Remote Method Invocation(RMI)

The **Remote Method Invocation (RMI)** model represents a distributed object application. RMI allows an object inside a JVM (a client) to invoke a method on an object running on a remote JVM (a server) and have the results returned to the client.

- Therefore, RMI implies a client and a server.

The server application typically creates an object and makes it accessible remotely.

- Therefore, the object is referred to as a remote object.
- The server registers the object that is available to clients.

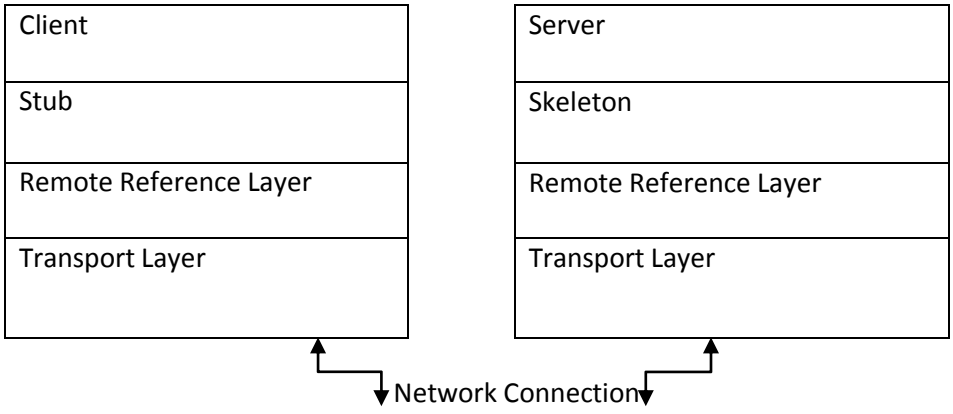
One of the ways this can be accomplished is through a naming facility provided as part of the JDK, which is called the **rmiregistry**. The server uses the registry to bind an arbitrary name to a remote object. A client application receives a reference to the object on the server and then invokes methods on it. The client looks up the name in the registry and obtains a reference to an object that is able to interface with the remote object. The reference is referred to as a remote object reference. Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

RMI Architecture

The interface that the client and server objects use to interact with each other is provided through **stubs/skeleton, remote reference, and transport layers**. Stubs and skeletons are Java objects that act as proxies to the client and server, respectively.

All the network-related code is placed in the stub and skeleton, so that the client and server will not have to deal with the network and sockets in their code. The remote reference layer handles the creation of and management of remote objects. The transport layer is the protocol that sends remote object requests across the network.

A simple diagram showing the above relationships is shown below.



Developing a distributed application using RMI involves the following steps:

1. Define a remote interface
2. Implement the remote interface
3. Develop the server
4. Develop a client
5. Generate Stubs and Skeletons, start the RMI registry, server, and client

The Remote Interface

The server's job is to accept requests from a client, perform some service, and then send the results back to the client.The server must specify **an interface that defines the methods available to clients as a service**. This remote interface defines the client view of the remote object.The remote interface is always written to **extend the java.rmi.Remote interface**. Remote is a "marker" interface that identifies interfaces whose methods may be invoked from a non-local virtual machine.

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

In the example above, add(int x,int y) is a remote method of the **remote interface** RemoteInterface. All methods defined in the remote interface are required to state that they **throw a RemoteException**. A RemoteException represents communication-related exceptions that may occur during the execution of a remote method call.

The Remote Object

An implementation of the RemoteInterface interface is shown below.

```
//ServerImplements.java
import java.rmi.*;
import java.rmi.server.*;
import java.lang.String;
public class ServerImplements extends UnicastRemoteObject implements RemoteInterface
 {
 public ServerImplements()throws RemoteException
 {
 super();
 }
 public int add(int x,int y)
 {
 return (x+y);
 }
}
```

The implementation is referred to as the **remote object**. The implementation class extends **UnicastRemoteObject** to link into the RMI system. This is not a requirement. A class that does not extend UnicastRemoteObject may use its exportObject() method to be linked into RMI. When a class extends UnicastRemoteObject, it **must provide a constructor declaring that it may throw a RemoteException object.** When this constructor calls super(), it activates code in UnicastRemoteObject, which performs the RMI linking and remote object initialization.

Writing the Server

```
//AdditionServer.java
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
```

```
public class  Server
{
 public static void main(String args[])
 {
 try
 {
 ServerImplements s=new ServerImplements();
 LocateRegistry.createRegistry(1099);
 Naming.rebind("SERVICE",s);
 System.out.println("Server Started ");
 }
 catch(Exception e)
 {
 System.out.println(e.getMessage());
 }
 }
}
```

The server creates the **remote object, registers it under some arbitrary name, then waits for remote requests.** The **java.rmi.registry.LocateRegistry** class allows the RMI registry service (provided as part of the JVM) to be started within the code by calling its createRegistry method.
This could have also been achieved by typing the following at a command prompt: **start rmiregistry.** The default port for RMI is 1099. The java.rmi.registry.Registry class provides two
methods for binding objects to the registry.
Naming.bind("ArbitraryName", remoteObj); throws an Exception if an object is already bound under the "ArbitrayName. "
Naming.rebind ("ArbitraryName", remoteObj); binds the object under the "ArbitraryName" if it does not exist or overwrites the object that is bound.
The example above acts as a server that creates a ServerImplements object and makes it available to clients by binding it under a name of **"SERVICE ".**

NOTE: If both the client and the server are running Java SE 5 or higher, no additional work is needed on the server side. Simply compile the **RemoteInterface.java**, **ServerImplements.java**, and **AdditionServer.java**, and the server can then be started. The reason for this is the introduction in Java SE 5 of dynamic generation of stub classes. Java SE 5 adds support for the dynamic generation of stub classes at runtime, eliminating the need to use the RMI stub compiler, **rmic**, to pre-generate stub classes for remote objects.
• Note that rmic must still be used to pre-generate stub classes for remote objects that need to support clients **running on earlier versions.**

Writing the Client

```
//Client.java
import java.rmi.*;
import java.io.*;
public class Client
{
public static void main(String args[])
 {
```

```
try
 {
String ip="rmi://127.0.0.1/SERVICE";
RemoteInterface s=
  (RemoteInterface)Naming.lookup(ip);
System.out.println("sum: "+ s.add(1,3));
 }
catch(Exception e)
 {
 System.out.println(e.getMessage());
 e.printStackTrace();
 }
 }
}
```

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).

Common Object Request Broker Architecture(CORBA)

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.

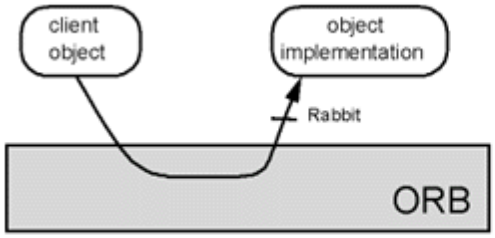
The OMG

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);
        :
};
```

The ORB

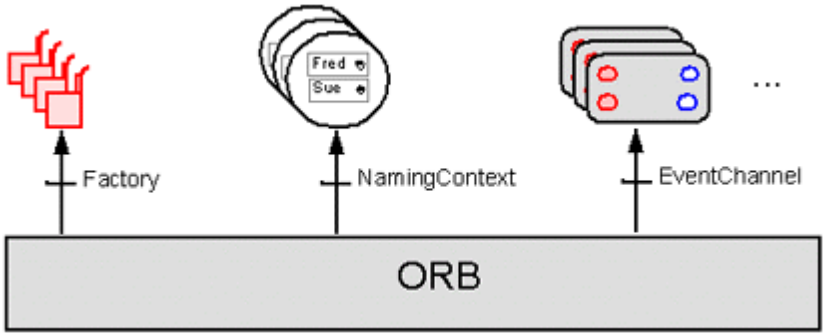
The ORB is the distributed service that implements the request to the remote object. It locates the remote object on the network, communicates the request to the object, waits for the results and when available communicates those results back to the client.
The ORB implements location transparency. Exactly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

Another important part of the CORBA standard is the definition of a set of distributed services to support the integration and interoperation of distributed objects. As depicted in the graphic below, the services, known as CORBA Services or COS, are defined on top of the ORB. That is, they are defined as standard CORBA objects with IDL interfaces, sometimes referred to as "Object Services."



There are several CORBA services. Below is a brief description of each:

Service	Description
Object life cycle	Defines how CORBA objects are created, removed, moved, and copied
Naming	Defines how CORBA objects can have friendly symbolic names
Events	Decouples the communication between distributed objects
Relationships	Provides arbitrary typed n-ary relationships between CORBA objects
Externalization	Coordinates the transformation of CORBA objects to and from external media
Transactions	Coordinates atomic access to CORBA objects
Concurrency Control	Provides a locking service for CORBA objects in order to ensure serializable access
Property	Supports the association of name-value pairs with CORBA objects
Trader	Supports the finding of CORBA objects based on properties describing the service offered by the object
Query	Supports queries on objects

CORBA Products

CORBA is a specification; it is a guide for implementing products. Several vendors provide CORBA products for various programming languages. The CORBA products that support the Java programming language include:

ORB	Description
The Java 2 ORB	The Java 2 ORB comes with Sun's Java 2 SDK. It is missing several features.
VisiBroker for Java	A popular Java ORB from Inprise Corporation. VisiBroker is also embedded in other products. For example, it is the ORB that is embedded in the Netscape Communicator browser.
OrbixWeb	A popular Java ORB from Iona Technologies.
WebSphere	A popular application server with an ORB from IBM.
Netscape Communicator	Netscape browsers have a version of VisiBroker embedded in them. Applets can issue request on CORBA objects without downloading ORB classes into the browser. They are already there.
Various free or shareware ORBs	CORBA implementations for various languages are available for download on the web from various sources.

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.