is distributed communication?

RMI sort of communication is an example of distributed communication where systems are distributed across the globe. Here Internet is not used and where it differs with Servlet communication. The systems involved in communication are known as distributed systems and

programming is known as distributed computing. If required, RMI can be connected to Internet to communicate with Web servers.

A distributed application needs the following to do.

- The client should be able to locate the remote objects existing on remote server. For example, RMI uses a naming service given by Naming class to bind remote objects with the rmi registry.
- The client should be able to communicate with the remote objects. This is taken care by RMI runtime environment. RMI programming feels in such way the remote method invocation is as if invoking on a local system. Small difference comes in Server side program where binding is performed by the remote object with the RMI runtime mechanism through RMI registry.
- To load methods for objects used in invocation. RMI passes objects between client and server and loads methods as per need.

As it given an example of bank transaction, does RMI take care of transaction management also?

No, RMI job is communication only – sending parameters to the remote method and getting back the return value of the remote method to the client. To take care of transactions, takes the help of EJB (Enterprise Java Beans). The backbone of EJB is RMI. EJB uses RMI for communication and adds extra features like transaction management, security, atomicity, load balancing and logging etc.

RMI does not use browsers and HTTP protocol. It comes with its own protocol known as RMI protocol with a default port number 1099 (which you can configure of your own). The protocol involves RMI runtime environment, Remote references for remote objects, RMI registry and some naming service etc.

Features of RMI

- It is object-oriented and communication is between Java-to-Java objects distributed.
- It is multithreaded.
- All the features of Java are applicable to RMI.
- The extended Java features are: a) Distributed connectivity b) Connecting legacy systems. Legacy means that we do not know in advance what type of server it is going to be connected in the network.

- RMI is seamless integration. Seamless implies communication through an object and information about operating system where the object developed is not known to the programmer when he writes the code.
- RMI has different type of garbage collection known as Distributed garbage collection.
- In RMI, the server exports a object to the client by storing a reference with it. When the client communication is over or disconnects with the server, the reference of client is lost on the server remote object. If the reference of the client no more exists, the remote object on the server is eligible for garbage collection. This is known as **distributed garbage collection**

Architecture of an RMI Application

The RMI Architecture is very simple involving **a client program, a server program, a stub and skeleton.**

In RMI, the client and server do not communicate directly; instead communicates through stub and skeleton (a special concept of RMI and this is how designers achieved distributed computing in Java). They are nothing but special programs generated by RMI compiler. They can be treated as proxies for the client and server. Stub program resides on client side and skeleton program resides on server. That is, the client sends a method call with appropriate parameters to stub. The stub in turn calls the skeleton on the server. The skeleton passes the stub request to the server to execute the remote method. The return value of the method is sent to the skeleton by the server. The skeleton, as you expect, sends back to the stub. Finally the return value reaches client through stub.

Note: The method existing on the server is a remote method to the client (because client and server may be far away anywhere in the globe); but for the server, it is local method only. So, for the client, the server is remote server (okay, in server point of view, the client is remote client, but in RMI, we do not call client as remote client), a program on the remote server is remote program and finally the method in the remote program is remote method. Know the meaning of remote server, remote program, remote method; this must be very clear to the beginner else the RMI concept is very confusing at the outset. So, anything residing on server is remote to client.

In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

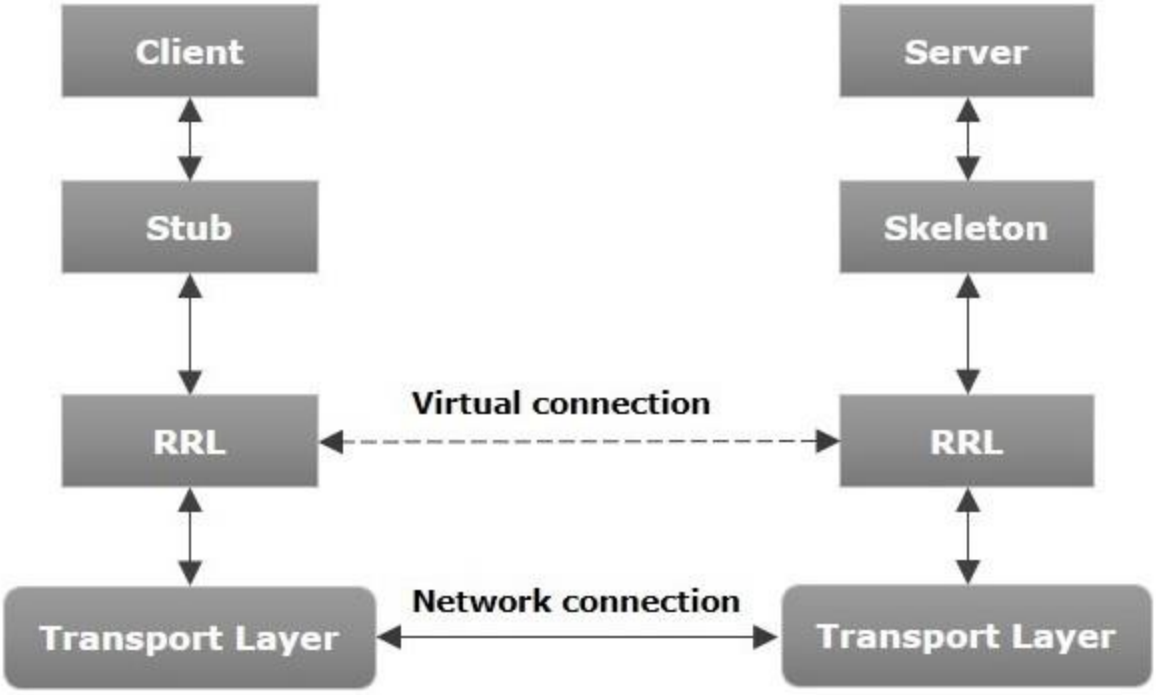


Figure 1: Architecture of RMI

Application Layer

This layer is nothing but the actual systems (client and server) involved in communication. A client Java program communicates with the other Java program on the server side. RMI is nothing but a communication between two JVMs placed on different systems.

Proxy Layer

The proxy layer consists of proxies (named as **stub** and **skeleton** by designers) for client and server. Stub is client side proxy and Skeleton is server side proxy. Stub and skeleton, as many people confuse, are not separate systems but they are after all programs placed on client and server. The stub and skeleton are not hard coded by the Programmer but they are generated by RMI compiler (this is shown in execution part later). Stub is placed on client side and skeleton is placed on server side. The client server communication goes through these proxies. Client sends its request of method invocation (to be executed on remote server) to stub. Stub in turn sends the request to skeleton. Skeleton passes the request to the server program. Server executes the method and sends the return value to the skeleton (to route to client). Skeleton sends to stub and stub to client program.

- **Stub** − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.

- **Skeleton** − This is the object which resides on the server side. Stub communicates with this skeleton to pass request to the remote object.

Remote Reference Layer (RRL)

Proxies are implicitly connected to RMI mechanism through Remote reference layer, the layer responsible for object communication and transfer of objects between client and server. It is responsible for dealing with semantics of remote invocations and implementation – specific tasks with remote objects. In this layer, actual implementation of communication protocols is handled.

It is the layer which manages the references made by the client to the remote object.

Transport Layer − This layer connects the client and the server. It manages the existing connection and also sets up new connections. Transport layer does not exist separately but is a part of Remote reference layer. Transport layer is responsible for actually setting up connections and handling the transport of data from one machine to another. It can be modified to handle encrypted streams, compression algorithms and a number of other security/performance related enhancements.

Marshaling and Unmarshaling

One more job of proxies is marshaling and unmarshaling. Marshaling is nothing but converting data into a special format suitable to pass through the distributed environment without losing object persistence. For this reason, the RMI mechanism implicitly serialize the objects involved in communication. The stub marshals the data of client and then sends to the skeleton. As the format is not understood by the server program, it is unmarshaled by the skeleton into the original format and passed to server. Similarly from server to client also.

A marshal stream includes a stream of objects that are used to transport parameters, exceptions, and errors needed for these streams for communicating with each other. Marshaling and unmarshaling are done by the RMI runtime mechanism implicitly without any programmers extra coding.

The marshaled stream is passed to RRL(Remote Reference Layer). Task of RRL is to identify the host, that is, remote machine. The marshaled stream is passed to Transport layer which performs routing.

Remote Objects

An object created on the server (by the server program) is known as remote object. To put technically, any object that implements **java.rmi.Remote** interface is known as remote object.

Objects with methods that can be invoked across JVMs are called remote objects

An object becomes remote by implementing a remote interface,which has the following characteristics:

- A remote interface extends the interface java.rmi.Remote
- Each method of the interface declares java.rmi.RemoteException in its throws clause

RMI treats a remote object differently from a non-remote object when the object is passed from one JVM to another JVM

- RMI passes a remote stub for a remote object to receiving JVM
- The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference
- The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object

Working of an RMI Application

The following points summarize how an RMI application works −

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called invoke() of the object remoteRef. It passes the request to the RRL on the server 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.

RMI Registry

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 RMI registry (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).

The following illustration explains the entire process −

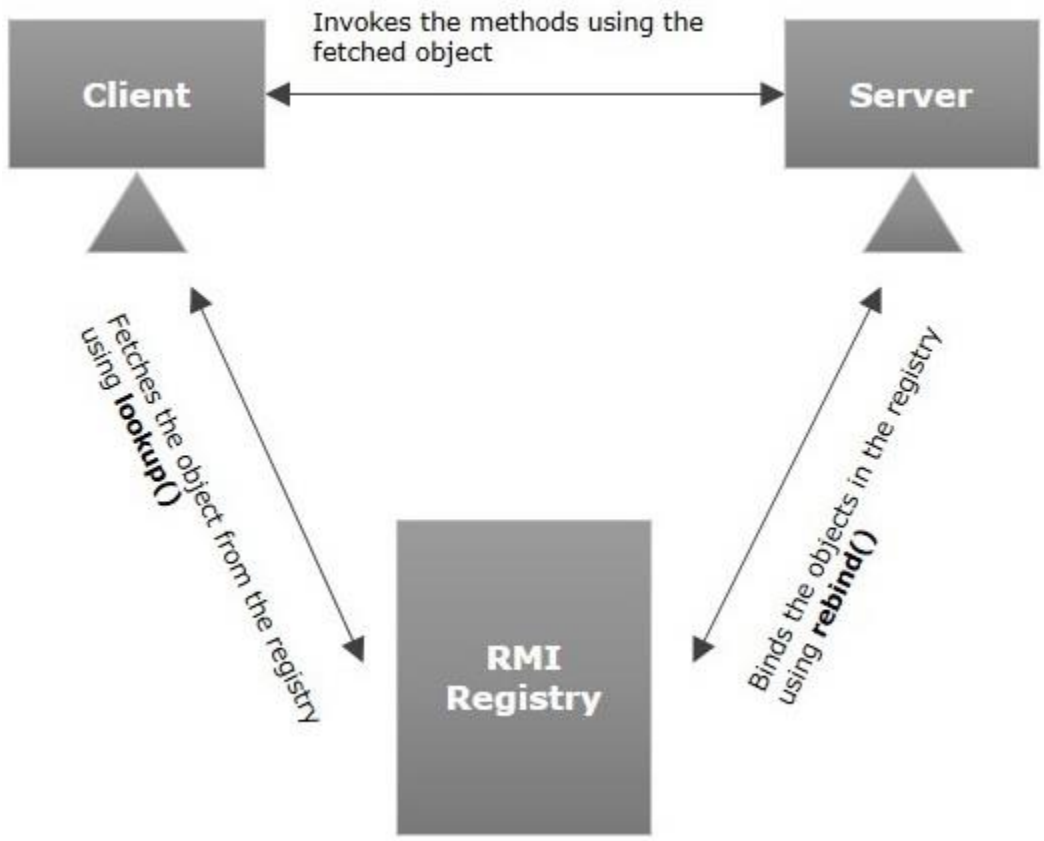


Figure 2: RMI Registry

Goals of RMI

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects.

Creating a simple RMI application

A typical RMI application includes four programs.

1. **Remote interface:** Write an interface that should extend an interface called *Remote* from *java.rmi* package.
2. **Implementation program:** Write a concrete class which extends *UnicastRemoteObject* and implements the above remote interface.
3. **Server Program:** Write a server program to register the object with RMI registry.
4. **Client program:** Write a client program to request for object and method invocation.

For communication, the other supporting programs are stub, skeleton and RMI registry.

RMI Example for adding two numbers

Define a remote interface

A remote interface specifies the methods that can be invoked remotely by a client. Clients program communicate to remote interfaces, not to classes implementing it. To be a remote interface, a interface must extend the **Remote** interface of **java.rmi** package.

In this Example , following is the code of **AdderInterface.java** interface

```
import java.rmi.*;
public interface AdderInterface extends Remote{
public int add(int x,int y)throws RemoteException;
}
```

Implementation of remote interface

For implementation of remote interface, a class must either extend **UnicastRemoteObject** or use **exportObject()** method of **UnicastRemoteObject** class

Following is the code for AdderClass.java file

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderClass extends UnicastRemoteObject implements AdderInterface{
AdderClass()throws RemoteException{
super();
}
public int add(int x,int y){
    return x+y;}
}
```

Create AdderServer and host rmi service

You need to create a server application and host rmi service AdderClass in it. This is done using **rebind()**method of **java.rmi.Naming** class. **rebind()** method take two arguments, first represent the name of the object reference and second argument is reference to instance of AdderClass

Following is the code inside **AdderServer.java** file

```
import java.rmi.*;
import java.rmi.registry.*;
public class AdderServer {
    public static void main(String args[]){
try{
    AdderInterface obj=new AdderClass();
    Naming.rebind("ADD",obj);//to rigister the name
    System.out.println("Server Started!");
//obj object is hosted with name ADD.

    }catch(Exception e){System.out.println(e);}
  }
}
```

Create client application

Client application contains a java program that invokes the **lookup()** method of the **Naming** class. This method accepts one argument, the rmi URL and returns a reference to an object of type **AdderInterface**. All remote method invocation is done on this object.

Following is the content of **Client.java** file

```
import java.rmi.*;
public class Client{
public static void main(String args[]){
try{
AdderInterface obj=(AdderInterface)Naming.lookup("ADD");
int res = obj.add(30,40);
System.out.println("The result of addition is : "+res);
}catch(Exception e){}
}
}
```

Steps to run this RMI application

- Save all the above java file into a directory and name it as "rmidemo"
- Compile all the java files

  ```
  javac *.java
  ```

  [Before using above command set the javac path]

  ```
  C:\Users\BIPIN\Desktop\rmidemo>set path=C:\Program Files\Java\jdk1.8.0_131\bin
  ```

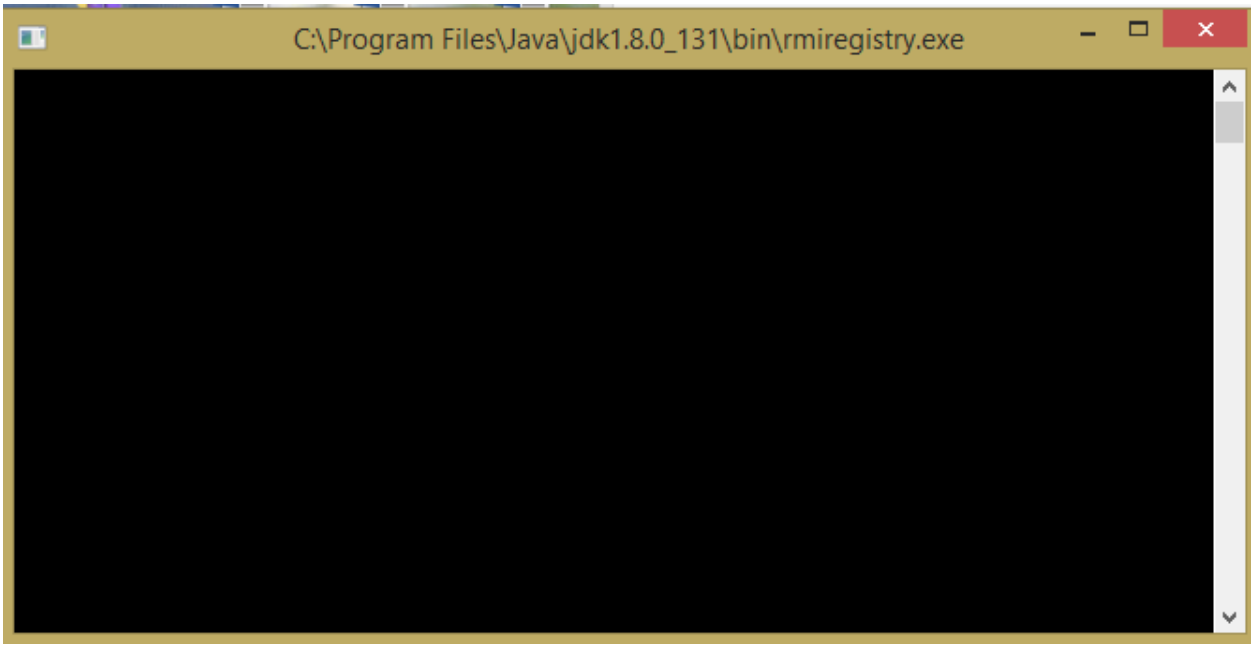
  ```
  C:\Users\BIPIN\Desktop\rmidemo>javac *.java
  ```

- Start RMI registry

  start rmiregistry

  ```
  C:\Users\BIPIN\Desktop\rmidemo>start rmiregistry
  ```

  You will see the new cmd like below:



- Run AdderServer file

```
C:\Users\BIPIN\Desktop\rmidemo>java AdderServer
Server Started!
```

- Run Client file in another command prompt

```
C:\Users\BIPIN\Desktop\rmidemo>set path=C:\Program Files\Java\jdk1.8.0_131\bin

C:\Users\BIPIN\Desktop\rmidemo>java Client
```

After running a client you will see the result like below

```
C:\Users\BIPIN\Desktop\rmidemo>java Client
The result of addition is : 70
```

RMI pros and cons

Remote method invocation has significant features that CORBA doesn't possess - most notably the ability to send new objects (code and data) across a network, and for foreign virtual machines to seamlessly handle the new object. Remote method invocation has been available since JDK 1.02, and so many developers are familiar with the way this technology works, and organizations may already have systems using RMI. **Its chief limitation, however, is that it is limited to Java Virtual Machines, and cannot interface with other languages.**

Pros	Cons
Portable across many platforms	Tied only to platforms with Java support
Can introduce new code to foreign JVMs	Security threats with remote code execution, and limitations on functionality enforced by security restrictions.
Java developers may already have experience with RMI (available since JDK1.02)	Learning curve for developers that have no RMI experience is comparable with CORBA
Existing systems may already use RMI - the cost and time to convert to a new technology may be prohibitive	Can only operate with Java systems - no support for legacy systems written in C++, Ada, Fortran, Cobol, and others (including future languages).

Introduction to CORBA

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) designed to facilitate the communication of systems that are deployed on diverse platforms. CORBA enables collaboration between systems on different operating systems, programming languages, and computing hardware. CORBA uses an object-oriented model although the systems that use the CORBA do not have to be object oriented. CORBA is an example of the distributed object paradigm

CORBA, or Common Object Request Broker Architecture, is a standard architecture for distributed object systems. It allows a distributed, heterogeneous collection of objects to interoperate.

The OMG

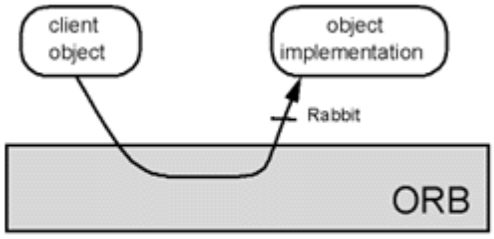
The Object Management Group (OMG) is responsible for defining CORBA. The OMG comprises over 700 companies and organizations, including almost all the major vendors and developers of distributed object technology, including platform, database, and application vendors as well as software tool and corporate developers.

CORBA Architecture

CORBA defines an architecture for distributed objects. The basic CORBA paradigm is that of a request for services of a distributed object. Everything else defined by the OMG is in terms of this basic paradigm.

The services that an object provides are given by its interface. Interfaces are defined in OMG's Interface Definition Language (IDL). Distributed objects are identified by object references, which are typed by IDL interfaces.

The figure below graphically depicts a request. A client holds an object reference to a distributed object. The object reference is typed by an interface. In the figure below the object reference is typed by the Rabbit interface. The Object Request Broker, or ORB, delivers the request to the object and returns any results to the client. In the figure, a jump request returns an object reference typed by the AnotherObject interface.




```
interface Rabbit {
        :
    AnotherObject jump(
        in long how_high);
        :
};
```

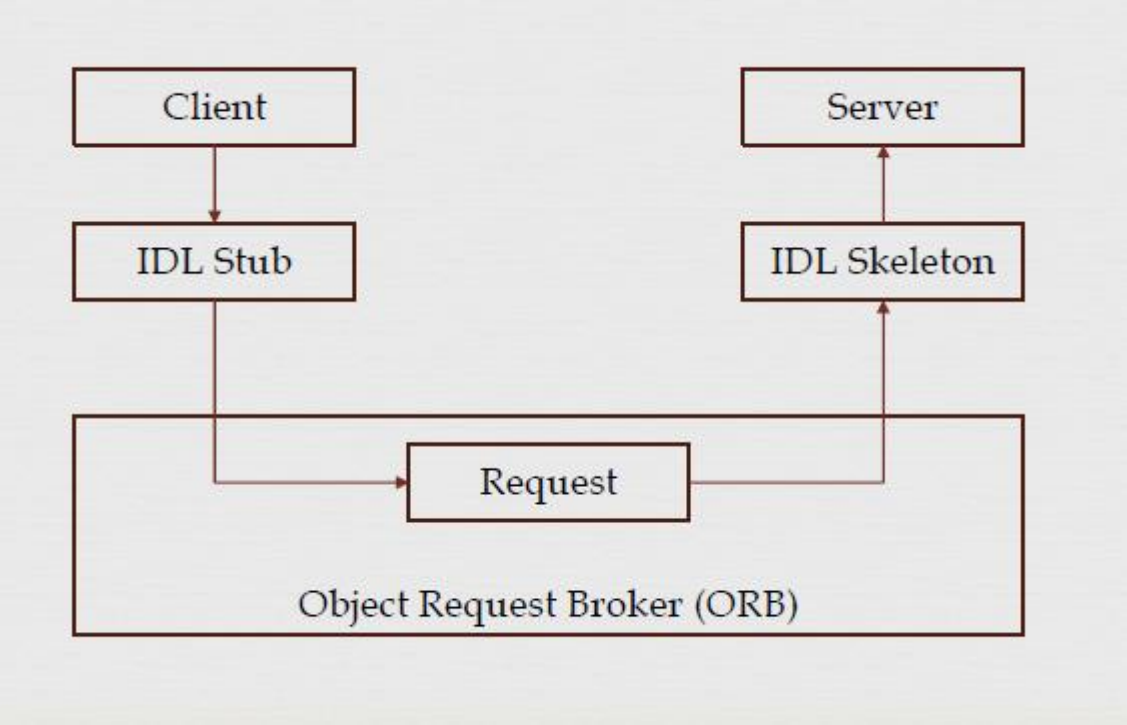


Figure 3: CORBA architecture

Components

- **Interface Definition Language (IDL):** used to define CORBA interfaces
- **Object Request Broker (ORB):** responsible for all interactions between remote objects and the applications that use them
- **The Portable Object Adaptor (POA):** responsible for object activation/deactivation, mapping object ids to actual object implementations.
- **Naming Service:** a standard service in CORBA that lets remote clients find remote objects on the networks
- **Inter-ORB Protocol (IIOP)**

IDL

CORBA uses an interface definition language (IDL) to specify the interfaces that objects present to the outer world.CORBA then specifies a mapping from IDL to a specific implementation language like C++ or Java. Standard mappings exist for Ada, C, C++, C++11, COBOL, Java, Lisp, PL/I, Object Pascal, Python, Ruby and Smalltalk. Non-standard mappings exist for C#, Erlang, Perl, Tcl and Visual Basic implemented by object request brokers (ORBs) written for those languages

ORB

The CORBA specification dictates there shall be an ORB through which an application would interact with other objects.

This is how it is implemented in practice

- The application simply initializes the ORB, and accesses an internal Object Adapter, which maintains things like reference counting, object (and reference) instantiation policies, and object lifetime policies
- The Object Adapter is used to register instances of the generated code classes
- Generated code classes are the result of compiling the user IDL code, which translates the high-level interface definition into an OS and language specific class base for use by the user application
- This step is necessary in order to enforce CORBA semantics and provide a clean user process for interfacing with the CORBA infrastructure

The ORB is the distributed service that implements the request to the remote object. It locates the remote object on the network, communicates the request to the object, waits for the results and when available communicates those results back to the client.

The ORB implements location transparency. Exactly the same request mechanism is used by the client and the CORBA object regardless of where the object is located. It might be in the same process with the client, down the hall or across the planet. The client cannot tell the difference.

The ORB implements programming language independence for the request. The client issuing the request can be written in a different programming language from the implementation of the CORBA object. The ORB does the necessary translation between programming languages. Language bindings are defined for all popular programming languages.

CORBA as a Standard for Distributed Objects

One of the goals of the CORBA specification is that clients and object implementations are portable. The CORBA specification defines an application programmer's interface (API) for clients of a distributed object as well as an API for the implementation of a CORBA object. This means that code written for one vendor's CORBA product could, with a minimum of effort, be rewritten to work with a different vendor's product. However, the reality of CORBA products on the market today is that CORBA clients are portable but object implementations need some rework to port from one CORBA product to another.

CORBA 2.0 added interoperability as a goal in the specification. In particular, CORBA 2.0 defines a network protocol, called IIOP (Internet Inter-ORB Protocol), that allows clients using a CORBA product from any vendor to communicate with objects using a CORBA product from

any other vendor. IIOP works across the Internet, or more precisely, across any TCP/IP implementation.

Interoperability is more important in a distributed system than portability. IIOP is used in other systems that do not even attempt to provide the CORBA API. In particular, IIOP is used as the transport protocol for a version of Java RMI (so called "RMI over IIOP"). Since EJB is defined in terms of RMI, it too can use IIOP. Various application servers available on the market use IIOP but do not expose the entire CORBA API. Because they all use IIOP, programs written to these different API's can interoperate with each other and with programs written to the CORBA API.

CORBA Services

Naming Service: Defines how CORBA objects can have friendly symbolic names

Transaction Service and Concurrency Control Service:

- Coordinates atomic access to CORBA objects
- Provides a locking service for CORBA objects in order to ensure serializable access

Persistent State Service: Objects can be stored in passive from in a persistent object store while they are not in use an can be activated when needed

Life Cycle Service: Defines how CORBA objects are created, removed, moved, and copied
Event Service: Decouples the communication between distributed objects

Notification Service: Extends event service**.** Adds:

- the ability to define notification as data structures
- ability to specify interested events using filters by event consumers and means of
- discovering event types
- ability to discover events that consumer are interested in by the suppliers

Security Service: Authentication, access control, security of communication etc.

CORBA pros and cons

CORBA is gaining strong support from developers, because of its ease of use, functionality, and portability across language and platform. CORBA is particularly important in large organizations, where many systems must interact with each other, and legacy systems can't yet be retired. CORBA provides the connection between one language and platform and another - its only limitation is that a language must have a CORBA implementation written for it. CORBA also appears to have a performance increase over RMI, which makes it an attractive option for systems that are accessed by users who require real-time interaction.

Pros	Cons
Services can be written in many different languages, executed on many different platforms, and accessed by any language with an interface definition language (IDL) mapping	Describing services require the use of an interface definition language (IDL) which must be learned. Implementing or using services require an IDL mapping to your required language - writing one for a language that isn't supported would take a large amount of work.
With IDL, the interface is clearly separated from implementation, and developers can create different implementations based on the same interface.	IDL to language mapping tools create code stubs based on the interface - some tools may not integrate new changes with existing code.
CORBA supports primitive data types, and a wide range of data structures, as parameters	CORBA does not support the transfer of objects, or code.
CORBA is ideally suited to use with legacy systems, and to ensure that applications written now will be accessible in the future.	The future is uncertain - if CORBA fails to achieve sufficient adoption by industry, then CORBA implementations become the legacy systems.
CORBA is an easy way to link objects and systems together.	Some training is still required, and CORBA specifications are still in a state of flux.
CORBA systems may offer greater performance.	Not all classes of applications need real-time performance, and speed may be traded off against ease of use for pure Java systems.

CORBA vs RMI

CORBA	RMI
Suitable for full range of distributed systems.	Suitable for small distributed application.
Has better scalability.	Not suitable where scalability is a major concern.
Client and server may be written in separate languages.	Client and server are both written in Java.
Describing a service requires the use of interface definition language (IDL).	IDL is not required.
Does not support transfer of objects.	Supports transfer of objects.
Heavily used in legacy enterprise systems.	Not used in legacy systems.
Difficult to learn and use.	Easy to learn and use.

References

1. javatpoint.com
2. way2java.com
3. studytonight.com
4. tutorialspoint.com