CMR College of Engineering & Technology
Kandlakoya (V), Medchal Road, Hyderabad - 501 401. Andhra Pradesh. INDIA
Phone No: 08418 - 200699. Fax No: 08418 - 200240.
E-Mail : principal@cmrcet.org , www.cmrcet.org

Department of Computer Science & Engineering

M.Tech(CSE)-I Year-II Semester

WEB SERVICES AND SERVICE ORIENTED ARCHITECHTURE

(B1513)

By

Mr.K.Yellaswamy

Assistant Professor

(B1513) WEB SERVICES AND SERVICE ORIENTED ARCHITECTURE

UNIT-1

I. Evolution and Emergence of Web Services
   a. Evolution of distributed computing
   b. Core distributed computing technologies
      i. Client/Server
      ii. CORBA
      iii. Java RMI
      iv. Microsoft DCOM
      v. Message-Oriented Middleware
   c. Common challenges in Distributed Computing
   d. Role of J2EE and XML in distributed computing
   e. Emergence of Web Services and Service Oriented Architecture(SOA)

II. Introduction to Web Services
   a. The Definition of web services
   b. Basic operational model of web services
   c. Tools and Technologies enabling web services
   d. Benefits and challenges of using web services

Lecture No: 1

Distributed Computing:

Distributed Computing is a type of computing in which different components and objects comprising an application can be located on different computers connected to a Network.

Example:
Java RMI (Remote Method Invocation)
OMG CORBA (Common Object Request Broker Architecture)
Microsoft DCOM (Distributed Component Object Model)
MOM (Message –oriented Middleware)

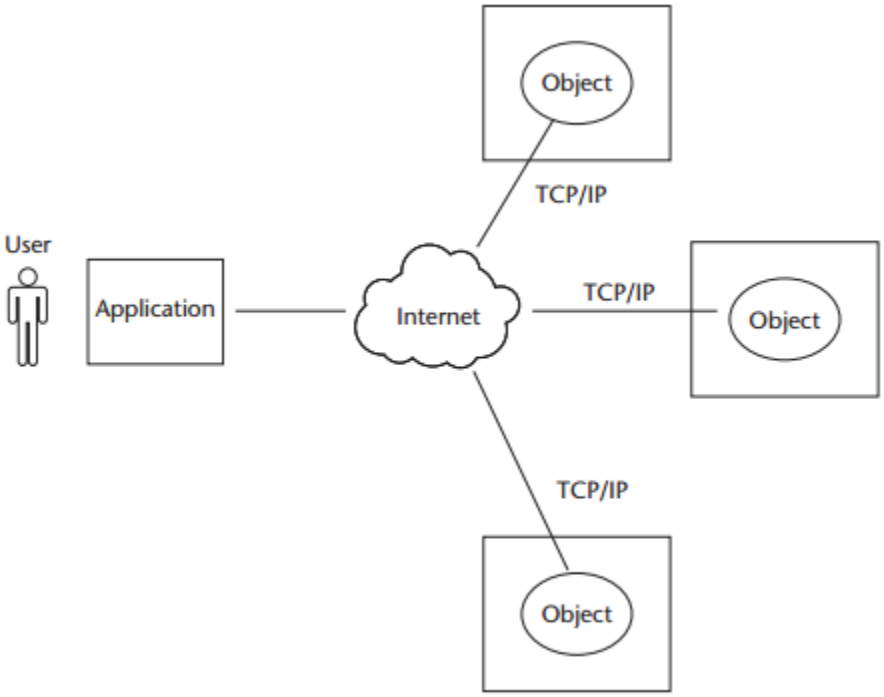


Figure 1.1 Internet-based distributed computing model.

Advantages of Distributed Computing:

- Higher Performance
- Collaboration
- Higher reliability and availability
- Scalability
- Extensibility
- Higher Productivity and lower development cycle time.
- Reuse

- Reduced Cost

Client-Server Applications:

In a two-tier architecture model,

- the first (Upper) tier handles the Presentation and business logic of the user application(Client).
- The Second/Lower tier handles the application organization and its data storage(Server).

This approach is commonly called client-server application Architecture.

- Generally the server in a Client/Server application model is a database server that mainly responsible for the organization and retrieval of data.
- The application client in this model handles most of the business processing and provides the graphical user interface of the application.

Some of the common limitations of the client-server application model are as follows:

- Complex business processing at the client side demands robust client systems.
- Security is more difficult to implement because the algorithms andlogic reside on the client side making it more vulnerable to hacking.
- Increased network bandwidth is needed to accommodate many calls to the server, which can impose scalability restrictions.
- Maintenance and upgrades of client applications are extremely difficult because each client has to be maintained separately.
- Client-server architecture suits mostly database-oriented standalone applications and does not target robust reusable componentoriented applications.

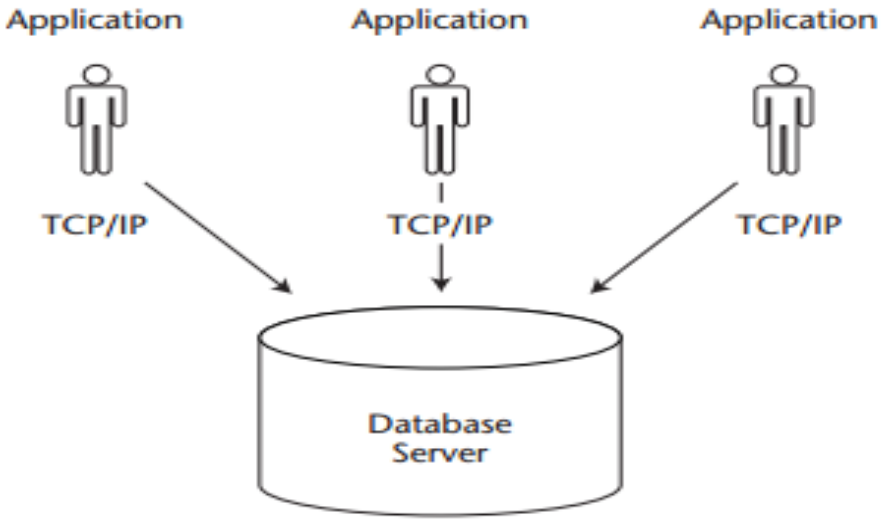


Figure 1.2 An example of a client-server application.

Lecture No: 2

CORBA:

Though RMI is a powerful mechanism for distributing and processing objects in a platform-independent manner, it has one significant drawback – it only works with objects that have been created using Java. Convenient though it might be if Java were the only language used for creating software objects, this simply is not the case in the real world. A more generic approach to the development of distributed systems is offered by CORBA (Common Object Request Broker Architecture),which allows objects written in a variety of programming languages to be accessed by client programs which themselves may be written in a variety of programming languages.

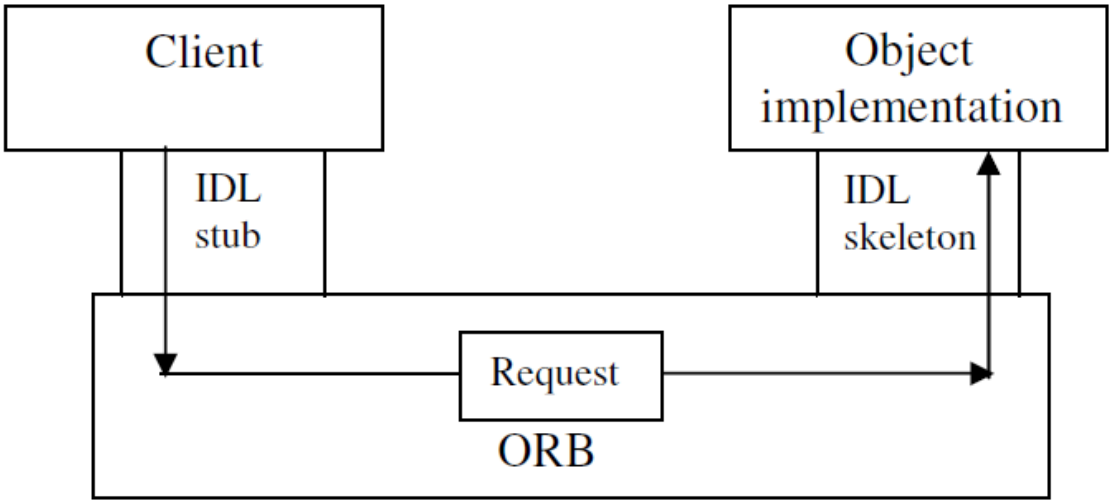


Figure 6.1 Remote method invocation when client and server are using the same ORB.

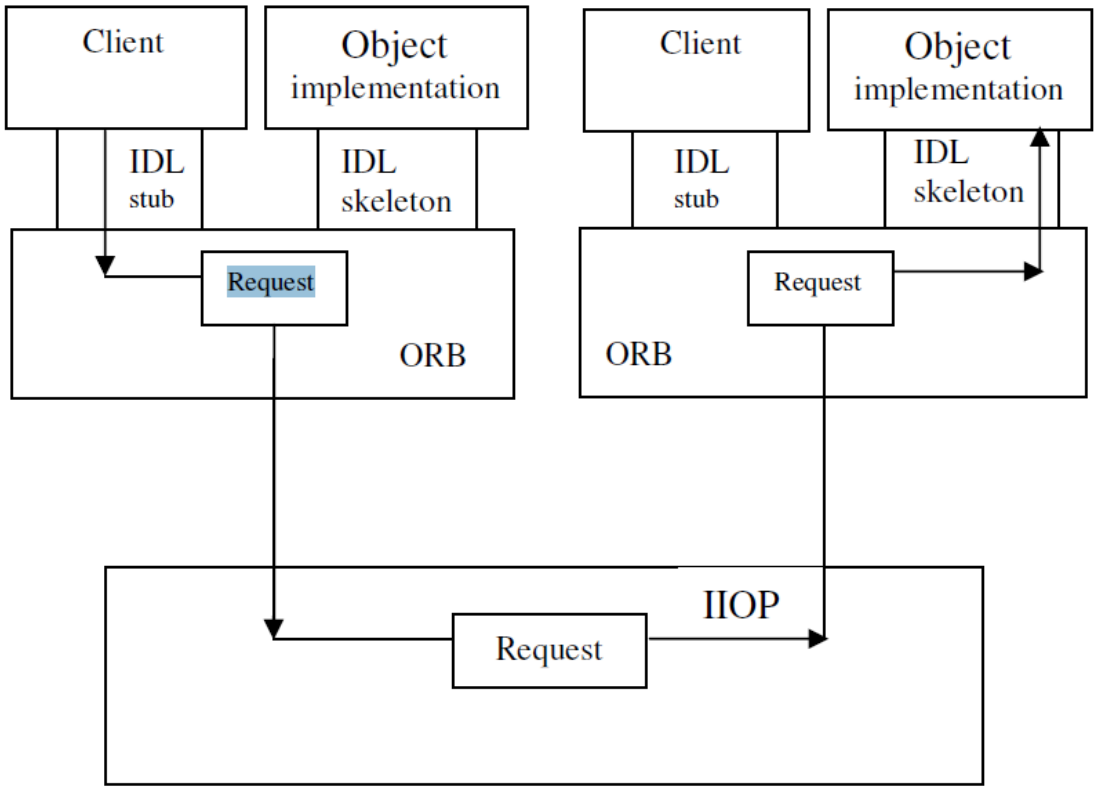


Figure 6.2 Remote invocation when client and server are using different ORBs.

There are several steps required to set up a CORBA client/server application. These steps are listed below:

- Use the idlj compiler to compile the above file, generating up to six files for each interface defined.

2. Implement each interface as a 'servant'.
3. Create the server (incorporating servants).
4. Compile the server and the idlj-generated files.
5. Create a client.
6. Compile the client.
7. Run the application.

1. Create the IDL file.

The file will be called *Hello.idl* and will hold a module called CmrcetMTechCORBAExample. This module will contain a single interface called *Hello* that holds the signature for operation *getGreeting* . The contents of this fi le are shown below.

//Hello.idl

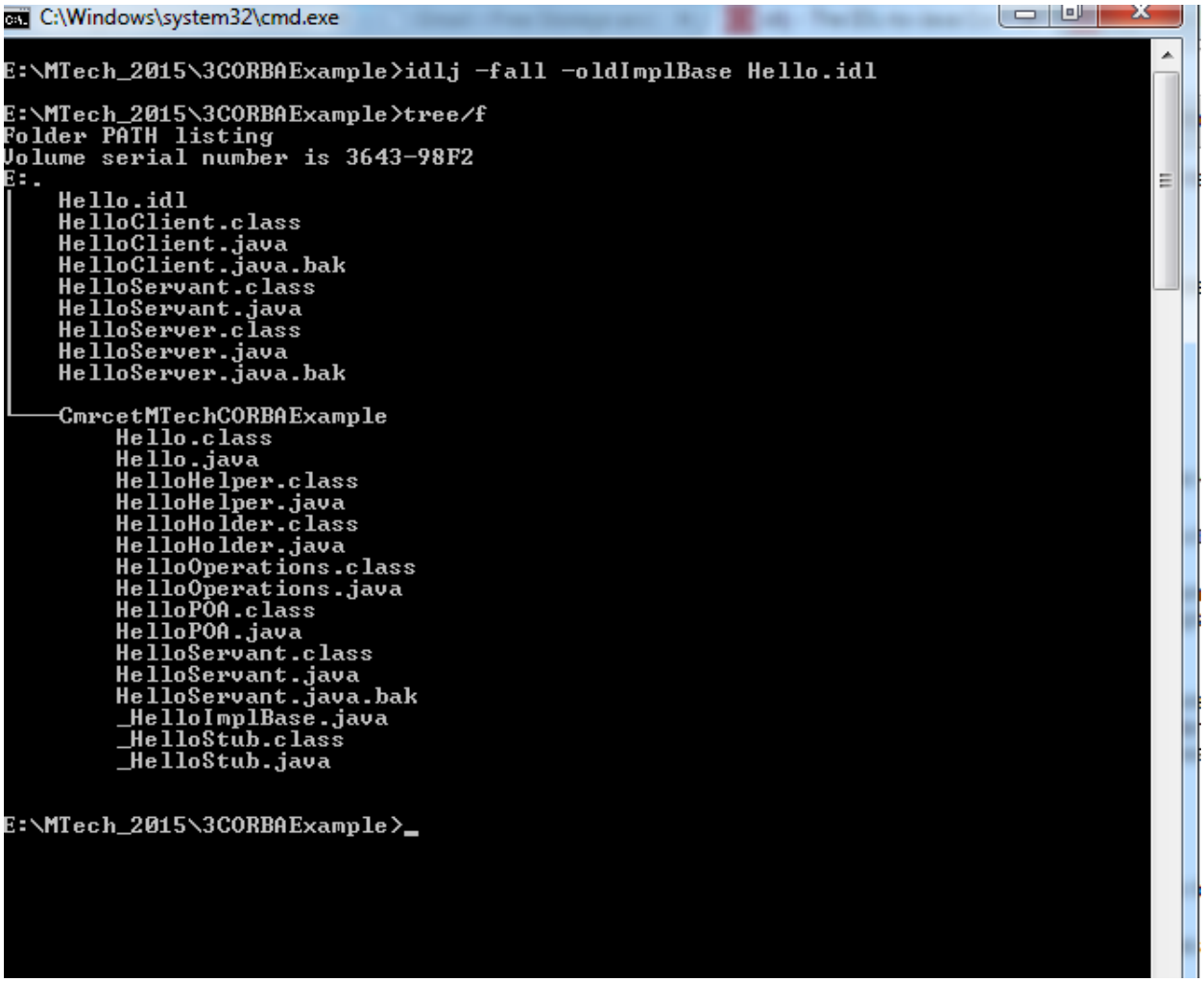
```
module CmrcetMTechCORBAExample
{
interface Hello
{
string getGreeting();
};
};
```

2. Compile the IDL fi le.

The *idlj* compiler defaults to generating only the client-side bindings. To vary this default behaviour, the *–f* option may be used. This is followed by one of three possible specifi ers: *client* , *server* and *all* . If client and server are to be run on the same machine, then ***all*** is appropriate and the following command line should be entered:

idlj –fall Hello.idl

This causes a sub-directory with the same name as the module (i.e., CmrcetMTechCORBAExample) to be created, holding the **six files** listed below.



• *Hello.java*

Contains the Java version of our IDL interface. It extends interface *HelloOperations* [See below], as well as org .*omg.CORBA.Object* (providing standard CORBA object functionality) and *org.omg.CORBA.portable.IDLEntity* .

• *HelloHelper.java*

Provides auxiliary functionality, notably the *narrow* method required to cast CORBA object references into *Hello* references.

• *HelloHolder.java*

Holds a public instance member of type *Hello* . If there were any out or inout arguments (which CORBA allows, but which do not map easily onto Java), this file would also provide operations for them.

• *HelloOperations.java*

Contains the Java method signatures for all operations in our IDL fi le. In this application, it contains the single method *getGreeting* .*HelloImplBase.java* .An abstract class comprising the server skeleton. It provides basic CORBA functionality for the server and implements the *Hello* interface. Each servant (interface implementation) that we create for this service must extend *HelloImplBase* .

• *HelloStub.java*

This is the client stub, providing CORBA functionality for the client. Like *HelloImplBase.java* , it implements the *Hello* interface.Prior to J2SE 1.3, the method signatures would have been specifi ed within *Hello.java* , but are now held within *HelloOperations.java* .

3. Implement the interface.

Here, we specify the Java implementation of our IDL interface. The implementation of an interface is called a 'servant', so we shall name our implementation classHelloServant. This class must extend _HelloImplBase.Here is the code:

```
// HelloServant.java
package CmrcetMTechCORBAExample;
class HelloServant extends _HelloImplBase
{
public String getGreeting()
{
return ("Welcome to CORBA Programming!");
}
}
```

This class will be placed inside the same file as our server code.

4. Create the server.

Our server program will be called HelloServer.java and will subsume the servant created in the last step. It will reside in the directory immediately above directory CmrcetMTechCORBAExample and will import package CmrcetMTechCORBAExample and the following three standard CORBA packages:

- org.omg.CosNaming (for the naming service);
- org.omg.CosNaming.NamingContextPackage (for specialexceptions thrown by the naming service);
- org.omg.CORBA (needed by all CORBA applications).

There are several steps required of the server…

(i) Create and initialize the ORB.

This is affected by calling static method init of class ORB (from package org.omg.CORBA). This method takes two arguments: a String array and a Properties object. The first of these is usually set to the argument list received by main, while the second is almost invariably set to null:

ORB orb = ORB.init(args,null);

[The argument args is not used here (or in many other such programs) in a Windows

environment, but it is simpler to supply it, since replacing it with null causes an

error message, due to ambiguity with an overloaded form of init that takes an Applet

argument and a Properties argument.]

(ii) Create a servant.

Easy enough:

HelloServant servant = new HelloServant();

(iii) Register the servant with the ORB.

This allows the ORB to pass invocations to the servant and is achieved by means of the ORB class's connect method:

orb.connect(servant);

(iv) Get a reference to the root naming context.

Method resolve_initial_references of class ORB is called with the String argument "NameService" (defined for all CORBA ORBs) and returns a CORBA Object reference that points to the naming context:

org.omg.CORBA.Object objectRef =orb.resolve_initial_references("NameService");

(v) 'Narrow' the context reference.

In order for the generic Object reference from the previous step to be usable, it must be 'narrowed' (i.e., typecast 'down' into its appropriate type). This is achieved by the use of method narrow of class NamingContextHelper (from package org.omg.CosNaming):

NamingContext namingContext = NamingContextHelper.narrow(objectRef);

(vi) Create a NameComponent object for our interface.

The NameComponent constructor takes two String arguments, the first of which supplies a name for our service. The second argument can be used to specify a category (usually referred to as a 'kind') for the first argument, but is typically left as an empty string. In our example, the service will be called 'Hello':

```
NameComponent nameComp = new NameComponent("Hello", "");
```

(vii) Specify the path to the interface.

This is effected by creating an array of NameComponent objects, each of which is a component of the path (in 'descending' order), with the last component specifying the name of the NameComponent reference that points to the service. For a service in the same directory, the array will contain a single element, as shown below.

```
NameComponent[] path = (nameComp};
```

(viii) Bind the servant to the interface path.

The rebind method of the NamingContext object created earlier is called with arguments that specify the path and service respectively:

```
namingContext.rebind(path,servant);
```

(ix) Wait for client calls.

Unlike our previous server programs, this is not achieved via an explicitly 'infinite'

loop. A call is made to method wait of (Java class) Object. This call is isolated within a code block that is declared synchronized, as shown below.

```
java.lang.Object syncObj = new java.lang.Object();

synchronized(syncObj)

{

syncObj.wait();

}
```

All of the above code will be contained in the server's main method. Since various

CORBA system exceptions may be generated; all the executable code will be held

Within a try block.

// HelloServer.java

```
import CmrcetMTechCORBAExample.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class HelloServer
{
public static void main(String[] args)
{
try
{
ORB orb = ORB.init(args,null);
HelloServant servant = new HelloServant();
orb.connect(servant);
org.omg.CORBA.Object objectRef =orb.resolve_initial_references("NameService");
NamingContext namingContext =NamingContextHelper.narrow(objectRef);
NameComponent nameComp =new NameComponent("Hello", "");
NameComponent[] path = {nameComp};
namingContext.rebind(path,servant);
java.lang.Object syncObj =new java.lang.Object();
synchronized(syncObj)
{
syncObj.wait();
}
}
```

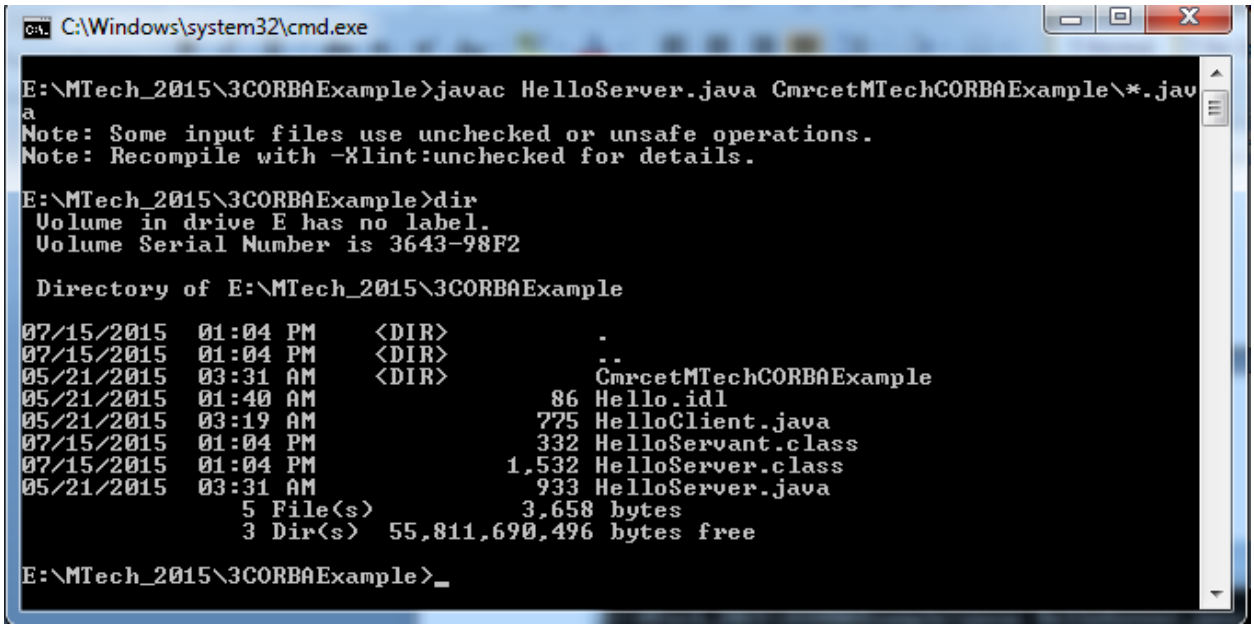
```
catch (Exception ex)
{
System.out.println("*** Server error! ***");
ex.printStackTrace();
}
}
}

class HelloServant extends _HelloImplBase
{
public String getGreeting()
{
return ("Welcome to CORBA Programming!");
}
}
```

5. Compile the server and the idlj-generated files.

From the directory above directory CmrcetMTechCORBAExample, execute the following command within a command window:

javac HelloServer.java CmrcetMTechCORBAExample *.java

(Correct errors and recompile, as necessary.)



6. Create a client.

Our client program will be called HelloClient.java and, like the server program, will import package CmrcetMTechCORBAExample. It should also import two of the three CORBA packages imported by the server: org.omg.CosNaming and org.omg.CORBA. There are several steps required of the client, most of them being identical to those required of the server, so the explanations given for the server in step 4 above are not repeated here…

(i) Create and initialise the ORB.

ORB orb = ORB.init(args,null);

(ii) Get a reference to the root naming context.

org.omg.CORBA.Object objectRef =orb.resolve_initial_references("NameService");

(iii) 'Narrow' the context reference.

NamingContext namingContext =NamingContextHelper.narrow(objectRef);

(iv) Create a NameComponent object for our interface.

NameComponent nameComp =new NameComponent("Hello", "");

(v) Specify the path to the interface.

NameComponent[] path = (nameComp};

(vi) Get a reference to the interface.

This is achieved by passing the above interface path to our naming context's resolve method, which returns a CORBA Object reference:

org.omg.CORBA.Object objectRef =namingContext.resolve(path);

(vii) 'Narrow' the interface reference.

We 'downcast' the reference from the previous step into a Hello reference via static method narrow of the idlj-generated class HelloHelper:

Hello helloRef = HelloHelper.narrow(objectRef);

(viii) Invoke the required method(s) and display results.

We use the reference from the preceding step to invoke the required method, just as though the call were being made to a local object:

System.out.println("Message received: "+ greeting);

As was the case with the server, our client may then generate CORBA system exceptions, and so all the executable code will be placed inside a try block.

The full program is shown below.

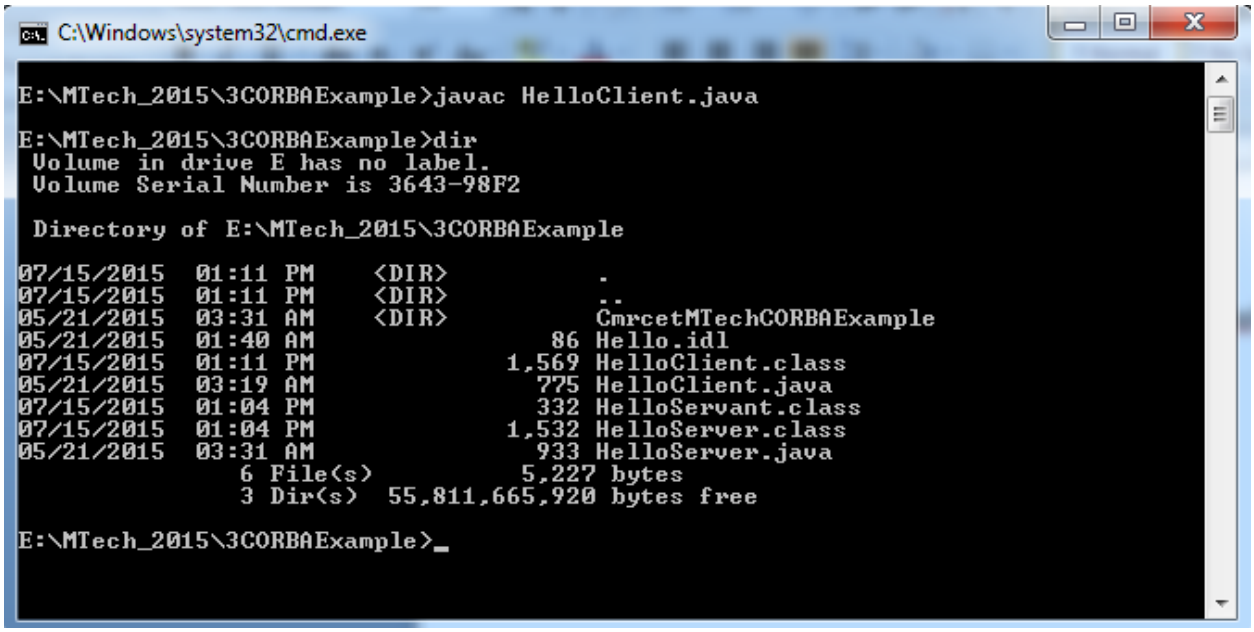
```
// HelloClient.java
import CmrcetMTechCORBAExample.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
public class HelloClient
{
public static void main(String[] args)
{
try
```

```
{
ORB orb = ORB.init(args,null);
org.omg.CORBA.Object objectRef =orb.resolve_initial_references("NameService");
NamingContext namingContext =NamingContextHelper.narrow(objectRef);
NameComponent nameComp =new NameComponent("Hello", "");
NameComponent[] path = {nameComp};
//Re-use existing object reference...
objectRef = namingContext.resolve(path);
Hello helloRef = HelloHelper.narrow(objectRef);
String greeting = helloRef.getGreeting();

System.out.println("Message received: "+ greeting);
}
catch (Exception ex)
{
System.out.println("*** Client error! ***");
ex.printStackTrace();
}
}
}
```

7. Compile the client.

From the directory above directory SimpleCORBAExample, execute the following command:

javac HelloClient.java



8. Run the application.

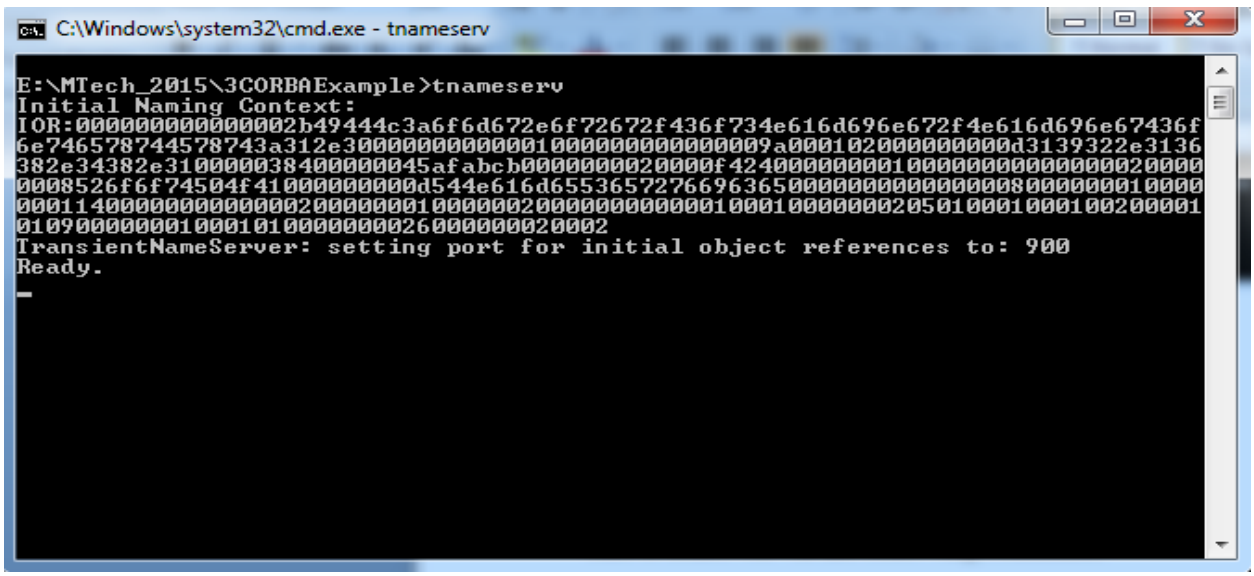
This requires three steps...

(i) Start the CORBA naming service.

This is achieved via the following command:

tnameserv

Starting the CORBA naming service under Java IDL.



The above command starts up the Java IDL Transient Nameservice as an object server that assumes a default port of 900. To use a different port (which would normally be necessary under Sun's Solaris operating system for ports below 1024),use the ORBInitialPort option to specify the port number.

For example:

tnameserv -ORBInitialPort 1234

(ii) Start the server in a new command window.

For our example program, the command will be:

java HelloServer

(Since there is no screen output from the server, no screenshot is shown here.)

Again, a port other than the default one can be specified. For example:

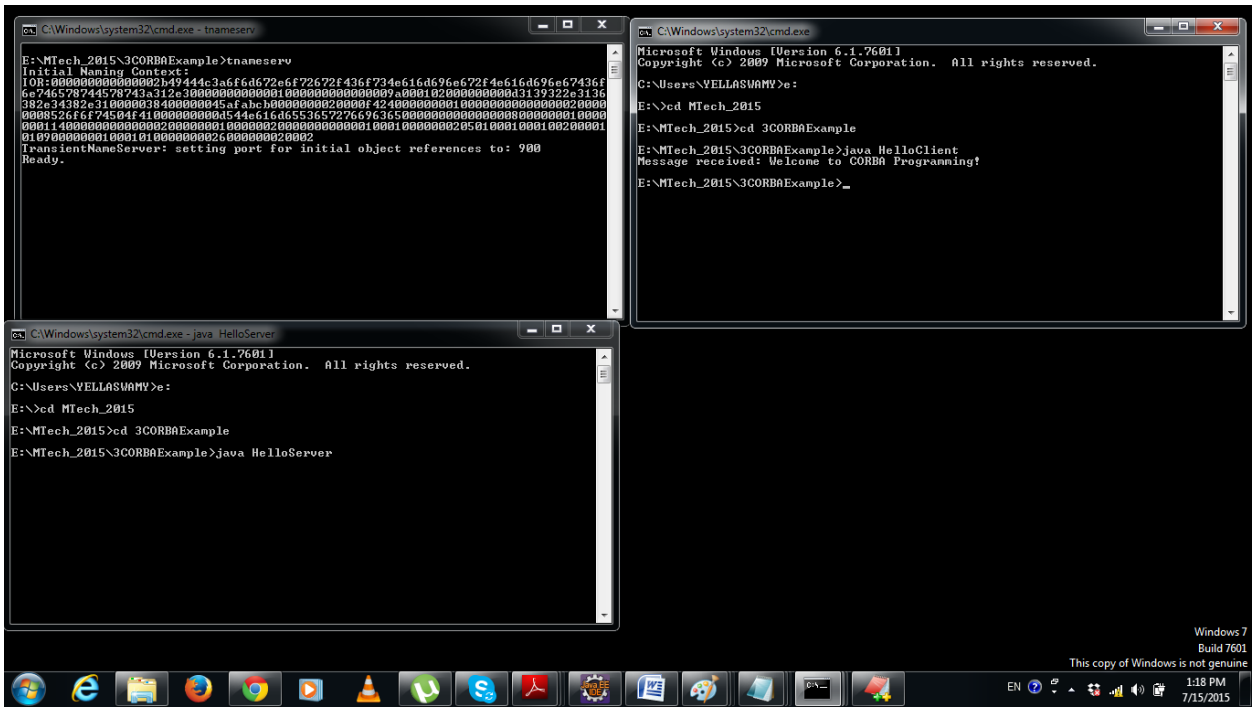
java HelloServer -ORBInitialPort 1234

(iii) Start the client in a third command window.

For our example program, the command will be:

java HelloClient

(As above, a non-default port can be specified.)



Lecture No: 3

Remote Method Invocation (RMI)

RMI is distributed technology which provides java to java communication over a network in RMI. RMI registry is mediator to establish client to server connection. RMI is part of javase.

With RMI we get the following problems.

1. RMI is small api. So we cannot develop complete application.

2. RMI is distributed homogeneous technology.

3. RMI protocol is not a firewall friendly protocol.

A client program can then use the same naming service to obtain a reference to this interface in the form of what is called a **stub**.

This stub is effectively a local surrogate (a 'stand-in' or placeholder) for the remote object.

On the remote system, there will be another surrogate called a **skeleton**.

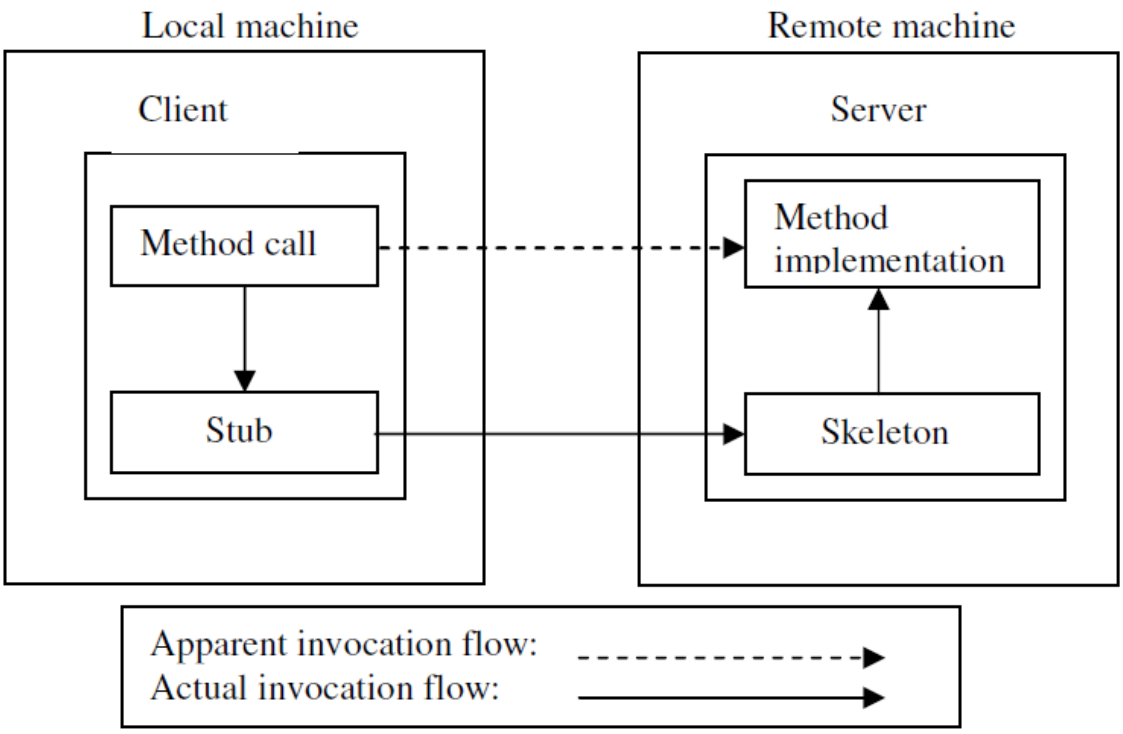


Figure 5.1 Using RMI to invoke a method of a remote object.

Implementation Details

The packages used in the implementation of an RMI client-server application are java.rmi, java.rmi.server and java.rmi.registry, though only the first two need to be used explicitly. The basic steps are listed below.

1. Create the interface.

2. Define a class that implements this interface.

3. Create the server process.

4. Create the client process.

/*The packages used in the implementation of an RMI client-server application are

java.rmi

java.rmi.server

java.rmi.registry

```
 */


 //Steps
 //step1:Create the Interface
 //step2:define a class that implements this interface
 //step3:Create the server process
 //step4:Create the client process
//step1:create the interface
import java.rmi.*;
public interface HelloInterface extends Remote
{       public String sayHello() throws RemoteException;
}



//step2:define a class that implements this interface
import java.rmi.*;
import java.rmi.server.*;
public class HelloInterfaceImpl extends UnicastRemoteObject implements HelloInterface
{
        //constructor

        public HelloInterfaceImpl() throws RemoteException
        {
                //No action Need here
        }
```

```
        public String sayHello() throws RemoteException
        {
                return ("Hello,Welcome to RMI Programming");
        }

}
```

```
//step3:Create the server process
import java.rmi.*;
public class HelloServer
{
private static final String HOST="localhost";
public static void main(String args[]) throws Exception
{
//create reference to an implementation object
HelloInterfaceImpl ref=new HelloInterfaceImpl();

//create the string URL holding the object's name..
```

```
String rmiObjectName="rmi://"+HOST+"/HelloInterface";



//Bind the object reference to the name...

Naming.rebind(rmiObjectName,ref);


//Display a message so that we know the process has been completed


System.out.println("Binding Complete....\n");

}

}
```

//step4:Create the client process

```
import java.rmi.*;
public class HelloClient
{
private static final String HOST="localhost";

public static void main(String args[])
        {
                try
                {
//obtain a reference to the object from the registry and typecast it into the appropriate type
                HelloInterface
msg=(HelloInterface)Naming.lookup("rmi://"+HOST+"/HelloInterface");

//use the above reference to invoke the remote object's method.....

System.out.println("Message Received:"+msg.sayHello());
```

```
                }
                catch (ConnectException conEx)
                {
                        System.out.println("Unable to connect to Server");
                        System.exit(1);
                }
                catch(Exception ex)
                {
                        ex.printStackTrace();
                        System.exit(1);
                }
        }

}
```

Compilation and Execution

---

1)Compile all files with javac

```
E:\RMIExample>javac *.java
```

---

2)compile the implementation class with the rmic compiler.This will cause a file with the name HelloInterfaceImpl_Stub.class to be created.

E:\IIICSEA\RMIExample>rmic HelloInterfaceImpl
Warning: generation and use of skeletons and static stubs for JRMP is deprecated. Skeletons are unnecessary, and static stubs have been superseded by dynamically generated stubs. Users are encouraged to migrate away from using rmic to generate skeletons and static stubs. See the documentation for java.rmi.server.UnicastRemoteObject.

E:\RMIExample>tree/f
Folder PATH listing
Volume serial number is 3643-98F2
E:.
    HelloClient.class
    HelloClient.java
    HelloInterface.class
    HelloInterface.java
    HelloInterfaceImpl.class
    HelloInterfaceImpl.java
    HelloInterfaceImpl_Stub.class
    HelloServer.class
    HelloServer.java

No subfolders exist
3)start the RMI registry

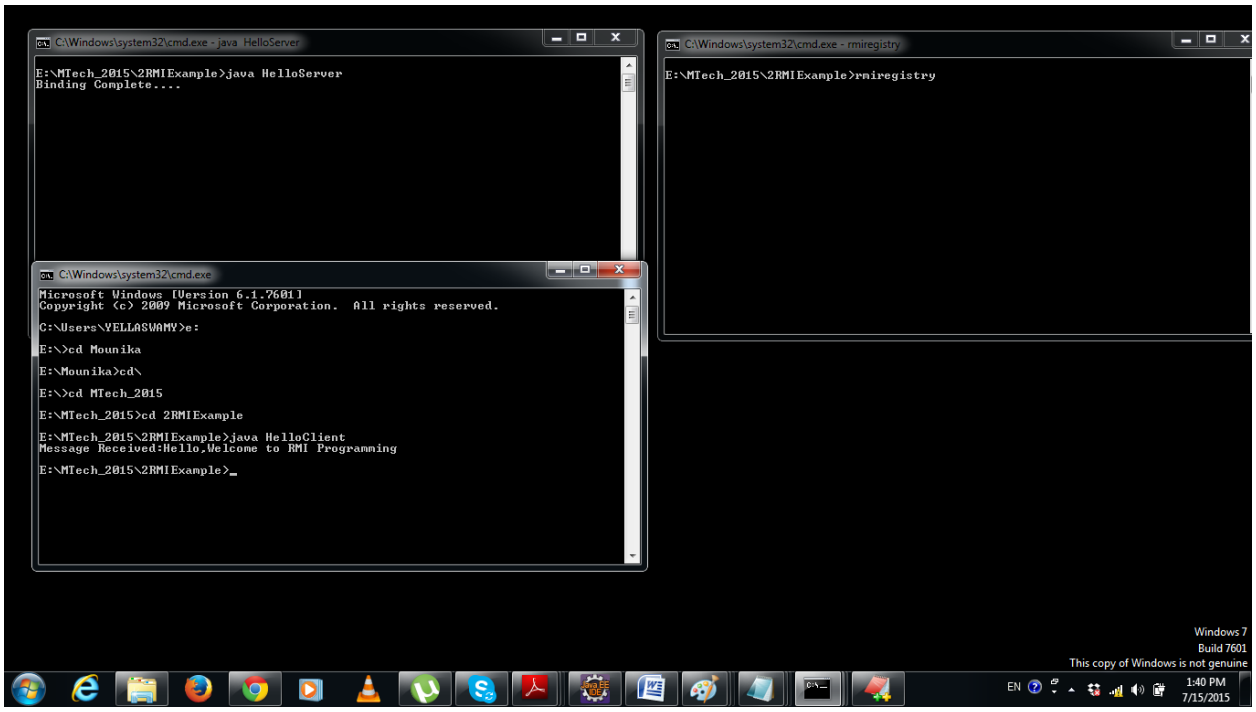
E:\RMIExample>rmiregistry

4) open a new window and run the server

E:\RMIExample>java HelloServer

5)open a third window and run the client

E:\RMIExample>java HelloClient


C:\Windows\system32\cmd.exe - java HelloServer
E:\MTech_2015\2RMIExample>java HelloServer
Binding Complete....
C:\Windows\system32\cmd.exe - rmiregistry
E:\MTech_2015\2RMIExample>rmiregistry
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\YELLASWAMY>e:
E:\>cd Mounika
E:\Mounika>cd\
E:\>cd MTech_2015
E:\MTech_2015>cd 2RMIExample
E:\MTech_2015\2RMIExample>java HelloClient
Message Received:Hello,Welcome to RMI Programming
E:\MTech_2015\2RMIExample>
Windows 7
Build 7601
This copy of Windows is not genuine
1:40 PM
7/15/2015