**Experiment : 2**

**Aim:**  Develop any distributed application using CORBA to demonstrate object brokering.

(Calculator or String operations).

**CORBA**

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together.

CORBA is a standard for distributing objects across networks so that operations on those objects can be called remotely. CORBA is not associated with a particular programming language, and any language with a CORBA binding can be used to call and implement CORBA objects. Objects are described in a syntax called Interface Definition Language (IDL).

CORBA includes four components:

Object Request Broker (ORB)

The Object Request Broker (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.

Naming service

The naming service holds references to CORBA objects.

CORBARequest node

The CORBARequest node acts as a CORBA client.

CORBA (Common Object Request Broker Architecture) provides a mechanism for handling data marshaling and demarshaling between different programming languages through the use of the Interface Definition Language (IDL) and the CORBA Object Request Broker (ORB).

Here's how CORBA handles data marshaling and demarshaling:

Interface Definition Language (IDL):

IDL is a language-independent specification used to define the interfaces of CORBA objects.

IDL provides a standardized way to describe the structure and types of data that will be exchanged between CORBA objects.

IDL defines a set of primitive types (e.g., integers, strings, booleans) and composite types (e.g., structs, sequences) that can be used in interface definitions.

IDL Compiler:

The IDL compiler takes the IDL interface definitions as input and generates language-specific stubs and skeletons.

The stubs and skeletons act as proxies between the client and server, handling the marshaling and demarshaling of data.

Marshaling:

Marshaling is the process of converting data from its native representation in a programming language to a standardized format that can be transmitted over the network.

When a client invokes a remote method on a CORBA object, the arguments passed to the method are marshaled into a platform-independent format.

The marshaling process involves converting data types, encoding the data, and packaging it into a message.

Demarshaling:

Demarshaling is the reverse process of marshaling, where the received data is converted back from the standardized format to the native representation of the programming language.

When the request message reaches the server, the ORB demarshals the data, extracting the arguments and converting them to the appropriate data types expected by the server object.

The demarshaled data is then passed to the server object, which processes the request and generates a response.

Stubs and skeletons are components generated by the CORBA (Common Object Request Broker Architecture) infrastructure to facilitate communication between clients and servers in a distributed system. They handle the marshaling and demarshaling of data, as well as the method invocations and communication protocols. Here's an explanation of stubs and skeletons:

Stubs:

Stubs are client-side components generated from the IDL (Interface Definition Language) interface definitions.

Stubs act as proxies for the remote CORBA objects that the client wants to access.

The stubs provide a local representation of the remote object's interface, allowing the client to invoke methods on the remote object as if it were a local object.

Stubs handle the marshaling of method arguments into a format suitable for transmission over the network.

When the client invokes a method on the stub, it marshals the arguments, sends the request to the server, and waits for the response.

Upon receiving the response, the stub demarshals the return value and any out/inout parameters, allowing the client to access the results.

Skeletons:

Skeletons are server-side components generated from the IDL interface definitions.

Skeletons implement the actual behavior of the remote CORBA objects.

When the server receives a request from a client, the ORB (Object Request Broker) routes the request to the appropriate skeleton based on the object's reference.

Skeletons handle the demarshaling of the received request, extracting the method arguments.

The skeleton then invokes the actual implementation of the corresponding method in the server code.

After processing the method, the skeleton marshals the return value and any out/inout parameters into a response message and sends it back to the client via the ORB.

1. Create a new folder C:\Calc in C:\ directory.

2. Create Calc.idl file using Notepad. Save it with in C:\Calc folder with idl as extension. Paste the following into the idl file:

module CalcApp

{

interface Calc

{

exception DivisionByZero {};

float sum(in float a, in float b);

float div(in float a, in float b) raises (DivisionByZero);

float mul(in float a, in float b);

float sub(in float a, in float b);

};

};

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

5. Now a java file named CalcServer is created in C:\Calc with the following code:

import CalcApp.\*;

import CalcApp.CalcPackage.DivisionByZero;

import org.omg.CosNaming.\*;

import org.omg.CosNaming.NamingContextPackage.\*;

import org.omg.CORBA.\*;

import org.omg.PortableServer.\*;

import java.util.Properties;

class CalcImpl extends CalcPOA

{

@Override

public float sum(float a, float b)

{

return a + b;

}

@Override

public float div(float a, float b) throws DivisionByZero

{

if (b == 0)

{

throw new CalcApp.CalcPackage.DivisionByZero();

}

else

{

return a / b;

}

}

@Override

public float mul(float a, float b)

{

return a \* b;

}

@Override

public float sub(float a, float b)

{

return a - b;

}

private ORB orb;

public void setORB(ORB orb\_val)

{

orb = orb\_val;

}

}

public class CalcServer

{

public static void main(String args[])

{

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

CalcImpl helloImpl = new CalcImpl();

helloImpl.setORB(orb);

// get object reference from the servant

org.omg.CORBA.Object ref = rootpoa.servant\_to\_reference(helloImpl);

Calc href = CalcHelper.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 = "Calc";

NameComponent path[] = ncRef.to\_name(name);

ncRef.rebind(path, href);

System.out.println("Ready..");

// wait for invocations from clients

orb.run();

} catch (Exception e)

{

System.err.println("ERROR: " + e);

e.printStackTrace(System.out);

}

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

}

}6. Create another file C:\Calc\CalcClient.java with the following code in it:LIENT requests");

/ //run() is called by the main thread

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import CalcApp.\*;

import CalcApp.CalcPackage.DivisionByZero;

import org.omg.CosNaming.\*;

import org.omg.CosNaming.NamingContextPackage.\*;

import org.omg.CORBA.\*;

import static java.lang.System.out;

public class CalcClient

{

static Calc calcImpl;

static BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

public static void main(String args[])

{

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 = "Calc";

calcImpl = CalcHelper.narrow(ncRef.resolve\_str(name));

// System.out.println(calcImpl);

while (true) {

out.println("1. Sum");

out.println("2. Sub");

out.println("3. Mul");

out.println("4. Div");

out.println("5. exit");

out.println("--");

out.println("choice: ");

try {

String opt = br.readLine();

if (opt.equals("5")) {

break;

}

else if (opt.equals("1"))

{

out.println("a+b= " + calcImpl.sum(getFloat("a"), getFloat("b"))); }

else if (opt.equals("2"))

{

out.println("a-b= " + calcImpl.sub(getFloat("a"), getFloat("b"))); }

else if (opt.equals("3"))

{

out.println("a\*b= " + calcImpl.mul(getFloat("a"), getFloat("b"))); }

else if (opt.equals("4"))

{

try {

out.println("a/b= " + calcImpl.div(getFloat("a"), getFloat("b")));

}

catch (DivisionByZero de)

{

out.println("Division by zero!!!"); }

}

}

catch (Exception e)

{

out.println("===");

out.println("Error with numbers");

out.println("===");

}

out.println("");

}

//calcImpl.shutdown();

} catch (Exception e) {

System.out.println("ERROR : " + e);

e.printStackTrace(System.out);

}

}

static float getFloat(String number) throws Exception

{

out.print(number + ": ");

return Float.parseFloat(br.readLine());

}

}

**OUTPUT::**









