Distributed Application

Introduction

- The term distributed applications, is used for applications that require two or more autonomous processes to cooperate in order to run them.
- A distributed application consists of one or more local or remote clients that communicate with one or more servers on several machines linked through a network.

Application architecture

- Single tier
  - A single-tier application is an application where the Presentation layer, Logical Layer and Data Layer lies in the same machine.
- 2 tier
  - A software architecture in which a presentation layer runs on a client and Business Layer and Data Layer runs on a server.
- 3 tier
  - Presentation Layer, Business Layer and Data Layer runs in different machines
- N tier
  - There can be multiple middle layer for multiple purpose

Client-Server

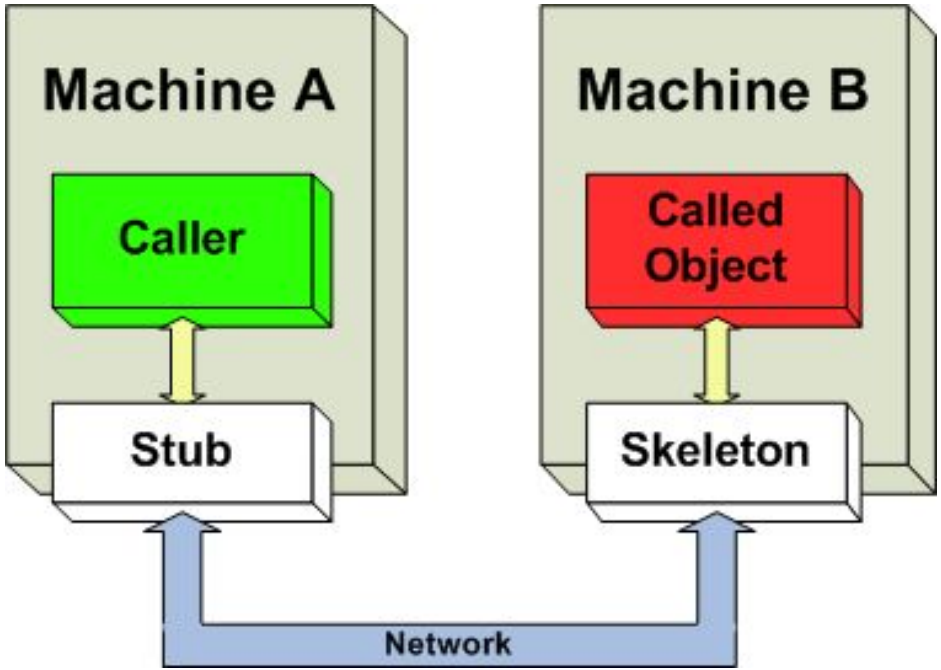
- 2 Tier application (can be 3 tier as well)
- Fat Client
  - Presentation Layer along with Application Logic resides on the client side
- Thin Client
  - Only presentation layer resides on the client side

Distributed Object

- Distributed objects are objects in a distributed computing environment that allow multiple systems or devices to interact with them as if they were local objects
- A distributed object behaves like a regular object, but the underlying infrastructure allows it to be used across different machines over a network.

How Distributed Objects Work:

- Client-Server Communication:
  - A client calls methods on an object as if it were local, but the object resides on a remote server.
  - The middleware handles the communication between the client and the server.
- Object References:
  - A distributed object reference allows a client to communicate with an object located elsewhere.
  - The reference is usually created via a registry or directory service.
- Stubs and Skeletons:
  - When a client calls a method on a remote object, a stub (proxy object) on the client side forwards the call to the skeleton (server-side object) on the remote machine.
  - The skeleton processes the call and sends the result back to the client through the stub.


Machine A
Caller
Stub
Machine B
Called Object
Skeleton
Network

- Stub
  - A **stub** is a client-side proxy object that represents the remote object. It acts as a local placeholder for the remote object and is responsible for
    - Forwarding method call
    - Marshaling
    - Sending Request
    - Receiving Response
- Skeleton
  - A **skeleton** is a server-side component that corresponds to the remote object. It is responsible for
    - Receiving Request
    - UnMarshaling
    - Invoking Method
    - Returning Result

Introduction of RMI

- 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*.
- Core feature of Java’s distributed computing model.

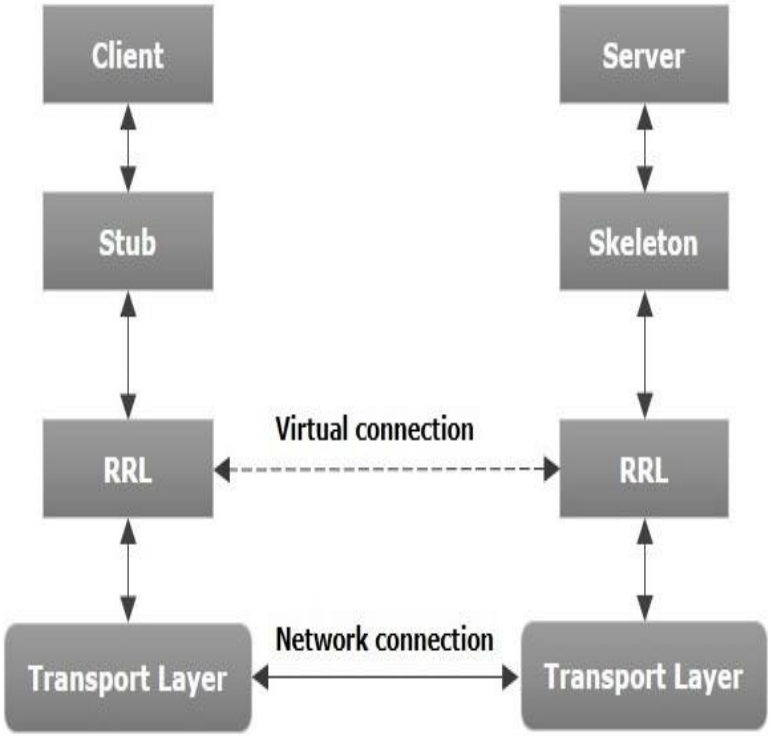
Benefits of RMI

- RMI simplifies the development of distributed applications by allowing objects to communicate across multiple JVMs.
- Leverages Java’s native object-oriented design, making it easier for developers familiar with Java to work with distributed objects.
- Abstracts the complexity of network communication, allowing developers to call methods on remote objects as if they were local objects.
- RMI supports automatic garbage collection of remote objects, managing memory efficiently across JVMs.

Limitation of RMI

- RMI is limited to Java-only environments.
- Serialization and deserialization of objects, combined with network transmission, introduce performance overhead.
- Latency from network communication can make remote method invocations slower than local calls.
- Java RMI is not ideal for large-scale systems.
- Only Serializable objects can be transferred between JVM.
- Large object serialization can also lead to performance bottlenecks.

RMI Architecture



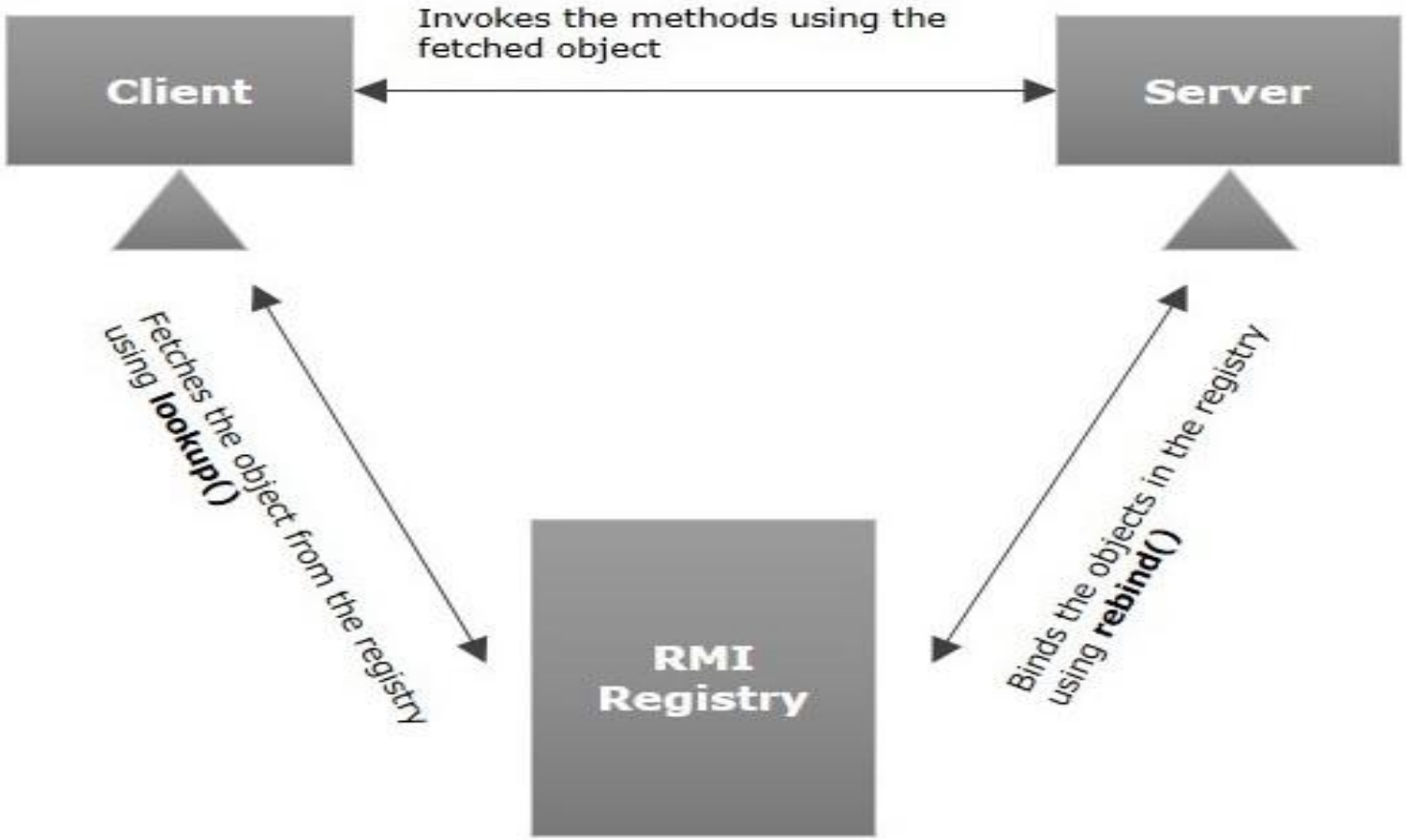
- In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).
- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.
- **RRL(Remote Reference Layer)** − 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.

Working of RMI

- 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 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).


Client
Invokes the methods using the fetched object
Server
Fetches the object from the registry using lookup()
RMI Registry
Binds the objects in the registry using rebind()

Steps to Write RMI program

- Create the remote interface
- Provide the implementation of the remote interface
- Compile the implementation class and create the stub and skeleton objects using the rmic tool
- Start the registry service by rmiregistry tool (You can start registry from code itself)
- Create and start the remote application
- Create and start the client application

- Creating Server
  - Step 1: Create interface that extends Remote
  - Step 2: Provide implementation that extends UniCastRemoteObject
  - Step 3: Create Server class with main method
  - Step 4 Create registry using: LocateRegistry.createRegistry(1099)
  - Step 5: Add the remote object to registry using Naming.rebind(“name”, object);
- Creating Client
  - Step 1: Create Client class with main method
  - Step 2: Create Registry object using LocateRegistry.getRegistry(host, port);
  - Step 3: Get object using registry.lookup(“object name”);
  - Step 4: Typecast the object to Interface and invoke the method required.