## RMI(Remote Method Invocation):

The RMI (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM. The RMI provides remote communication between the applications using two objects stub and skeleton. RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

A primary goal for the RMI designers was to allow programmers to develop distributed Java programs with the same syntax and semantics used for non-distributed programs. To do this, they had to carefully map how Java classes and objects work in a single Java Virtual Machine (JVM) to a new model of how classes and objects would work in a distributed (multiple JVM) computing environment.

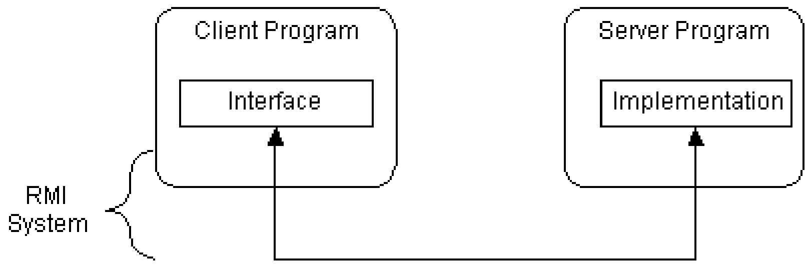
## Interface of RMI:

The RMI architecture is based on one important principle: the definition of behavior and the implementation of that behavior are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.

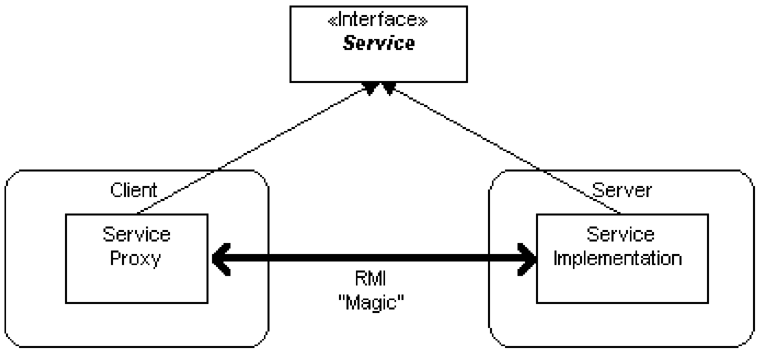
This fits nicely with the needs of a distributed system where clients are concerned about the definition of a service and servers are focused on providing the service.

Specifically, in RMI, the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class. Therefore, the key to understanding RMI is to remember that *interfaces define behavior* and *classes define implementation*.

While the following diagram illustrates this separation,



remember that a Java interface does not contain executable code. RMI supports two classes that implement the same interface. The first class is the implementation of the behavior, and it runs on the server. The second class acts as a proxy for the remote service and it runs on the client. This is shown in the following diagram.



A client program makes method calls on the proxy object, RMI sends the request to the remote JVM, and forwards it to the implementation. Any return values provided by the implementation are sent back to the proxy and then to the client's program.

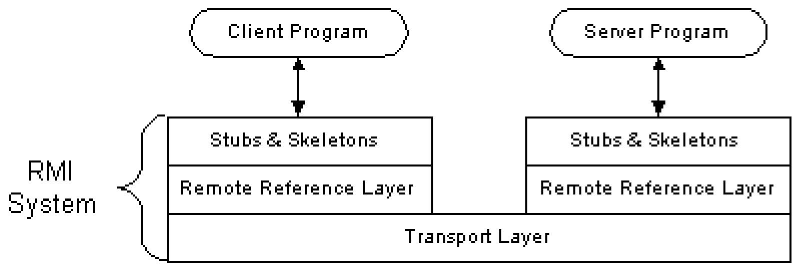
**RMI Architecture Layers:**

With an understanding of the high-level RMI architecture,

The RMI implementation is essentially built from three abstraction layers. The first is the Stub and Skeleton layer, which lies just beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

The next layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one (unicast) link. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via Remote Object Activation.

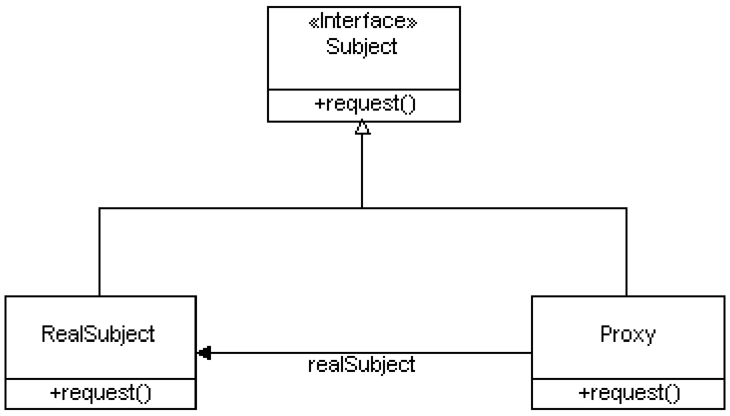
The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.



By using a layered architecture each of the layers could be enhanced or replaced without affecting the rest of the system. For example, the transport layer could be replaced by a UDP/IP layer without affecting the upper layers.

## 1. Stub and Skeleton Layer:

The stub and skeleton layer of RMI lie just beneath the view of the Java developer. In this layer, RMI uses the Proxy design pattern. In the Proxy pattern, an object in one context is represented by another (the proxy) in a separate context. The proxy knows how to forward method calls between the participating objects. The following class diagram illustrates the Proxy pattern.



In RMI's use of the Proxy pattern, the stub class plays the role of the proxy, and the remote service implementation class plays the role of the “Real Subject”.

A skeleton is a helper class that is generated for RMI to use. The skeleton understands how to communicate with the stub across the RMI link. The skeleton carries on a conversation with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub.

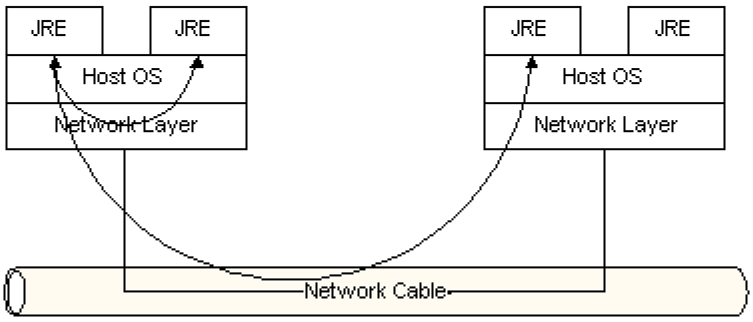
## 2. Remote Reference Layer

The remote reference layer defines and supports the invocation semantics of the RMI connection. This layer maintains the session during the method call. This layer provides a *RemoteRef* object that represents the link to the remote service implementation object. The stub objects use the *invoke()* method in *RemoteRef* to forward the method call. The *RemoteRef* object understands the invocation semantics for remote services.

### 3. Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.

Even if two JVMs are running on the same physical computer, they connect through their host computer's TCP/IP network protocol stack. (This is why you must have an operational TCP/IP configuration on your computer to run the Exercises in this course). The following diagram shows the unfettered use of TCP/IP connections between JVMs.



TCP/IP provides a persistent, stream-based connection between two machines based on an IP address and port number at each end. Usually a DNS name is used instead of an IP address; this means you could talk about a TCP/IP connection between *subdomain.domain.com:3080* and *subdomain.anydomain.com:4488*. In the current release of RMI, TCP/IP connections are used as the foundation for all machine-to-machine connections.

## Creating RMI Application:

There are six steps to writing the RMI applications.

a. Create the remote interface
b. Provide the implementation of the remote interface
c. Compile the implementation class and create the stub and skeleton
d. Start the registry service by rmiregistry tool
e. Create and start the remote application
f. Create and start the client application

The client application needs only two files, remote interface and client application. In the RMI application, both client and server interact with the remote interface. The client application invokes methods on the proxy object; RMI sends the request to the remote JVM. The return value is sent back to the proxy object and then to the client application.

## Parameters in RMI

RMI supports method calls to remote objects. When these calls involve passing parameters or accepting a return value, how does RMI transfer these between JVMs? What semantics are used? Does RMI support pass-by-value or pass-by-reference? The answer depends on whether the parameters are primitive data types, objects, or remote objects.

## 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 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) |

## 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. CORBA can be written in C, C++, COBAL and JAVA.

## 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 major components that make CORBA architecture include are:

a. *Interface Definition Language (IDL)*
b. *Object Request Broker (ORB)*
c. *The Portable Object Adaptor (POA)*
d. *Naming Service, and*
e. *Inter-ORB Protocol (IIOP).*

**Interface Definition Languages:** CORBA uses the concept of an interface Definition language to implement the services. IDI. is a neutral programming language method of defining how a service is implemented. The syntax of this language is independent of any programming language. IDI. is used to define an interface, which describes the methods that are available and the parameters that are required when a call is made for a remote object

**Object Request Broker (ORB):** The foundation of CORBA architecture is based on the Object Request Broker (ORB). The ORB is prime mechanism that acts as a middleware between the client and the server. It provides a heterogeneous environment in which a client can remotely communicate with the server irrespective of the hardware, the operating systems, and the software development language used. The ORB helps to establish connection with remote server. When a client wants to use the services of a server, it first needs reference to the object that is providing the service. Here, the ORB plays its role; the ORB resolves the client request for the object reference on behalf of the client and thereby enables the client and the server to establish the connectivity

**Portable Object Adaptor (POA):** The POA connects the server object implementation to the ORB. It extends the functionality of the ORB and some its services include activation and deactivation of the object implementations, generation and management of object references, mapping of object references to their implementations, and dispatching of client requests to server objects through a skeleton.

**Naming Service:** Defines how CORBA objects can be looked up by a name. It is a Common Object Service (COS) and allows an object to be published using a symbolic name and allows clients to obtain references to the object using a standard API The CORBA naming service provides a naming structure for remote objects

**Internet Inter-ORB Protocol (IOP):** The CORBA standard includes specifications for inter-ORB communication protocols that transmit object requests between various ORBS running on the network. The protocols are independent of the particular ORB implementations running at either end. An ORB implemented in Java can talk to an ORB implemented in C, as long as they are both compliant with the CORBA standard. The inter-ORB protocol delivers messages between two ORBS. These messages might be method requests, return values, error messages etc. The inter-ORB protocol (IIOP) also deals with differences between two ORB implementations. If you want two ORBS to talk, just make sure they both speak the same inter-ORB protocol (IOP). The inter-ORB Protocol (IOP) is based on TCP/IP and so is extensively used on the Internet.