Unit 8- RMI and CORBA

Introduction

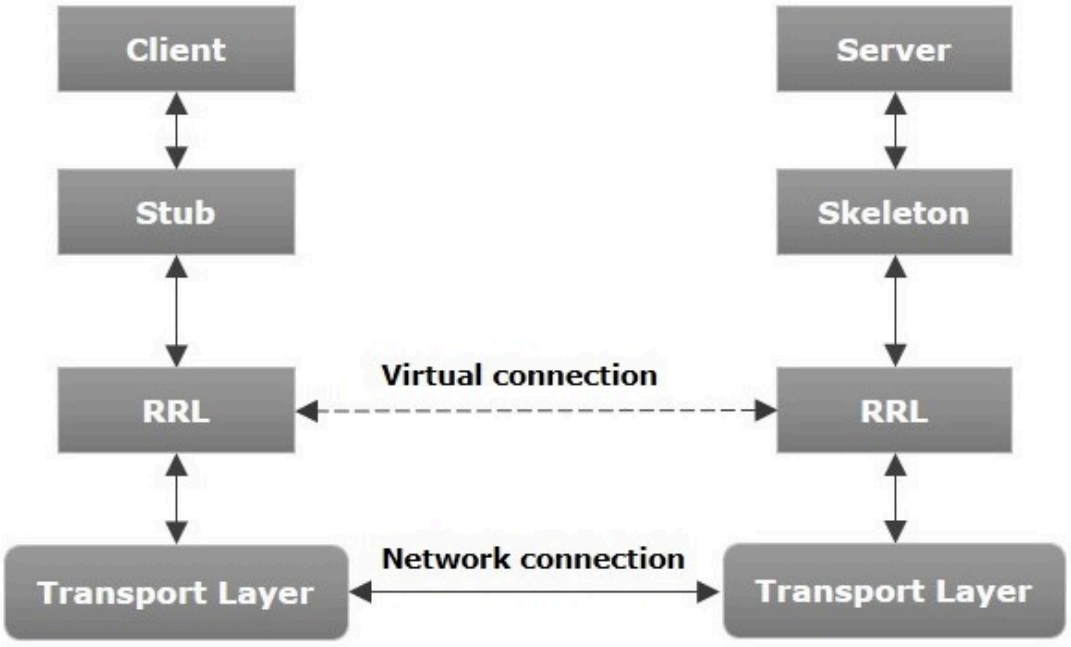
- RMI and CORBA are both part of distributed java programming.
- Distributed java programming are of two types: Client/Server model and Distributed Object model
- RMI and CORBA are the part of Distributed Object model.
- Distributed Object model system is a collection of objects that isolates the requesters of services (clients) from the providers of services (servers) by a well-defined encapsulating interface.
- In this model an object is defined in a remote JVM and this can be invoked by another JVM.

RMI(Remote Method Invocation)

- In simple terms Java RMI is an api that allows an object running on one JVM to access/invoke methods on an object running in another JVM.

Architecture

The RMI system is structured into three primary layers:



Stub/Skeleton Layer:

- Stub (Client-Side Proxy): Acts as a gateway for the client, representing the remote object locally. It forwards method calls from the client to the remote object.
    - Initiate connection with virtual JVM
    - Write and transmit parameters to virtual JVM
    - Waits for the result, reads the return value and return the value to the caller
- Skeleton (Server-Side Dispatcher): Receives incoming requests from the stub and dispatches them to the appropriate method implementations on the server. Note: In Java 2 SDK, v1.2 and later, skeletons are no longer required; the functionality is handled dynamically.
    - Read params for remote method
    - Invokes methods on the actual remote object
    - Write and transmit the result to the caller

Remote Reference Layer:

- Manages the references to remote objects, handling the semantics of remote method invocations, such as invocation to a single object or to a replicated object.

Transport Layer:

- Responsible for setting up connections, managing them, and tracking remote objects. It handles the low-level communication between JVMs, typically over TCP/IP.

Process:

1. **Defining the Remote Interface:**
    a. Create an interface that extends java.rmi.Remote.
    b. Each method in this interface must declare java.rmi.RemoteException in its throws clause.
2. **Implementing the Remote Object:**
    a. Develop a class that implements the remote interface.
    b. This class should extend java.rmi.server.UnicastRemoteObject to allow the creation of a remote object.
3. **Generating Stubs (and Skeletons in earlier versions):**
    a. Use the rmic compiler to generate the stub classes required for remote communication.
    b. In versions prior to Java 2 SDK, v1.2, skeletons were also generated.
4. **Setting Up the RMI Registry:**
    a. Start the RMI registry using the rmiregistry command. This registry allows clients to look up remote objects by name.

5. **Binding the Remote Object:**
    a. In the server application, bind the remote object to the RMI registry with a unique name using the Naming.rebind() method.
6. **Client Lookup and Invocation:**
    a. The client looks up the remote object in the RMI registry using Naming.lookup() and invokes methods as if it were a local object. The stub handles the communication details.

Implementation

We will need to create four files:

- One to create the remote interface
- One to implement the remote object
- One for the server to run
- One for the client

FactorialService.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;
//we always need to extends to remote while creating an remote interface
//also it may throw RemoteException so we will need write that as well
public interface FactorialService extends Remote {
  long factorial(int n) throws RemoteException;
}
```

FactImp.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
//must extend UnicastRemoteObject and implement FactorialService(this is the interface defined before)
public class FactImp extends UnicastRemoteObject implements FactorialService {
// we need to define this constructor because it extends teh UnicastRemoteObject
  protected FactImp() throws RemoteException {
//  we call the super method to properly initialize the UnicastRemoteObject
//  so that it can set up the necessary infrastructure for the remote object,
//  such as exporting it to the RMI runtime.
      super();
  }
//    this func name must be the same as defined in interface
  public long factorial(int n) throws RemoteException {
    if (n <= 0) {
       throw new IllegalArgumentException("Number must be non-negative.");
    }
    return fact(n);
  }
  private long fact(int n) {
     long fact = 1;
     for (int i = 1; i <= n; i++) {
        fact *= i;
     }
     return fact;
  }
}
```

FactorialServer.java

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

public class FactorialServer {
//    we define the main func here
  public static void main(String[] args) {
     try {
        // Create and export the remote object
        FactorialService service = new FactImp();
```

```
        // Start the RMI registry on port 1099
        LocateRegistry.createRegistry(1099);
        // Bind the remote object's stub in the registry
        Naming.rebind("FactorialService", service);
        System.out.println("Factorial server ready.");
    } catch (Exception e) {
        System.err.println("Server exception: " + e.toString());
        e.printStackTrace();
    }
  }
}
```

FactorialClient.java

```
import java.rmi.Naming;

public class FactorialClient {
  public static void main(String[] args) {
    try {
        // calling the remote object on the server
        FactorialService service = (FactorialService) Naming.lookup("FactorialService");
        int num = 10;
        long result = service.factorial(num);
        System.out.println(result);
    } catch (Exception e) {
        System.err.println("Client error" + e.toString());
        e.printStackTrace();
    }
  }
}
```

Now compile these four files and run the server and client. In the older version we need to start the registry but not on the newer version.

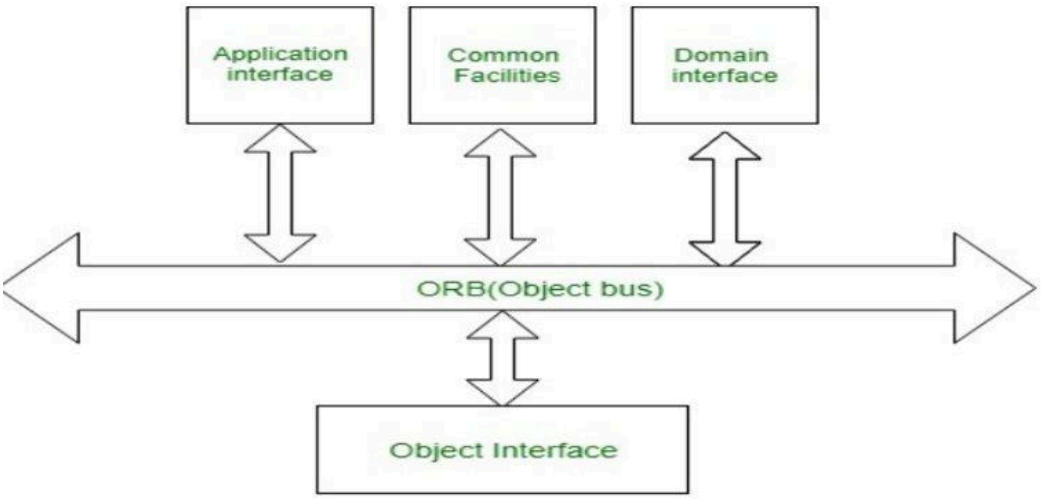
Common Object Request Broker Architecture (CORBA):

- The Common Object Request Broker Architecture (or CORBA) is an industry standard developed by the Object Management Group (OMG) to aid in distributed objects programming.
- It is important to note that CORBA is simply a specification for the design and implementation of distributed object-oriented computing systems.
- Similar to RMI it was developed for building distributed computing systems but the main difference is that RMI was created for java applications whereas CORBA is language and platform independent.

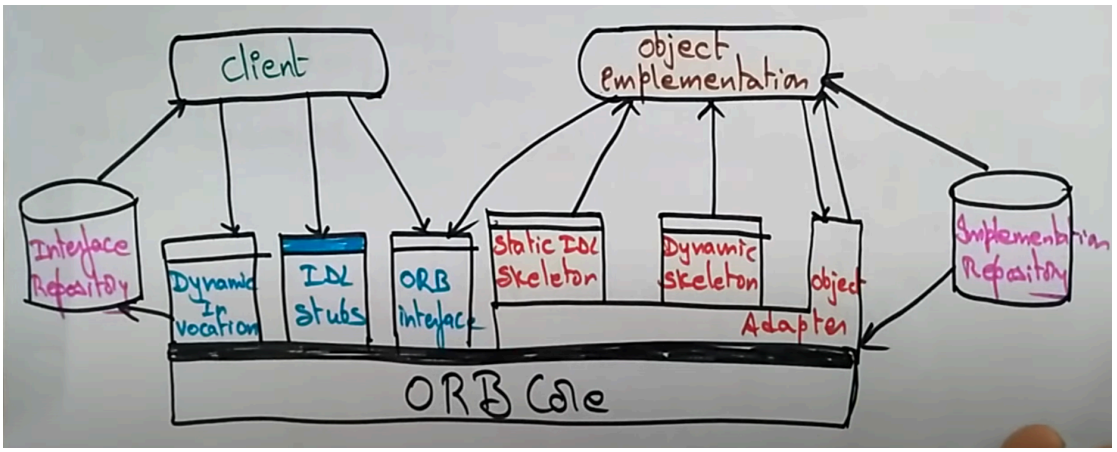
Differences Between RMI and CORBA

RMI	CORBA
RMI is a Java-specific technology.	CORBA has implementation for many languages.
It uses Java interface for implementation.	It uses Interface Definition Language (IDL) to separate interface from implementation.
RMI objects are garbage collected automatically.	CORBA objects are not garbage collected because it is language independent and some languages like C++ does not support garbage collection.
RMI programs can download new classes from remote JVM's.	CORBA does not support this code sharing mechanism.
RMI passes objects by remote reference or by value.	CORBA passes objects by reference.
Java RMI is a server-centric model.	CORBA is a peer-to-peer system.
RMI uses the Java Remote Method Protocol as its underlying remoting protocol.	CORBA use Internet Inter- ORB Protocol as its underlying remoting protocol.
The responsibility of locating an object implementation falls on JVM.	The responsibility of locating an object implementation falls on Object Adapter either Basic Object Adapter or Portable Object Adapter.

Architecture:



CORBA's architecture is centered around the Object Request Broker (ORB), which facilitates communication between clients and servers in a distributed environment.



---

1. ORB Core

- **Definition:** ORB handles the communication, marshaling, and unmarshaling of parameters so that the parameter handling is transparent for a CORBA server and client applications.
- **Role:** It acts as middleware that allows clients to invoke methods on remote objects without needing to know their physical location or implementation details.

2. ORB Interface

- **Role:** Provides an API for both clients and servers to interact with the ORB.The client initiates the request to access services provided by the remote object. It interacts with the ORB core either through generated stubs (IDL stubs) or directly using dynamic invocation interfaces.
    - IDL Stubs**:** These are generated from the Interface Definition Language (IDL) and act as a local representation of the remote object for the client.
    - Dynamic Invocation**:** This provides an alternative way for the client to invoke methods on a remote object without relying on precompiled stubs.
- **Functionality:** It offers methods for locating objects, invoking methods, and handling object references

3. Object Implementation(server)

- **Role:** This is the actual implementation of the remote object, containing the business logic.
- **Interaction:** The object implementation processes the client's request and sends the response back.

4. Object Adapter

- **Definition:** A component that connects the ORB with the actual object implementation.
- **Role:** It is responsible for:
    - Activating and deactivating objects.
    - Managing object references.
    - Forwarding requests to the appropriate object implementation.

5. CORBA IDL stubs and skeletons

- CORBA IDL stubs and skeletons serve as the ``glue'' between the client and server applications, respectively, and the ORB.
- The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler.
- The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

6. Dynamic Invocation Interface (DII)

- This interface allows a client to directly access the underlying request mechanisms provided by an ORB.
- DII uses interface repository at run time to discover interfaces
- No need for pre-compiled stubs.

7. Dynamic Skeleton Interface (DSI)

- This is the server side's analogue to the client side's DII.
- The DSI allows an ORB to deliver requests to an object implementation without the need of precompiled skeletons.
- Implemented via a DIR(Dynamic Invocation Routine)
- ORB invokes DIR for every DSI request it makes.

8. Interface Repository

- **Definition:** A database that stores metadata about interfaces defined in IDL.
- **Role:** Provides information about the methods and parameters of objects at runtime.
- **Usage:** Used primarily by the Dynamic Invocation Interface to make method calls without precompiled stubs.

9. Implementation Repository

- **Definition:** A database that stores information about the location and lifecycle of object implementations.
- **Role:** Helps the ORB find and activate the correct object implementation at runtime.

Summary

- The **client** uses either **IDL stubs** or **dynamic invocation** to send a request to the ORB.
- The ORB processes the request and forwards it to the server-side **skeleton** or **dynamic skeleton**, which bridges the request to the **object implementation**.
- The **object adapter** assists in managing object references and activation, while the **Interface Repository** and **Implementation Repository** provide metadata and object lifecycle information.
- The ORB ensures seamless communication, making distributed object interactions appear local to both clients and servers.

Example:

```
module HelloApp {
   // Define the interface
   interface Hello {
      // Declare a method
      string sayHello(“hello”);
   };
};
```

IDL (Interface definition language):

- IDL is a generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language. It can be written in C C++, Java,COBOL etc
- Why?
    - IDL is used to create target language stubs and skeletons for building CORBA clients and servers.
    - In java we cannot separate a class definition from its implementations as we can in C++.

    - Therefore CORBA uses IDL for defining interfaces between clients and servers.
- Basic Syntax of IDL
    - **Modules**: For grouping related interfaces.
    - **Interfaces**: Define the operations (methods) and attributes of an object.
    - **Data Types**: Define input, output, and return types of methods.
- How IDL Works in CORBA:
    - **Write the IDL File**: The developer defines the object interface in IDL.
    - **Compile the IDL File**: An IDL compiler generates:
        - **Client Stubs**: Code for the client to invoke remote methods.
        - **Server Skeletons**: Code for the server to handle client requests.
    - **Implement the Application**: The client and server use the generated code for communication.
- Example:

```
module HelloApp {
   // Define the interface
   interface Hello {
      // Declare a method
      string sayHello(“hello”);
   };
};
```