RMI and CORBA

Introduction RMI

- RMI (Remote Method Invocation) applications offers two separate programs, a server and a **client**.
- It is a way that a programmer makes use of Java programming language and its development environment remotely.
- It is about how the objects on different computers interact in a distributed network.
- Basically, RMI provides the mechanism through which server and client can communicate and pass information back and forth. The RMI consists of **three layers** :
  - Stub/Skeleton layer
  - Remote Reference layer
  - Transport layer

Introduction RMI - [1]

- It is an API that provides a mechanism to create a distributed application in Java.
- It enables to access methods on an object running in another JVM.
- The connection between client and server applications is established using two objects called stub and skeleton.

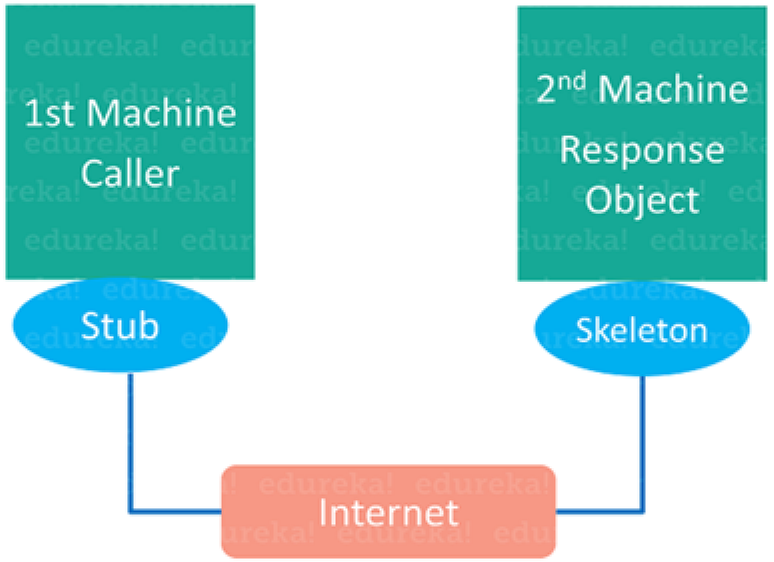
Introduction RMI - [2]

Stub

- It builds an information block and sends to server.

It includes

- An identifier of the remote object to be used
- Method name which is to be invoked.
- Parameters to the remote JVM



Introduction RMI - [3]

Skeleton Object:

- It passes the request from the stub object to the remote object.
- It calls the desired method on the real object present on the server.
- It forwards the parameters received from the stub object to the method.

Architecture of RMI - [1]

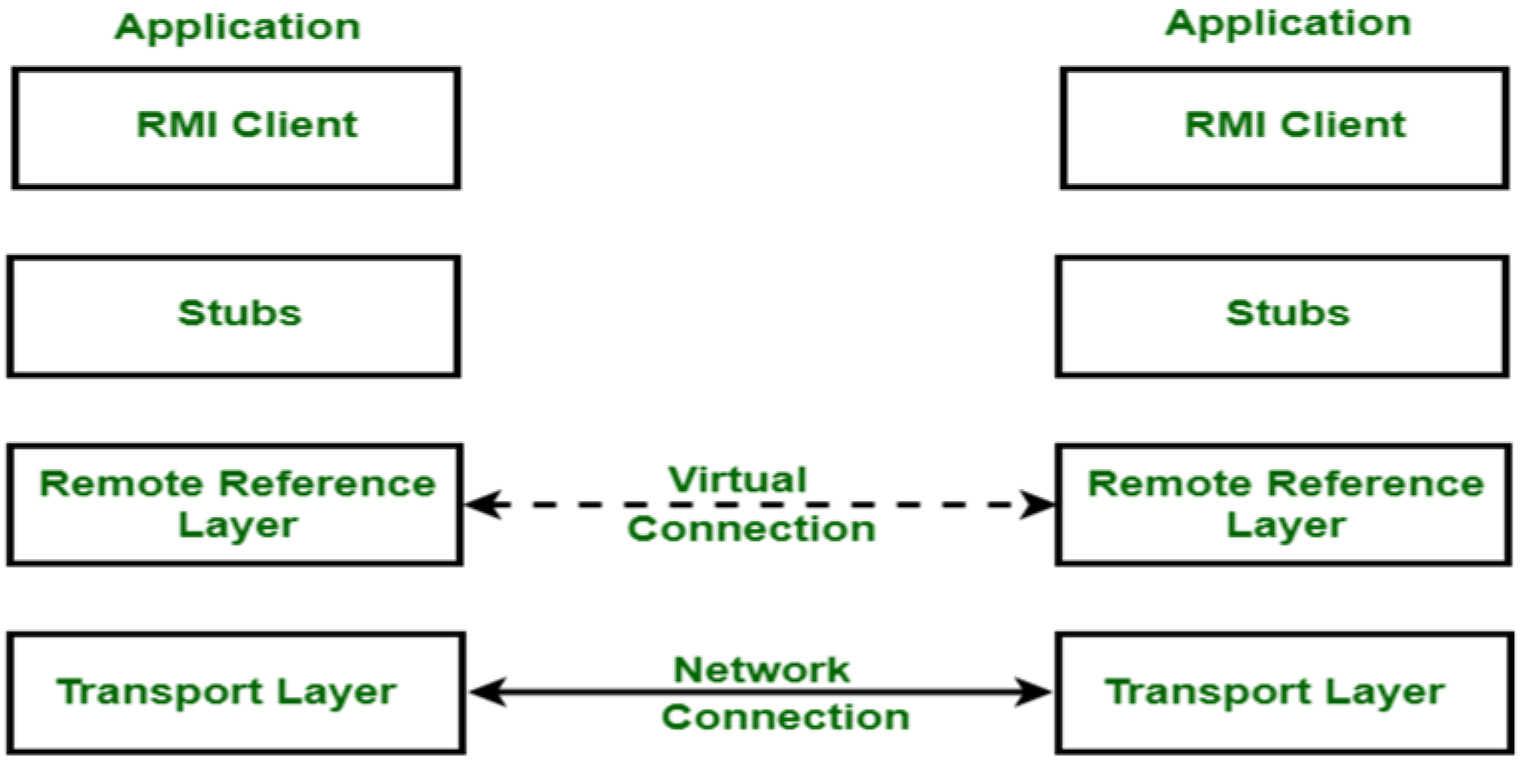


Figure – RMI Architecture

RMI Application Creation

- Defining a remote interface.
- Implementing the remote interface.
- Creating Stub and Skeleton Objects from the implementation class using RMIC (RMI Compiler)
- Start the RMI registry
- Create and execute the server application.
- Create and execute the client application.

Ref:: https://www.edureka.co/blog/remote-method-invocation-in-java/

RMI Application

https://www.tutorialspoint.com/java_rmi/java_rmi_application.htm

CORBA - Introduction

- Common Object Request Broker Architecture (CORBA)
- It is an open specification for the design and implementation of distributed object-oriented computing systems.
- It is a product of the Object Management Group.
- It was developed with the goal of enabling the development of extensible, reusable, and less costly computing systems through open standards.

CORBA - Introduction - [1]

- CORBA is similar to Java Remote Method Invocation (RMI)
- Both technologies were developed for building distributed computing systems.
- RMI is designed for distributed applications, which are always based on pure Java, whereas CORBA works with applications written in various programming languages.

CORBA Components

Object Request Broker (ORB)

- ORB handles the communication, marshaling, and unmarshaling of parameters so that the parameter handling is transparent for a CORBA server and client applications.

CORBA Server

- The CORBA server creates CORBA objects and initializes them with an ORB.
- The server places references to the CORBA objects inside a naming service so that clients can access them.

CORBA Components - [1]

Naming Service

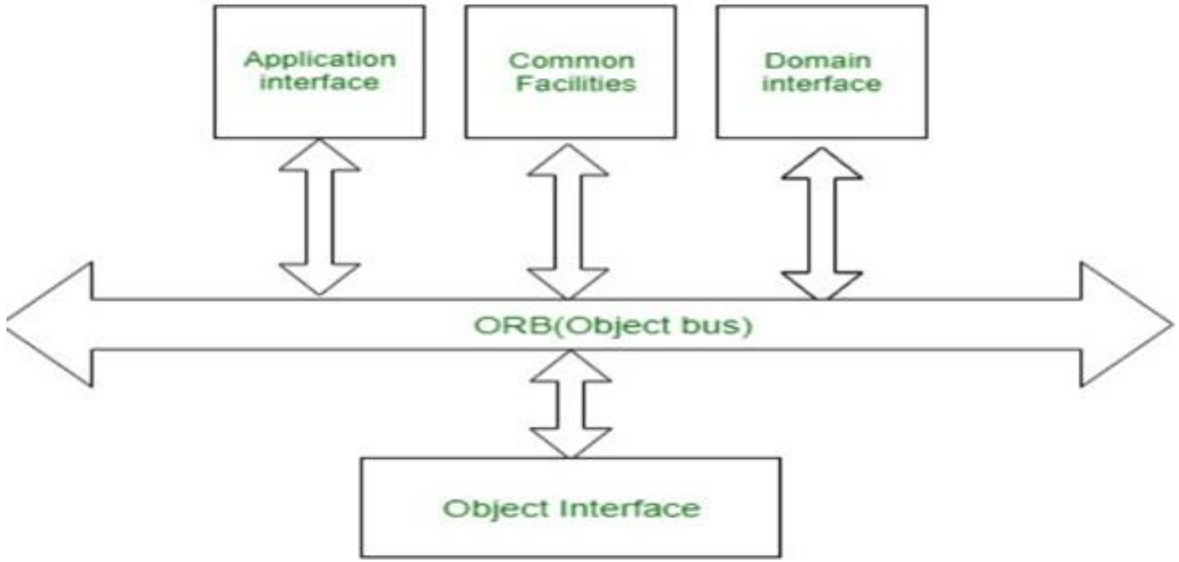
- The naming service holds references to CORBA objects.

CORBARequest node

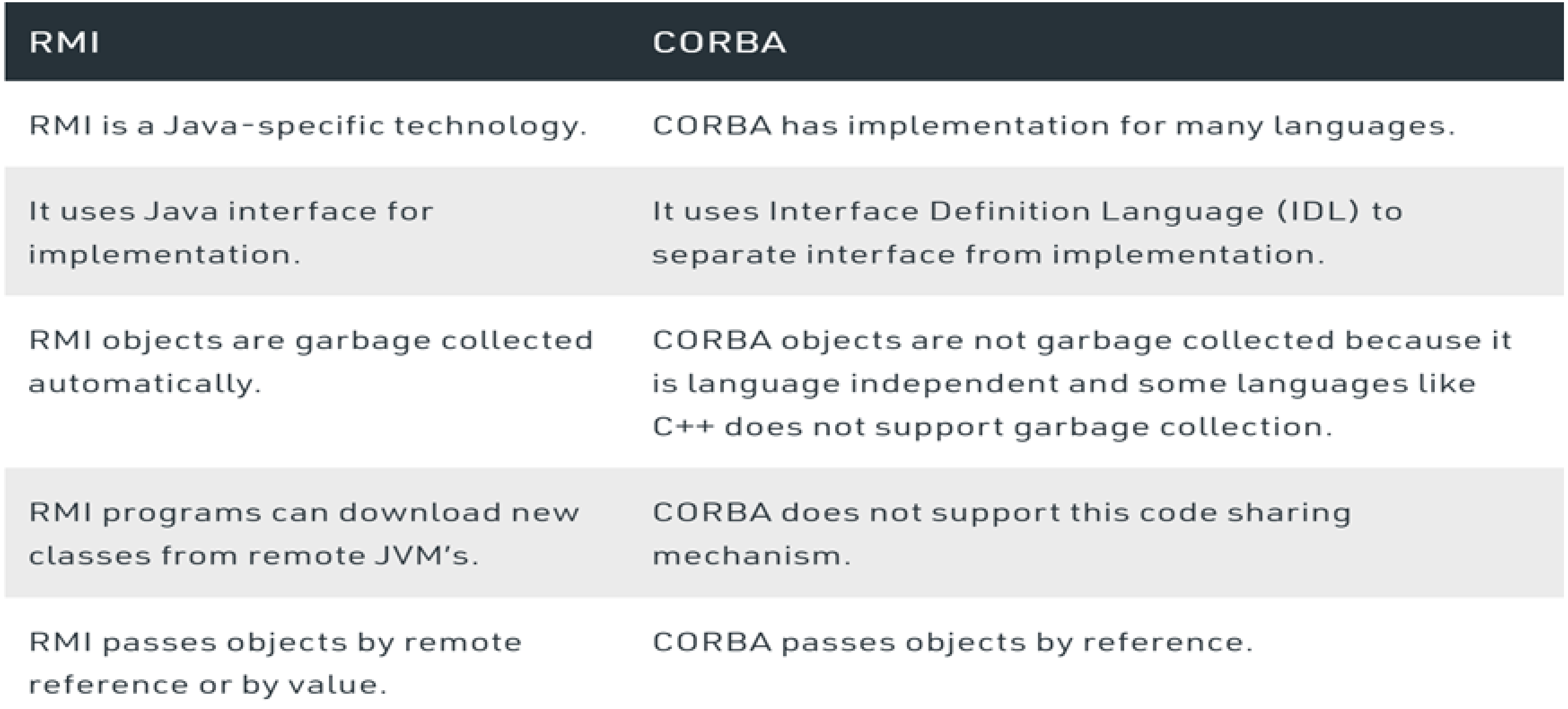
- CORBARequest node acts as a CORBA client

CORBA Reference Model

- It is a specification, which defines a broad vary of services for building distributed client-server applications.

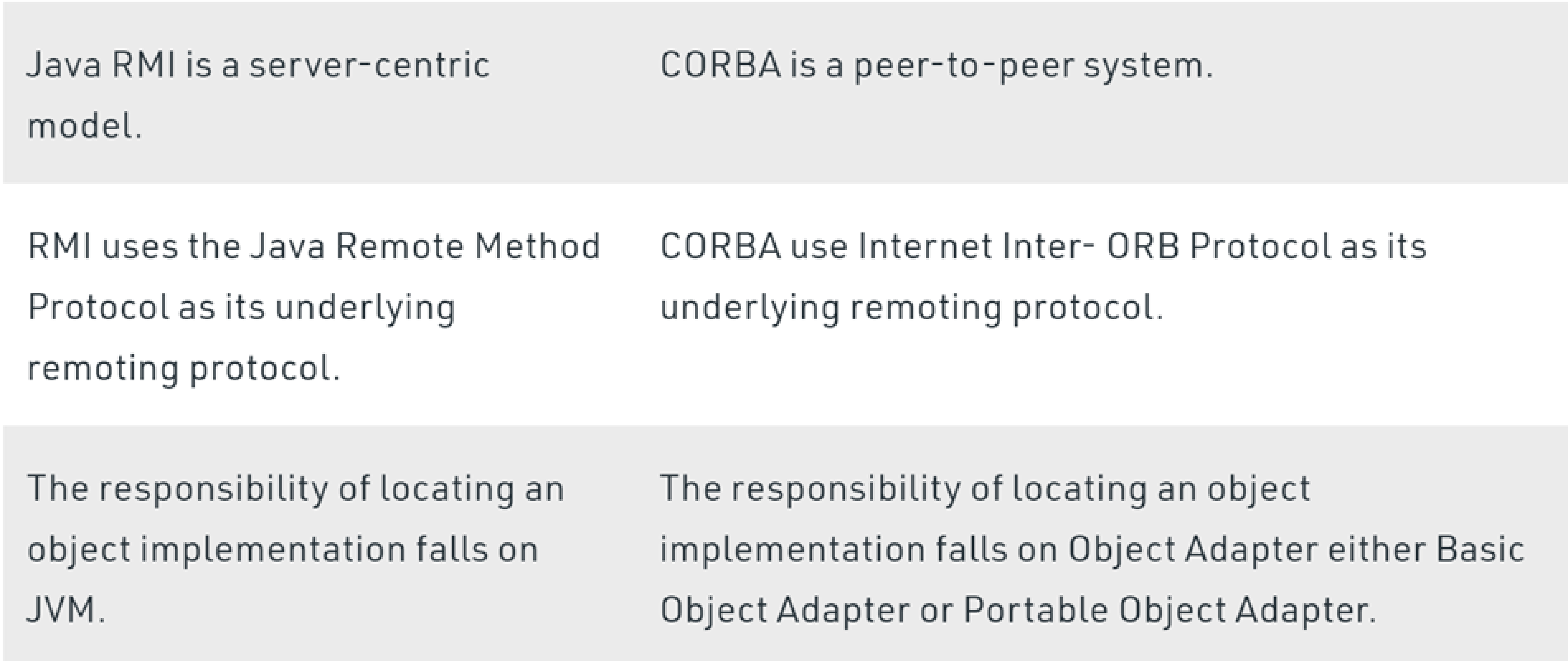


RMI vs CORBA



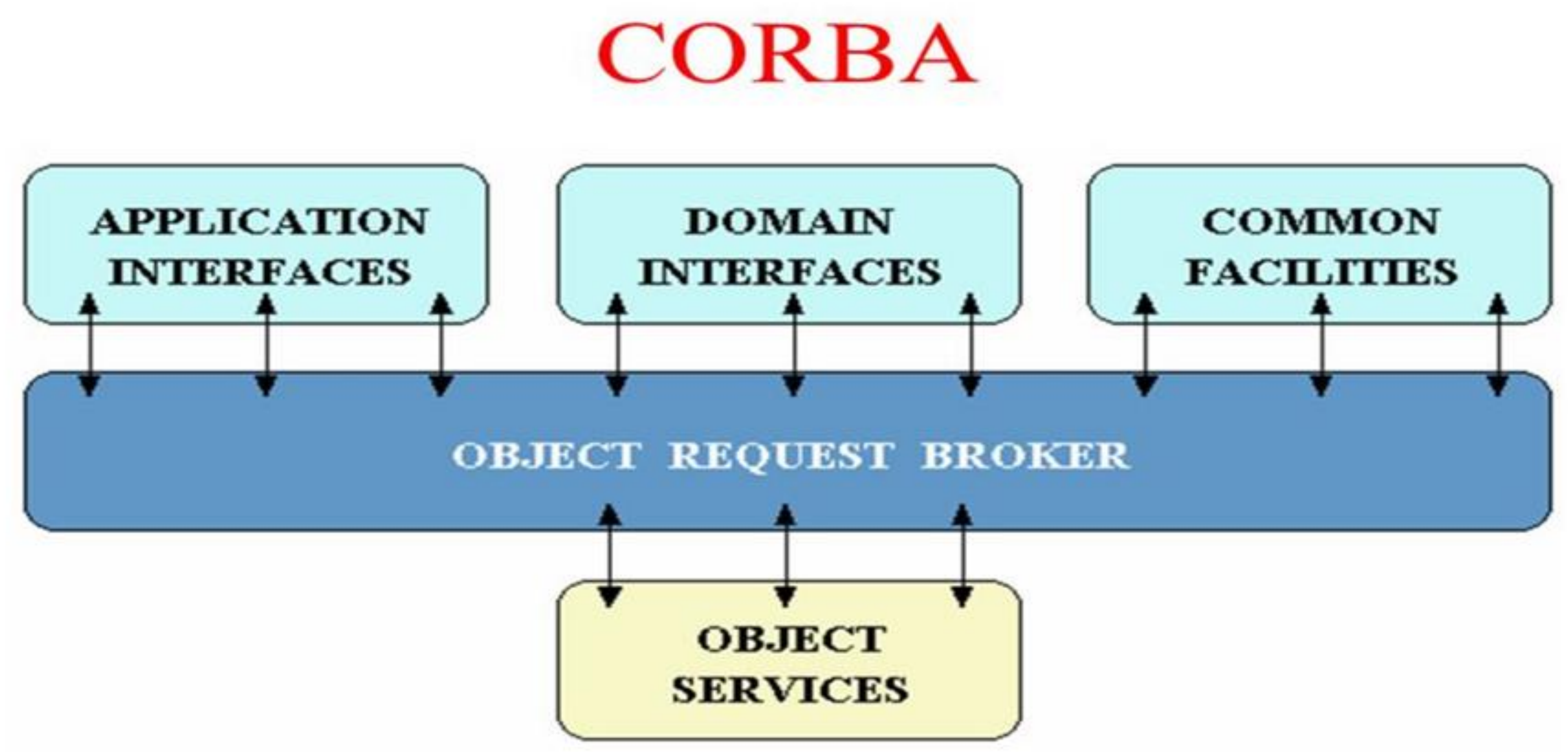
RMI	CORBA
RMI is a Java-specific technology.	CORBA has implementation for many languages.
It uses Java interface for implementation.	It uses Interface Definition Language (IDL) to separate interface from implementation.
RMI objects are garbage collected automatically.	CORBA objects are not garbage collected 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.

RMI vs CORBA

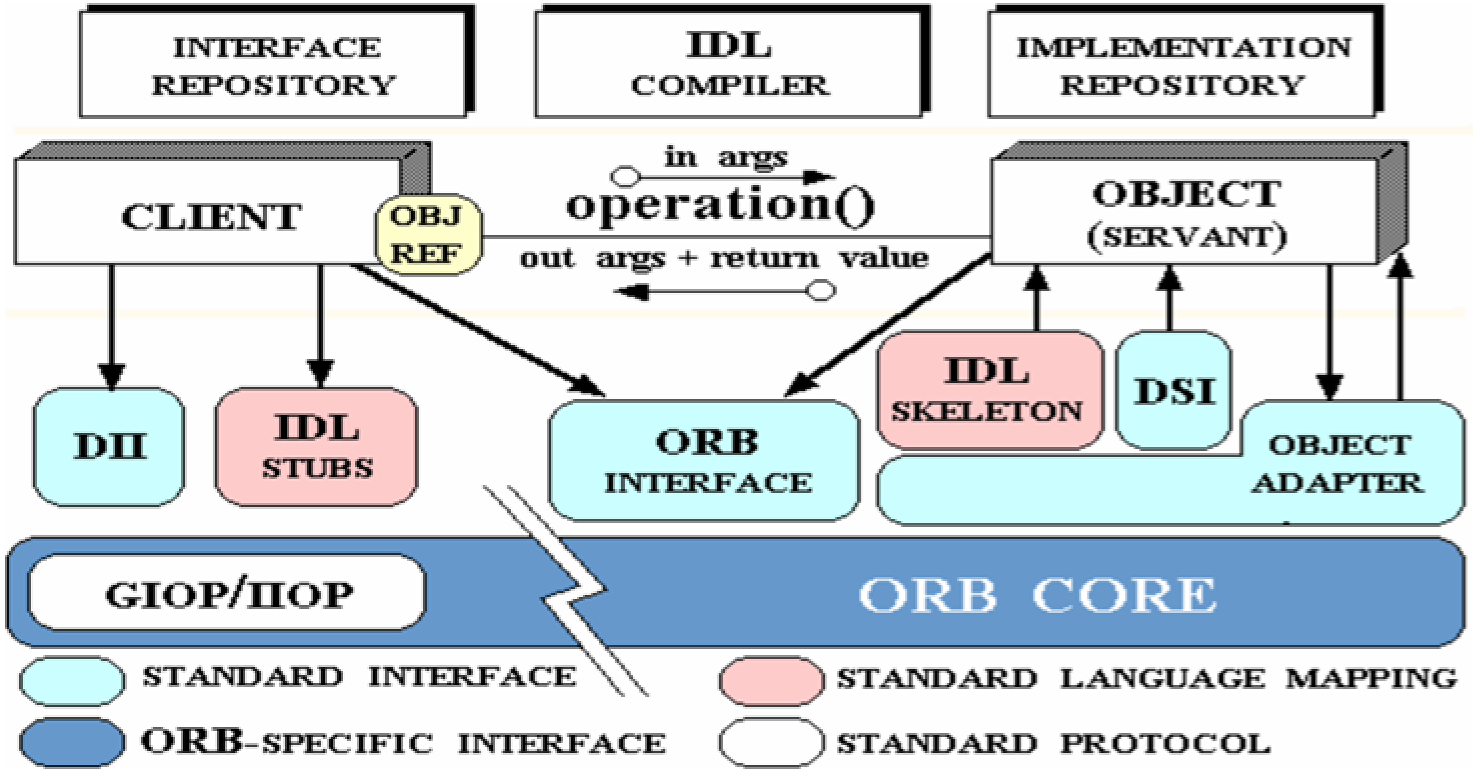


Java RMI is a server-centric model.	CORBA is a peer-to-peer system.
RMI uses the Java Remote Method Protocol as its underlying remoting protocol.	CORBA use Internet Inter- ORB Protocol as its underlying remoting protocol.
The responsibility of locating an object implementation falls on JVM.	The responsibility of locating an object implementation falls on Object Adapter either Basic Object Adapter or Portable Object Adapter.

CORBA Reference Model



CORBA ORB Architecture



CORBA ORB Architecture - [1]

- **Object --** This is a CORBA programming entity that consists of an *identity*, an *interface*, and an *implementation*, which is known as a *Servant*.
- **Servant --** This is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.
- **Client --** This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, i.e., obj->op(args).

CORBA ORB Architecture - [2]

Object Request Broker (ORB)

- The ORB provides a mechanism for transparently communicating client requests to target object implementations.
- The ORB simplifies distributed programming by decoupling the client from the details of the method invocations.

CORBA ORB Architecture - [3]

ORB Interface

- An ORB is a logical entity that may be implemented in various ways.
- To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB.
- This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

CORBA ORB Architecture - [4]

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.

CORBA ORB Architecture - [5]

Dynamic Invocation Interface (DII)

- This interface allows a client to directly access the underlying request mechanisms provided by an ORB.
- Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in.
- Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking *deferred synchronous* (separate send and receive operations) and *one way* (send-only) calls.

CORBA ORB Architecture - [6]

Dynamic Skeleton Interface (DSI)

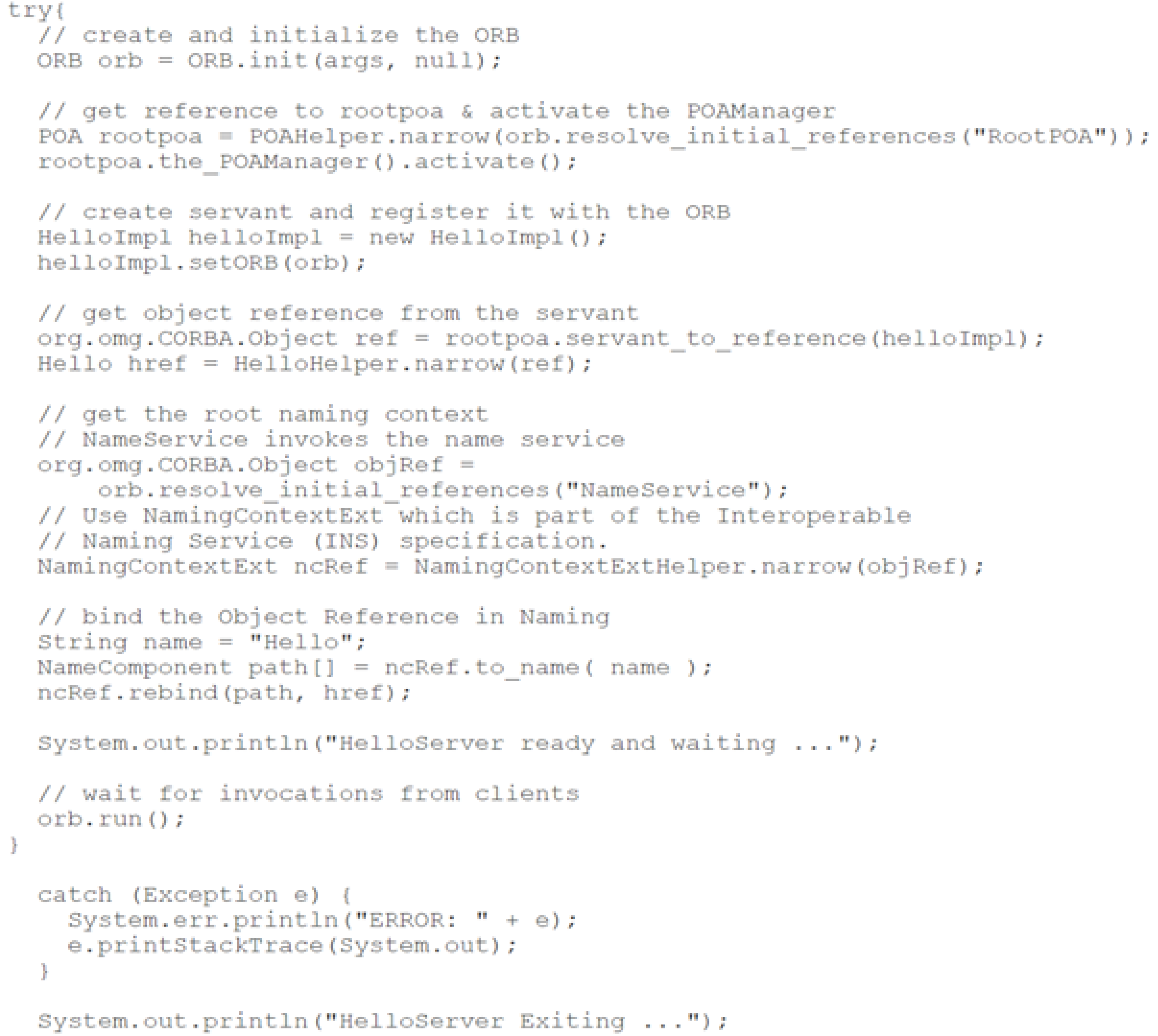
- This is the server side's analogue to the client side's DII.
- The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing.
- The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

CORBA ORB Architecture - [7]

Object Adapter

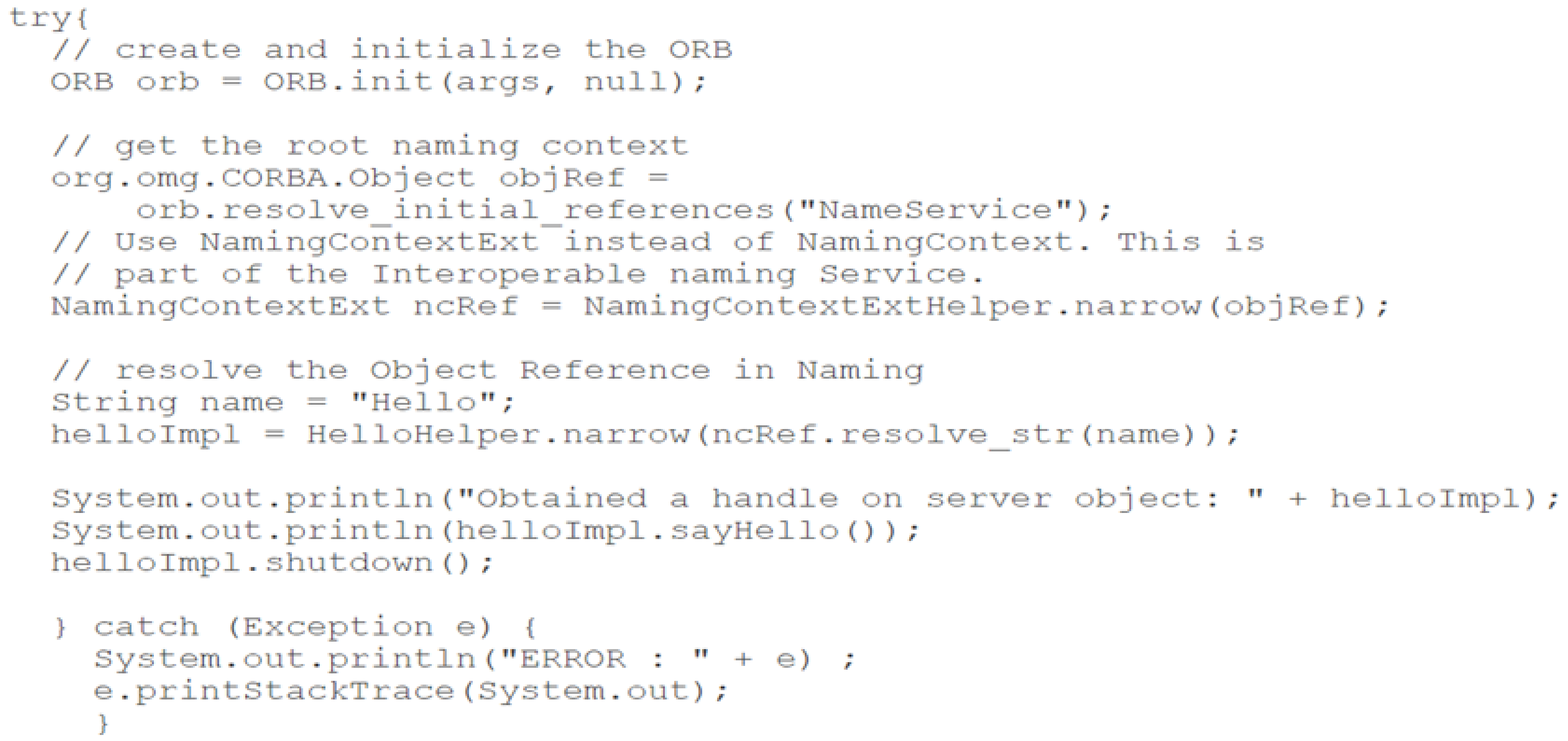
- This assists the ORB with delivering requests to the object and with activating the object.
- More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).

Server Program



```
try{
  // create and initialize the ORB
  ORB orb = ORB.init(args, null);

  // get reference to rootpoa & activate the POAManager
  POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
  rootpoa.the_POAManager().activate();

  // create servant and register it with the ORB
  HelloImpl helloImpl = new HelloImpl();
  helloImpl.setORB(orb);

  // get object reference from the servant
  org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
  Hello href = HelloHelper.narrow(ref);

  // get the root naming context
  // NameService invokes the name service
  org.omg.CORBA.Object objRef =
      orb.resolve_initial_references("NameService");
  // Use NamingContextExt which is part of the Interoperable
  // Naming Service (INS) specification.
  NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

  // bind the Object Reference in Naming
  String name = "Hello";
  NameComponent path[] = ncRef.to_name( name );
  ncRef.rebind(path, href);

  System.out.println("HelloServer ready and waiting ...");

  // wait for invocations from clients
  orb.run();
}

  catch (Exception e) {
    System.err.println("ERROR: " + e);
    e.printStackTrace(System.out);
  }

  System.out.println("HelloServer Exiting ...");
```

Client Program Ref: https://docs.oracle.com/javase/7/docs/technotes/guides/idl/jidlExample.html



```
try{
  // create and initialize the ORB
  ORB orb = ORB.init(args, null);

  // get the root naming context
  org.omg.CORBA.Object objRef =
      orb.resolve_initial_references("NameService");
  // Use NamingContextExt instead of NamingContext. This is
  // part of the Interoperable naming Service.
  NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

  // resolve the Object Reference in Naming
  String name = "Hello";
  helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));

  System.out.println("Obtained a handle on server object: " + helloImpl);
  System.out.println(helloImpl.sayHello());
  helloImpl.shutdown();

  } catch (Exception e) {
    System.out.println("ERROR : " + e) ;
    e.printStackTrace(System.out);
    }
```