## Lab Manual

**COMPUTER LABORATORY-IX**

**(Distributed Computing System)**

## ASSIGNMENT NO.1 A

**Problem Statement** 1a):

## To develop any distributed application through implementing client-server communication programs based on Java Sockets.

**Tools / Environment:**

Java Programming Environment, rmiregistry, jdk 1.8, Eclipse IDE.

## Related Theory:

**Socket:** In distributed computing, network communication is one of the essential parts of any system, and the socket is the endpoint of every instance of network communication. In Java communication, it is the most critical and basic object involved.

A socket is a handle that a local program can pass to the networking API to connect to another machine. It can be defined as *the terminal of a communication link through which two programs*

*/processes/threads running on the network can communicate with each other. The TCP layer can easily identify the application location and access information through the port number assigned to the respective sockets.*

During an instance of communication, a client program creates a socket at its end and tries to connect it to the socket on the server. When the connection is made, the server creates a socket at its end and then server and client communication is established.

## Designing the solution:

The **java.net** package provides classes to facilitate the functionalities required for networking. The **socket** class programmed through Java using this package has the capacity of being independent of the platform of execution; also, it abstracts the calls specific to the operating system on which it is invoked from other Java interfaces. The **ServerSocket** class offers to observe connection invocations, and it accepts such invocations from different clients through another socket. High-level wrapper classes, such as **URLConnection** and **URLEncoder**, are more appropriate. If you want to establish a connection to the Web using a URL, then these classes will use the socket internally.

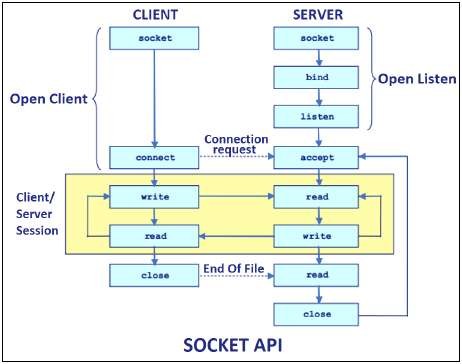
The **java.net** package provides support for the two common network protocols −

* **TCP** − TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
* **UDP** − UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

## Socket programming for TCP:

The following steps occur when establishing a TCP connection between two computers using sockets −

* + The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
  + The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
  + After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.
  + The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.
  + On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.
  + After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.
  + TCP is a two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets.



## Socket programming for UDP:

UDP is used only when the entire information can be bundled into a single packet and there is no dependency on the other packet. Therefore, the usage of UDP is quite limited, whereas TCP is

widely used in IP applications. UDP sockets are used where limited bandwidth is available, and the overhead associated with resending packets is not acceptable.

To connect using a UDP socket on a specific port, use the following code:

DatagramSocket udpSock = new DatagramSocket(3000);

A datagram is a self-contained, independent message whose time of arrival, confirmation of arrival over the network, and content cannot be guaranteed. atagramPacket objects are used to send data over DatagramSocket. Every DatagramPacket object consists of a data buffer, a remote host to whom the data needs to be sent, and a port number on which the remote agent would be listened.

## Writing the source code:

The socket program using TCP/UDP will demonstrate how data can be sent and received between the client and the server.

## Compilation and Executing the solution: If you’re using Eclipse :

1. Compile both of them on two different terminals or tabs
2. Run the Server program first
3. Then run the Client program
4. Type messages in the Client Window which will be received and showed by the Server Window simultaneously if you are developing echo server application.
5. Close the socket connection by typing something like "Exit".

## Conclusion:

In this assignment, the students learned about client-server communication through different protocols and sockets. They also learned about Java support through the socket API for TCP and UDP programming.

## ASSIGNMENT NO. 1 B

**Problem Statement:**

**To develop any distributed application through implementing client-server communication programs based on Java RMI .**

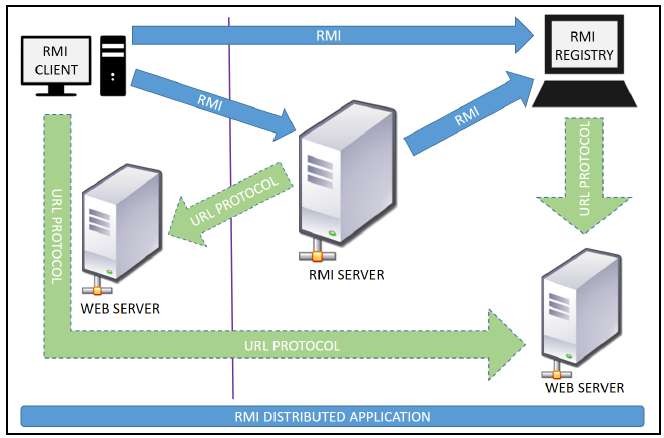
**Tools / Environment:**

Java Programming Environment, jdk 1.8, rmiregistry

## Related Theory:

RMI provides communication between java applications that are deployed on different servers and connected remotely using objects called **stub** and **skeleton**. This communication architecture makes a distributed application seem like a group of objects communicating across a remote connection. These objects are encapsulated by exposing an interface, which helps access the private state and behavior of an object through its methods.

The following diagram shows how RMI happens between the RMI client and RMI server with the help of the RMI registry:



**RMI REGISTRY** is a remote object registry, a Bootstrap naming service, that is used by **RMI SERVER** on the same host to bind remote objects to names. Clients on local and remote hosts then look up the remote objects and make remote method invocations.

## Key terminologies of RMI:

The following are some of the important terminologies used in a Remote Method Invocation. **Remote object**: This is an object in a specific JVM whose methods are exposed so they could be invoked by another program deployed on a different JVM.

**Remote interface**: This is a Java interface that defines the methods that exist in a remote object. A remote object can implement more than one remote interface to adopt multiple remote interface behaviors.

**RMI**: This is a way of invoking a remote object's methods with the help of a remote interface. It can be carried with a syntax that is similar to the local method invocation.

**Stub**: This is a Java object that acts as an entry point for the client object to route any outgoing requests. It exists on the client JVM and represents the handle to the remote object.

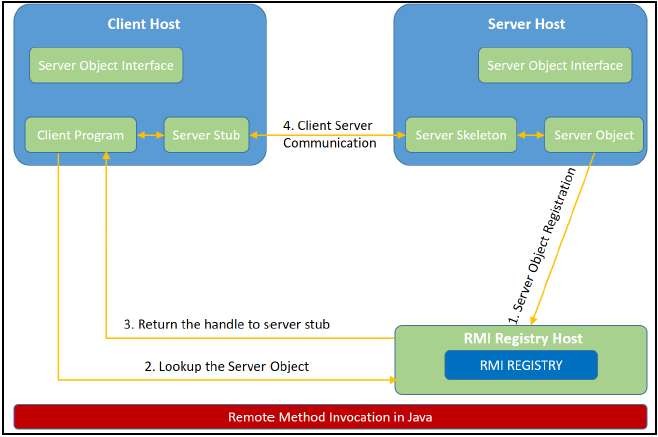
If any object invokes a method on the stub object, the stub establishes RMI by following these steps:

* 1. It initiates a connection to the remote machine JVM.
  2. It marshals (write and transmit) the parameters passed to it via the remote JVM.
  3. It waits for a response from the remote object and unmarshals (read) the returned value or exception, then it responds to the caller with that value or exception.

**Skeleton**: This is an object that behaves like a gateway on the server side. It acts as a remote object with which the client objects interact through the stub. This means that any requests coming from the remote client are routed through it. If the skeleton receives a request, it establishes RMI through these steps:

1. It reads the parameter sent to the remote method.
2. It invokes the actual remote object method.
3. It marshals (writes and transmits) the result back to the caller (stub).

The following diagram demonstrates RMI communication with stub and skeleton involved:



## Designing the solution:

The essential steps that need to be followed to develop a distributed application with RMI are as follows:

1. Design and implement a component that should not only be involved in the distributed application, but also the local components.
2. Ensure that the components that participate in the RMI calls are accessible across networks.
3. Establish a network connection between applications that need to interact using the RMI.

**Remote interface definition**: The purpose of defining a remote interface is to declare the methods that should be available for invocation by a remote client.

Programming the interface instead of programming the component implementation is an essential design principle adopted by all modern Java frameworks, including Spring. In the same pattern, the definition of a remote interface takes importance in RMI design as well.

1. **Remote object implementation**: Java allows a class to implement more than one interface at a time. This helps remote objects implement one or more remote interfaces. The remote object class may have to implement other local interfaces and methods that it is responsible for. Avoid adding complexity to this scenario, in terms of how the arguments or return parameter values of such component methods should be written.
2. **Remote client implementation**: Client objects that interact with remote server objects can be written once the remote interfaces are carefully defined even after the remote objects are deployed.

Let's design a project that can sit on a server. After that different client projects interact with this project to pass the parameters and get the computation on the remote object execute and return the result to the client components.

## Implementing the solution:

**Consider building an application to perform diverse mathematical operations.**

The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.

## Creating remote interface, implement remote interface, server-side and client-side program and Compile the code.

This application uses four source files. The first file, **AddServerIntf.java**, defines the remote interface that is provided by the server. It contains one method that accepts two **double** arguments and returns their sum. All remote interfaces must extend the **Remote** interface, which is part of **java.rmi**. **Remote** defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a **RemoteException**.

The second source file, **AddServerImpl.java**, implements the remote interface. The implementation of the **add()** method is straightforward. All remote objects must extend **UnicastRemoteObject**, which provides functionality that is needed to make objects available from remote machines.

The third source file, **AddServer.java**, contains the main program for the server machine. Its primary function is **to update the RMI registry on that machine**. This is done by using the **rebind()** method of the **Naming** class (found in **java.rmi**). That method associates a name with an object reference. The first argument to the **rebind()** method is a string that names the server as “AddServer”. Its second argument is a reference to an instance of **AddServerImpl**.

The fourth source file, **AddClient.java**, implements the client side of this distributed application. **AddClient.java** requires three command-line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URL uses the **rmi** protocol. The string includes the IP address or name of the server and the string “AddServer”. The program then invokes the **lookup( )** method of the **Naming** class. This method accepts one argument, the **rmi** URL, and returns a reference to an object of type **AddServerIntf**. All remote method invocations can then be directed to this object.

The program continues by displaying its arguments and then invokes the remote **add()** method. The sum is returned from this method and is then printed.

Use **javac** to compile the four source files that are created.

## Generate a Stub

Before using client and server, the necessary stub must be generated. In the context of RMI, a

*stub* is a Java object that resides on the client machine. Its function is to present the same

interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine.

All of this information must be sent to the remote machine. That is, an object passed as an argument to a remote method call must be serialized and sent to the remote machine. If a response must be returned to the client, the process works in reverse. **The serialization and deserialization facilities are also used if objects are returned to a client.**

To generate a stub the command is RMIcompiler is invoked as follows:

## rmic AddServerImpl.

This command generates the file **AddServerImpl\_Stub.class**.

## Install Files on the Client and Server Machines

Copy **AddClient.class**, **AddServerImpl\_Stub.class**, **AddServerIntf.class**

to a directory on the client machine.

Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl\_ Stub.class**, and **AddServer.class** to a directory on the server machine.

## Start the RMI Registry on the Server Machine

Java provides a program called **rmiregistry**, which executes on the server machine. It maps names to object references. Start the RMI Registry from the command line, as shown here:

start rmiregistry

## Start the Server

The server code is started from the command line, as shown here:

java AddServer

The **AddServer** code instantiates **AddServerImpl** and registers that object with the name “AddServer”.

## Start the Client

The **AddClient** software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. You may invoke it from the command line by using one of the two formats shown here:

java AddClient 192.168.13.14 7 8

## Conclusion:

Remote Method Invocation (RMI) allows you to build Java applications that are distributed among several machines. Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This is an important feature, because it allows you to build distributed applications.

## ASSIGNMENT NO. 2

**Problem Statement:**

To develop any distributed application using Message Passing Interface (MPI).

## Tools / Environment:

Java Programming Environment, JDK1.8 or higher, MPI Library (mpi.jar), MPJ Express (mpj.jar)

## Related Theory:

**Message passing** is a popularly renowned mechanism to implement parallelism in applications; it is also called MPI. The MPI interface for Java has a technique for identifying the user and helping in lower startup overhead. It also helps in collective communication and could be executed on both **shared memory and distributed systems**. MPJ is a familiar Java API for MPI implementation. mpiJava is the near flexible Java binding for MPJ standards.

Currently developers can produce more efficient and effective parallel applications using message passing.

A basic prerequisite for message passing is a good communication API. Java comes with various ready-made packages for communication, notably an interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. The parallel computing world is mainly concerned with

`symmetric' communication, occurring in groups of interacting peers. This symmetric model of communication is captured in the successful Message Passing Interface standard (MPI).

## Message-Passing Interface Basics:

Every MPI program must contain the preprocessor directive:

#include <mpi.h>

The mpi.h file contains the definitions and declarations necessary for compiling an MPI program.

**MPI\_Init** initializes the execution environment for MPI. It is a “share nothing” modality in which the outcome of any one of the concurrent processes can in no way be influenced by the intermediate results of any of the other processes. Command has to be called before any other MPI call is made, and it is an error to call it more than a single time within the program. **MPI\_Finalize** cleans up all the extraneous mess that was first put into place by MPI\_Init.

The principal weakness of this limited form of processing is that the processes on different nodes run entirely independent of each other. It cannot enable capability or coordinated computing. **To get the different processes to interact, the concept of communicators is needed.** MPI programs are made up of concurrent processes executing at the same time that in almost all cases

are also communicating with each other. To do this, an object called the “communicator” is provided by MPI. Thus the user may specify any number of communicators within an MPI program, each with its own set of processes. “**MPI\_COMM\_WORLD**” communicator contains all the concurrent processes making up an MPI program.

The size of a communicator is the number of processes that makes up the particular communicator. The following function call provides the value of the number of processes of the specified communicator:

int MPI\_Comm\_size(MPI\_Comm comm, int \_size).

The function "MPI\_Comm\_size” required to return the number of processes; int size. MPI\_Comm\_size(MPI\_COMM\_WORLD,&size); This will put the total number of processes in the MPI\_COMM\_WORLD communicator in the variable size of the process data context. Every process within the communicator has a unique ID referred to as its “rank”. MPI system automatically and arbitrarily assigns a unique positive integer value, starting with 0, to all the processes within the communicator. The MPI command to determine the process rank is:

int MPI\_Comm\_rank (MPI\_Comm comm, int \_rank).

The send function is used by the source process to define the data and establish the connection of the message. The send construct has the following syntax:

int MPI\_Send (void \_message, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)

The first three operands establish the data to be transferred between the source and destination processes. The first argument points to the message content itself, which may be a simple scalar or a group of data. The message data content is described by the next two arguments. The second operand specifies the number of data elements of which the message is composed. The third operand indicates the data type of the elements that make up the message.

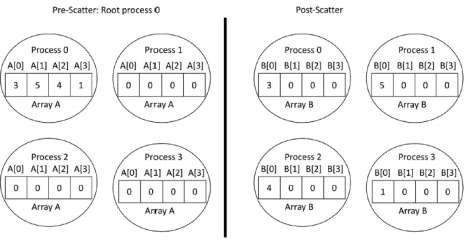
The receive command (MPI\_Recv) describes both the data to be transferred and the connection to be established. The MPI\_Recv construct is structured as follows:

int MPI\_Recv (void \_message, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status \_status)

The source field designates the rank of the process sending the message.

**Communication Collectives:** Communication collective operations can dramatically expand interprocess communication from point-to-point to n-way or all-way data exchanges.

**The scatter operation:** The scatter collective communication pattern, like broadcast, shares data of one process (the root) with all the other processes of a communicator. But in this case it partitions a set of data of the root process into subsets and sends one subset to each of the processes. Each receiving process gets a different subset, and there are as many subsets as there are processes. In this example the send array is A and the receive array is B. B is initialized to 0. The root process (process 0 here) partitions the data into subsets of length 1 and sends each subset to a separate process.



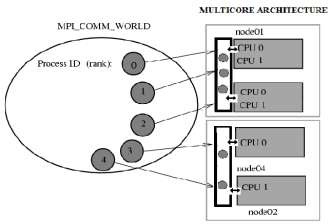
**MPJ Express** is an open source Java message passing library that allows application developers to write and execute parallel applications **for multicore processors and compute clusters / clouds.** The software is distributed under the MIT (a variant of the LGPL) license. MPJ Express is a message passing library that can be used by application developers to execute their parallel Java applications on compute clusters or network of computers.

MPJ Express is essentially a middleware that supports communication between individual processors of clusters. **The programming model followed by MPJ Express is Single Program Multiple Data (SPMD).**

The multicore configuration is meant for users who plan to write and execute parallel Java applications using MPJ Express on their desktops or laptops which contains shared memory and multicore processors. In this configuration, users can write their message passing parallel application using MPJ Express and it will be ported automatically on multicore processors. We except that users can first develop applications on their laptops and desktops using multicore configuration, and then take the same code to distributed memory platforms

## Designing the solution:

While designing the solution, we have considered the multi-core architecture as per shown in the diagram below. The communicator has processes as per input by the user. MPI program will execute the sequence as per the supplied processes and the number of processor cores available for the execution.



## Implementing the solution:

1. For implementing the MPI program in multi-core environment, we need to install MPJ express library.
   1. Download MPJ Express (mpj.jar) and unpack it.
   2. Set MPJ\_HOME and PATH environment variables:
   3. export MPJ\_HOME=/path/to/mpj/
   4. export PATH=$MPJ\_HOME/bin:$PATH
2. Write Hello World parallel Java program and save it as HelloWorld.java (Asign2.java).
3. Compile a simple Hello World (Asign) parallel Java program
4. Running MPJ Express in the Multi-core Configuration.

**Conclusion:**

There has been a large amount of interest in parallel programming using Java. mpj is an MPI binding with Java along with the support for multicore architecture so that user can develop the code on its own laptop or desktop. This is an effort to develop and run parallel programs according to MPI standard.

## ASSIGNMENT NO. 3

**Problem Statement:**

To develop any distributed application with CORBA program using JAVA IDL.

## Tools / Environment:

Java Programming Environment, JDK 1.8

## Related Theory: Common Object Request Broker Architecture (CORBA):

CORBA is an acronym for Common Object Request Broker Architecture. It is an open source, vendor-independent architecture and infrastructure developed by the **Object Management Group** (**OMG**) to integrate enterprise applications across a distributed network. CORBA specifications provide guidelines for such integration applications, based on the way they want to interact, irrespective of the technology; hence, all kinds of technologies can implement these standards using their own technical implementations.

When two applications/systems in a distributed environment interact with each other, there are quite a few unknowns between those applications/systems, including the technology they are developed in (such as Java/ PHP/ .NET), the base operating system they are running on (such as Windows/Linux), or system configuration (such as memory allocation). **They communicate mostly with the help of each other's network address or through a naming service.** Due to this, these applications end up with quite a few issues in integration, including content (message) mapping mismatches.

An application developed based on CORBA standards with standard **Internet Inter-ORB Protocol** (**IIOP**), irrespective of the vendor that develops it, should be able to smoothly integrate and operate with another application developed based on CORBA standards through the same or different vendor.

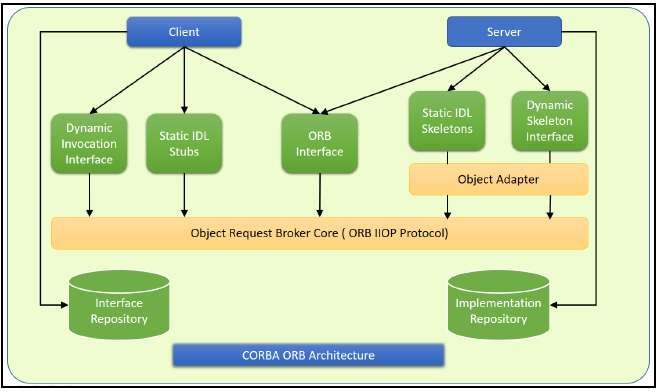
Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example. All applications related to, say, "HR&Benefits" maintain an object model with details of the organization, employees with demographic information, benefits, payroll, and deductions. They are only different in the way they handle the details, based on the country and region they are operating for. For each object type, similar to the HR&Benefits systems, we can define an interface using the **Interface Definition Language** (**OMG IDL**).

The contract between these applications is defined in terms of an interface for the server objects that the clients can call. This IDL interface is used by each client to indicate when they should call any particular method to marshal (read and send the arguments).

The target object is going to use the same interface definition when it receives the request from the client to unmarshal (read the arguments) in order to execute the method that was requested by the client operation. Again, during response handling, the interface definition is helpful to marshal (send from the server) and unmarshal (receive and read the response) arguments on the client side once received.

The IDL interface is a design concept that works with multiple programming languages including C, C++, Java, Ruby, Python, and IDLscript. This is close to writing a program to an interface, a concept we have been discussing that most recent programming languages and frameworks, such as Spring. The interface has to be defined clearly for each object. The systems encapsulate the actual implementation along with their respective data handling and processing, and only the methods are available to the rest of the world through the interface. Hence, the clients are forced to develop their invocation logic for the IDL interface exposed by the application they want to connect to with the method parameters (input and output) advised by the interface operation.

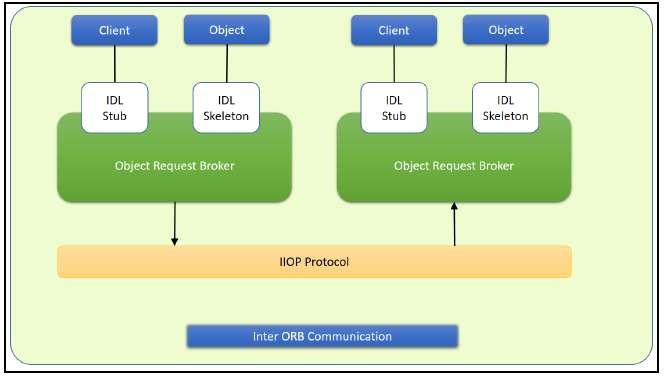
The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons, The objects are written (on the right) and a client for it (on the left), as represented in the diagram. The client and server use stubs and skeletons as proxies, respectively. The IDL interface follows a strict definition, and even though the client and server are implemented in different technologies, they should integrate smoothly with the interface definition strictly implemented.



In CORBA, each object instance acquires an object reference for itself with the electronic token identifier. Client invocations are going to use these object references that have the ability to figure out which ORB instance they are supposed to interact with. The stub and skeleton represent the client and server, respectively, to their counterparts. They help establish this communication through ORB and pass the arguments to the right method and its instance during the invocation.

## Inter-ORB communication

The following diagram shows how remote invocation works for inter-ORB communication. It shows that the clients that interacted have created **IDL Stub** and **IDL Skeleton** based on **Object Request Broker** and communicated through **IIOP Protocol**.



To invoke the remote object instance, the client can get its object reference using a naming service. Replacing the object reference with the remote object reference, the client can make the invocation of the remote method with the same syntax as the local object method invocation. ORB keeps the responsibility of recognizing the remote object reference based on the client object invocation through a naming service and routes it accordingly.

## Java Support for CORBA

CORBA complements the Java™ platform by providing a distributed object framework, services to support that framework, and interoperability with other languages. The Java platform complements CORBA by providing a portable, highly productive implementation environment, and a very robust platform. By combining the Java platform with CORBA and other key enterprise technologies, the Java Platform is the ultimate platform for distributed technology solutions.

CORBA standards provide the proven, interoperable infrastructure to the Java platform. IIOP (Internet Inter-ORB Protocol) manages the communication between the object components that power the system. The Java platform provides a portable object infrastructure that works on every major operating system. CORBA provides the network transparency, Java provides the implementation transparency. **An *Object Request Broker (ORB)* is part of the Java Platform. The ORB is a runtime component that can be used for distributed computing using IIOP communication. Java IDL is a Java API for interoperability and integration with CORBA.** Java IDL included both a Java-based ORB, which supported IIOP, and the **IDL-to-Java**

**compiler**, for generating client-side stubs and server-side code skeletons. J2SE v.1.4 includes an **Object Request Broker Daemon (ORBD), which is used to enable clients to transparently locate and invoke persistent objects on servers in the CORBA environment.**

When using the **IDL programming model,** the interface is everything! It defines the points of entry that can be called from a remote process, such as the types of arguments the called procedure will accept, or the value/output parameter of information returned. Using IDL, the programmer can make the entry points and data types that pass between communicating processes act like a standard language.

CORBA is a language-neutral system in which the argument values or return values are limited to what can be represented in the involved implementation languages. In CORBA, object orientation is limited only to objects that can be passed by reference (the object code itself cannot be passed from machine-to-machine) or are predefined in the overall framework. Passed and returned types must be those declared in the interface.

With RMI, the interface and the implementation language are described in the same language, so you don't have to worry about mapping from one to the other. Language-level objects (the code itself) can be passed from one process to the next. Values can be returned by their actual type, not the declared type. Or, you can compile the interfaces to generate IIOP stubs and skeletons which allow your objects to be accessible from other CORBA-compliant languages.

## The IDL Programming Model:

The IDL programming model, known as Java™ IDL, consists of both the Java CORBA ORB and the idlj compiler that maps the IDL to Java bindings that use the Java CORBA ORB, as well as a set of APIs, which can be explored by selecting the org.omg prefix from the Package section of the API index.

Java IDL adds CORBA (Common Object Request Broker Architecture) capability to the Java platform, providing standards-based interoperability and connectivity. Runtime components include a Java ORB for distributed computing using IIOP communication.

To use the IDL programming model, define remote interfaces using OMG Interface Definition Language (IDL), then compile the interfaces using idlj compiler. When you run the idlj compiler over your interface definition file, it generates the Java version of the interface, as well as the class code files for the stubs and skeletons that enable applications to hook into the ORB.

**Portable Object Adapter (POA) :** An *object adapter* is the mechanism that connects a request using an object reference with the proper code to service that request. The Portable Object Adapter, or POA, is a particular type of object adapter that is defined by the CORBA specification. The POA is designed to meet the following goals:

* Allow programmers to construct object implementations that are portable between different ORB products.
* Provide support for objects with persistent identities.

## Designing the solution:

Here the design of how to create a complete CORBA (Common Object Request Broker Architecture) application using IDL (Interface Definition Language) to define interfaces and Java IDL compiler to generate stubs and skeletons. You can also create CORBA application by defining the interfaces in the Java programming language.

The server-side implementation generated by the idlj compiler is the *Portable Servant Inheritance Model*, also known as the POA(Portable Object Adapter) model. This document presents a sample application created using the default behavior of the idlj compiler, which uses a POA server-side model.

## Creating CORBA Objects using Java IDL:

* 1. In order to distribute a Java object over the network using CORBA, one has to define it's own CORBA-enabled interface and it implementation. This involves doing the following:
* Writing an interface in the CORBA Interface Definition Language
* Generating a Java base interface, plus a Java stub and skeleton class, using an IDL-to-Java compiler
* Writing a server-side implementation of the Java interface in Java Interfaces in IDL are declared much like interfaces in Java.

## Modules

Modules are declared in IDL using the module keyword, followed by a name for the module and an opening brace that starts the module scope. Everything defined within the scope of this module (interfaces, constants, other modules) falls within the module and is referenced in other IDL modules using the syntax *modulename*::x. e.g.

// IDL module jen {

module corba {

interface NeatExample ...

};

};

## Interfaces

The declaration of an interface includes an interface header and an interface body. The header specifies the name of the interface and the interfaces it inherits from (if any). Here is an IDL interface header:

interface PrintServer : Server { ...

# This header starts the declaration of an interface called PrintServer that inherits all the methods and data members from the Server interface.

## Data members and methods

The interface body declares all the data members (or attributes) and methods of an interface. Data members are declared using the attribute keyword. At a minimum, the declaration includes a name and a type.

readonly attribute string myString;

The method can be declared by specifying its name, return type, and parameters, at a minimum.

string parseString(in string buffer);

This declares a method called parseString() that accepts a single string argument and returns a string value.

## A complete IDL example

Now let's tie all these basic elements together. Here's a complete IDL example that declares a module within another module, which itself contains several interfaces:

module OS {

module services { interface Server {

readonly attribute string serverName; boolean init(in string sName);

};

interface Printable {

boolean print(in string header);

};

interface PrintServer : Server { boolean printThis(in Printable p);

};

};

};

The first interface, Server, has a single read-only string attribute and an init() method that accepts a string and returns a boolean. The Printable interface has a single print()method that accepts a string header. Finally, the PrintServer interface extends the Server interface and adds a printThis() method that accepts a Printable object and returns a boolean. In all cases, we've declared the method arguments as input-only (i.e., pass-by-value), using the in keyword.

## Turning IDL Into Java

**Once the remote interfaces in IDL are described, you need to generate Java classes that act as a starting point for implementing those remote interfaces in Java using an IDL-to-Java compiler.** Every standard IDL-to-Java compiler generates the following 3 Java classes from an IDL interface:

* A Java interface with the same name as the IDL interface. This can act as the basis for a Java implementation of the interface (but you have to write it, since IDL doesn't provide any details about method implementations).
* A *helper* class whose name is the name of the IDL interface with "Helper" appended to it (e.g., ServerHelper). The primary purpose of this class is to provide a static narrow() method that can safely cast CORBA Object references to the Java interface type. The helper class also provides other useful static methods, such as read() and write() methods that allow you to read and write an object of the corresponding type using I/O streams.
* A *holder* class whose name is the name of the IDL interface with "Holder" appended to it (e.g., ServerHolder). This class is used when objects with this interface are used as out or inout arguments in remote CORBA methods. Instead of being passed directly into the remote method, the object is wrapped with its holder before being passed. When a remote method has parameters that are declared as out or inout, the method has to be able to update the argument it is passed and return the updated value. The only way to guarantee this, even for primitive Java data types, is to force out and inout arguments to be wrapped in Java holder classes, which are filled with the output value of the argument when the method returns.

The idltoj tool generate 2 other classes:

* **A client *stub* class,** called \_*interface-name*Stub, that acts as a client-side implementation of the interface and knows how to convert method requests into ORB requests that are forwarded to the actual remote object. The stub class for an interface named Server is called \_ServerStub.
* **A server *skeleton* class,** called \_*interface-name*ImplBase, that is a base class for a server- side implementation of the interface. The base class can accept requests for the object from the ORB and channel return values back through the ORB to the remote client. The skeleton class for an interface named Server is called \_ServerImplBase.

So, in addition to generating a Java mapping of the IDL interface and some helper classes for the Java interface, the *idltoj* compiler also creates subclasses that act as an interface between a CORBA client and the ORB and between the server-side implementation and the ORB.

## This creates the five Java classes: a Java version of the interface, a helper class, a holder class, a client stub, and a server skeleton.

1. **Writing the Implementation**

The IDL interface is written and generated the Java interface and support classes for it, including the client stub and the server skeleton. Now, concrete server-side implementations of all of the methods on the interface needs to be created.

## Implementing the solution:

Here, we are demonstrating the "Hello World" Example. **To create this example, create a directory named hello/ where you develop sample applications and create the files in this directory.**

## Defining the Interface (Hello.idl)

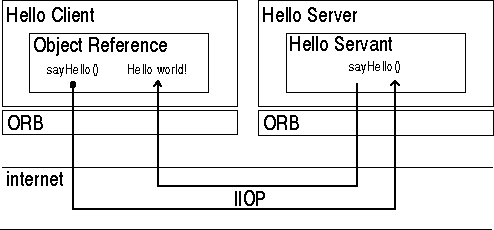
The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL).To complete the application, you simply provide the server **(HelloServer.java)** and client **(HelloClient.java)** implementations.

## Implementing the Server (HelloServer.java)

The example server consists of two classes, the servant and the server. The servant, HelloImpl, is the implementation of the Hello IDL interface; each Hello instance is implemented by a HelloImpl instance. The servant is a subclass of HelloPOA, which is generated by the idlj compiler from the example IDL. The servant contains one method for each IDL operation, in this example, the sayHello() and shutdown() methods. Servant methods are just like ordinary Java methods; the extra code to deal with the ORB, with marshaling arguments and results, and so on, is provided by the skeleton.

The HelloServer class has the server's main() method, which:

* Creates and initializes an ORB instance
* Gets a reference to the root POA and activates the POAManager
* Creates a servant instance (the implementation of one CORBA Hello object) and tells the ORB about it
* Gets a CORBA object reference for a naming context in which to register the new CORBA object
* Gets the root naming context
* Registers the new object in the naming context under the name "Hello"
* Waits for invocations of the new object from the client.



## Implementing the Client Application (HelloClient.java)

The example application client that follows:

* Creates and initializes an ORB
* Obtains a reference to the root naming context
* Looks up "Hello" in the naming context and receives a reference to that CORBA object
* Invokes the object's sayHello() and shutdown() operations and prints the result.

**Building and Executing the solution:**

The Hello World program lets you learn and experiment with all the tasks required to develop almost any CORBA program that uses static invocation, which uses a client stub for the invocation and a server skeleton for the service being invoked and is used when the interface of the object is known at compile time.

This example requires a naming service, which is a CORBA service that allows **CORBA objects** to be named by means of binding a name to an object reference. The **name binding** may be stored in the naming service, and a client may supply the name to obtain the desired object reference. The two options for Naming Services with Java include **orbd**, a daemon process containing a Bootstrap Service, a Transient Naming Service,

To run this client-server application on the development machine:

1. Change to the directory that contains the file Hello.idl.
2. Run the IDL-to-Java compiler, idlj, on the IDL file to create stubs and skeletons. This step assumes that you have included the path to the java/bin directory in your path.

**idlj -fall Hello.idl**

You must use the -fall option with the idlj compiler to generate both client and server- side bindings. This command line will generate the default server-side bindings, which assumes the POA Inheritance server-side model.

The files generated by the idlj compiler for Hello.idl, with the -fall command line option, are:

* HelloPOA.java:

This abstract class is the stream-based server skeleton, providing basic CORBA functionality for the server. It extends org.omg.PortableServer.Servant, and implements the InvokeHandler interface and the HelloOperations interface. The server class HelloImpl extends HelloPOA.

* \_HelloStub.java:

This class is the client stub, providing CORBA functionality for the client. It extends org.omg.CORBA.portable.ObjectImpl and implements the Hello.java interface.

* Hello.java:

This interface contains the Java version of IDL interface written. The Hello.java interface extends org.omg.CORBA.Object, providing standard CORBA object functionality. It also extends the HelloOperations interface and org.omg.CORBA.portable.IDLEntity.

* HelloHelper.java

This class provides auxiliary functionality, notably the narrow() method required to cast CORBA object references to their proper types. **The Helper class is responsible for reading and writing the data type to CORBA streams, and inserting and extracting the data type from AnyS.** The Holder class delegates to the methods in the Helper class for reading and writing.

* HelloHolder.java

This final class holds a public instance member of type Hello. Whenever the IDL type is an out or an inout parameter, the Holder class is used. It provides operations for org.omg.CORBA.portable.OutputStream and org.omg.CORBA.port able.InputStream arguments, which CORBA allows, but which do not map easily to Java's semantics. The Holder class delegates to the methods in the Helper class for reading and writing. It implements org.omg.CORBA.portable.Streamable.

* HelloOperations.java

This interface contains the methods sayHello() and shutdown(). The IDL-to-Java mapping puts all of the operations defined on the IDL interface into this file, which is shared by both the stubs and skeletons.

1. Compile the .java files, including the stubs and skeletons (which are in the directory directory HelloApp). This step assumes the java/bin directory is included in your path.

javac \*.java HelloApp/\*.java

1. Start orbd.

To start orbd from a UNIX command shell, enter:

orbd -ORBInitialPort 1050&

Note that 1050 is the port on which you want the name server to run. The - ORBInitialPort argument is a required command-line argument.

1. Start the HelloServer:

To start the HelloServer from a UNIX command shell, enter:

java HelloServer -ORBInitialPort 1050 -ORBInitialHost localhost&

You will see HelloServer ready and waiting... when the server is started.

1. Run the client application:

java HelloClient -ORBInitialPort 1050 -ORBInitialHost localhost

When the client is running, you will see a response such as the following on your terminal: Obtained a handle on server object: IOR: (binary code) Hello World! HelloServer exiting...

After completion kill the name server (orbd).

## Conclusion:

CORBA provides the network transparency, Java provides the implementation transparency. CORBA complements the Java™ platform by providing a distributed object framework, services to support that framework, and interoperability with other languages. The Java platform complements CORBA by providing a portable, highly productive implementation environment. The combination of Java and CORBA allows you to build more scalable and more capable applications than can be built using the JDK alone.

## ASSIGNMENT NO.4

**Problem Statement:**

To develop Token Ring distributed algorithm for leader election.

## Tools / Environment:

Java Programming Environment, JDK 1.8, Eclipse Neon(EE).

## Related Theory: Election Algorithm:

1. Many distributed algorithms require a process to act as a coordinator.
2. The coordinator can be any process that organizes actions of other processes.
3. *A* coordinator may fail.
4. How is a new coordinator chosen or elected?

## Assumptions:

Each process has a unique number to distinguish them. Processes know each other's process number.

There are two types of Distributed Algorithms:

1. Bully Algorithm
2. Ring Algorithm

## Bully Algorithm:

1. **When a process, P, notices that the coordinator is no longer responding to requests, it initiates an election.**
2. P sends an ELECTION message to all processes with higher numbers.
3. If no one responds, P wins the election and becomes a coordinator.
4. If one of the higher-ups answers, it takes over. P’s job is done.

## When a process gets an ELECTION message from one of its lower-numbered colleagues:

* 1. Receiver sends an OK message back to the sender to indicate that he is alive and will take over.
  2. Eventually, all processes give up apart of one, and that one is the new coordinator.
  3. The new coordinator announces *its* victory by sending all processes a **CO-ORDINATOR**

message telling them that it is the new coordinator.

## If a process that *was* previously down comes back:

* 1. It holds an election.
  2. If it happens to be the highest process currently running, it will win the election and take over the coordinators job.

## “Biggest guy" always wins and hence the name bully algorithm.

**Ring Algorithm:**

**Initiation:**

1. When a process notices that coordinator is not functioning:
2. Another process (initiator) initiates the election by sending "ELECTION" message (containing its own process number)

## Leader Election:

1. Initiator sends the message to it's successor (if successor is down, sender skips over it and goes to the next member along the ring, or the one after that, until a running process is located).
2. At each step, sender adds its own process number to the list in the message.
3. When the message gets back to the process that started it all: Message comes back to initiator. In the queue the **process with maximum ID Number wins**.

Initiator announces the winner by sending another message around the ring.

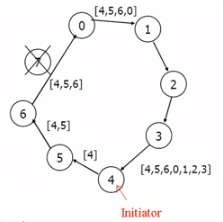
## Designing the solution:

* 1. **For Ring Algorithm Initiation:**

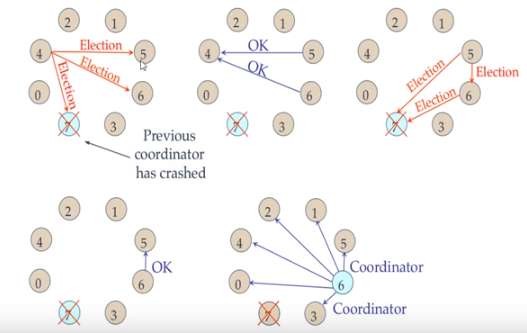
1. Consider the Process 4 understands that Process 7 is not responding.
2. Process 4 initiates the Election by sending "ELECTION" message to it's successor (or next alive process) with it's ID.

## Leader Election:

1. Messages comes back to initiator. Here the initiator is 4.
2. Initiator announces the winner by sending another message around the ring. Here the process with highest process ID is 6. The initiator will announce that Process 6 is Coordinator.



## For Bully Algorithm:



**Implementing the solution: For Ring Algorithm:**

* + 1. Creating Class for Process which includes
       1. State: Active / Inactive
       2. Index: Stores index of process.
       3. ID: Process ID
    2. Import Scanner Class for getting input from Console
    3. Getting input from User for number of Processes and store them into object of classes.
    4. Sort these objects on the basis of process id.
    5. Make the last process id as "inactive".
    6. Ask for menu 1.Election 2.Exit
    7. Ask for initializing election process.
    8. These inputs will be used by Ring Algorithm.

**Compiling and Executing the solution:**

1. Create Java Project in Eclipse
2. Create Package
3. Add class in package Ring.java.
4. Compile and Execute in Eclipse.

The output is associated in the above section.

**Conclusion:** Election algorithms **are designed to choose a coordinator.** We have two election algorithms for two different configurations of distributed system. **The Bully** algorithm applies to system where every process can send a message to every other process in the system and **The Ring** algorithm applies to systems organized as a ring (logically or physically). In this algorithm we assume that the link between the process are unidirectional and every process can message to the process on its right only.

## ASSIGNMENT NO. 5

**Problem Statement:**

To create a simple web service and write any distributed application to consume the web service.

## Tools / Environment:

Java Programming Environment, JDK 8, Netbeans IDE with GlassFish Server

## Related Theory: Web Service:

A web service can be defined as a collection of open protocols and standards for exchanging information among systems or applications.

A service can be treated as a web service if:

* + The service is discoverable through a simple lookup
  + It uses a standard XML format for messaging
  + It is available across internet/intranet networks.
  + It is a self-describing service through a simple XML syntax
  + The service is open to, and not tied to, any operating system/programming language

## Types of Web Services:

There are two types of web services:

1. **SOAP**: SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services. Since it’s XML based, it’s platform and language independent. So, our server can be based on JAVA and client can be on .NET, PHP etc. and vice versa.
2. **REST**: REST (Representational State Transfer ) is an architectural style for developing web services. It’s getting popularity recently because it has small learning curve when compared to SOAP. Resources are core concepts of Restful web services and they are uniquely identified by their URIs.

## Web service architectures:

As part of a web service architecture, there exist three major roles.

**Service Provider** is the program that implements the service agreed for the web service and exposes the service over the internet/intranet for other applications to interact with.

**Service Requestor** is the program that interacts with the web service exposed by the Service Provider. It makes an invocation to the web service over the network to the Service Provider and exchanges information.

**Service Registry** acts as the directory to store references to the web services.

The following are the steps involved in a basic SOAP web service operational behavior:

1. The client program that wants to interact with another application prepares its request content as a SOAP message.
2. Then, the client program sends this SOAP message to the server web service as an HTTP POST request with the content passed as the body of the request.
3. The web service plays a crucial role in this step by understanding the SOAP request and converting it into a set of instructions that the server program can understand.
4. The server program processes the request content as programmed and prepares the output as the response to the SOAP request.
5. Then, the web service takes this response content as a SOAP message and reverts to the SOAP HTTP request invoked by the client program with this response.
6. The client program web service reads the SOAP response message to receive the outcome of the server program for the request content it sent as a request.

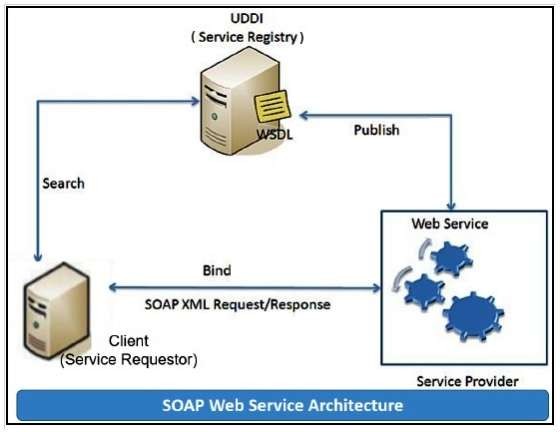
## SOAP web services:

**Simple Object Access Protocol** (**SOAP**) is an XML-based protocol for accessing web services. It is a W3C recommendation for communication between two applications, and it is a platform- and language-independent technology in integrated distributed applications.

While XML and HTTP together make the basic platform for web services, the following are the key components of standard SOAP web services:

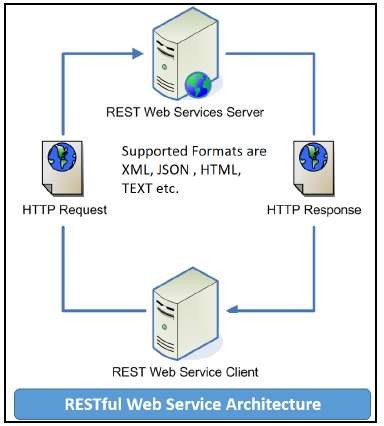
**Universal Description, Discovery, and Integration** (**UDDI**)*:* UDDI is an XMLbased framework for describing, discovering, and integrating web services. It acts as a directory of web service interfaces described in the WSDL language.

**Web Services Description Language** (**WSDL**)*:* WSDL is an XML document containing information about web services, such as the method name, method parameters, and how to invoke the service. WSDL is part of the UDDI registry. It acts as an interface between applications that want to interact based on web services. The following diagram shows the interaction between the UDDI, Service Provider, and service consumer in SOAP web services:

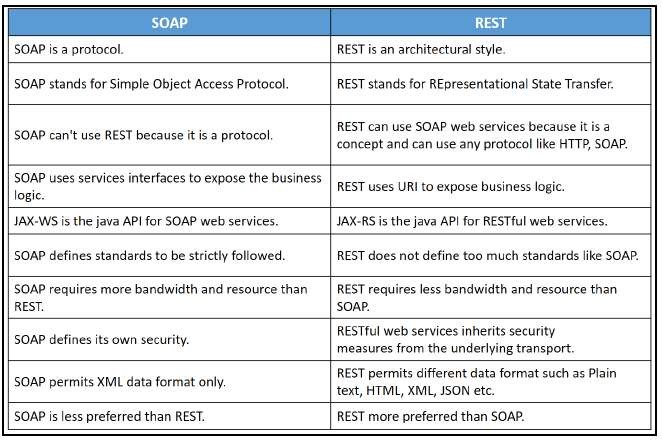


## RESTful web services

**REST** stands for **Representational State Transfer**. RESTful web services are considered a performance-efficient alternative to the SOAP web services. REST is an architectural style, not a protocol. Refer to the following diagram:



While both SOAP and RESTful support efficient web service development, the difference between these two technologies can be checked out in the following table :

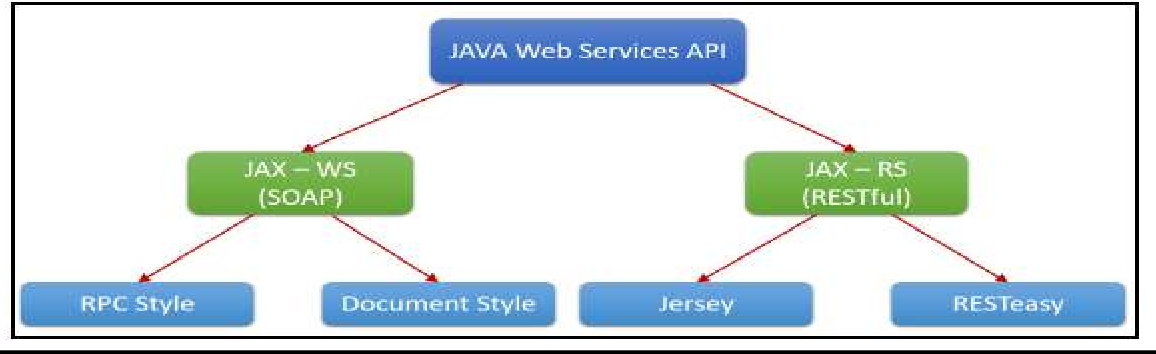


## Designing the solution:

Java provides it’s own API to create both SOAP as well as RESTful web services.

1. **JAX-WS**: JAX-WS stands for Java API for XML Web Services. JAX-WS is XML based Java API to build web services server and client application.
2. **JAX-RS**: Java API for RESTful Web Services (JAX-RS) is the Java API for creating REST web services. JAX-RS uses annotations to simplify the development and deployment of web services.

Both of these APIs are part of standard JDK installation, so we don’t need to add any jars to work with them.



**Students are required to implement both i.e. using SOAP and RESTful APIs.**

**Implementing the solution:**

1. **Creating a web service CalculatorWSApplication:**
   * Create New Project for CalculatorWSApplication.
   * Create a package org.calculator
   * Create class CalculatorWS.
   * Right-click on the CalculatorWS and create New Web Service.
   * IDE starts the glassfish server, builds the application and deploys the application on server.

## Consuming the Webservice:

* + Create a project with an CalculatorClient
  + Create package org.calculator.client;
  + add java class CalculatorWS.java, addresponse.java, add.java, CalculatorWSService.java and ObjectFactory.java

## Creating servlet in web application

* + Create new jsp page for creating user interface.

**Compiling and Executing the solution:**

Right Click on the Project and Choose Run.

## Conclusion:

This assignment, described the Web services approach to the Service Oriented Architecture concept. Also, described the Java APIs for programming Web services and demonstrated examples of their use by providing detailed step-by-step examples of how to program Web services in Java.

## ASSIGNMENT NO. 6

**Problem Statement:**

To develop any distributed application using Messaging System in Publisher-Subscriber paradigm.

## Tools / Environment:

Java Programming Environment, JDK 8, Eclipse IDE, Apache ActiveMQ 4.1.1, JMS

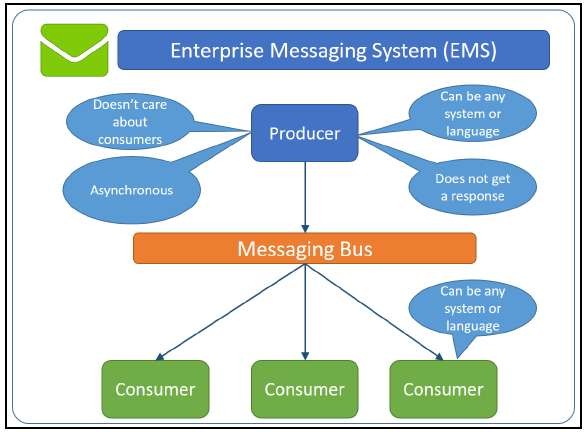
## Related Theory:

Large distributed systems are often overwhelmed with complications caused by heterogeneity and interoperability. Heterogeneity issues may arise due to the use of different programming languages, hardware platforms, operating systems, and data representations. Interoperability denotes the ability of heterogeneous systems to communicate meaningfully and exchange data or services. With the introduction of middleware, heterogeneity can be alleviated and interoperability can be achieved.

Middleware is a layer of software between the distributed application and the operating system and consists of a set of standard interfaces that help the application use networked resources and services.

## Enterprise Messaging System:

EMS, or the messaging system, defines system standards for organizations so they can define their enterprise application messaging process with a semantically precise messaging structure. EMS encourages you to define a loosely coupled application architecture in order to define an industry-accepted message structure; this is to ensure that published messages would be persistently consumed by subscribers. Common formats, such as XML or JSON, are used to do this. EMS recommends these messaging protocols: DDS, MSMQ, AMQP, or SOAP web services. Systems designed with EMS are termed **Message-Oriented Middleware (MOM)**. An asynchronous communication is used while messaging in EMS.



## Java Messaging Service

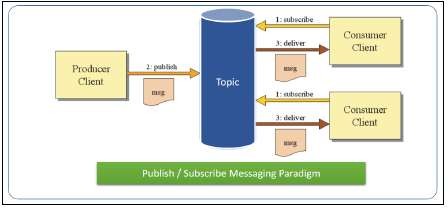
**Java's implementation of an EMS in the Application Programming Interface (API) format is known as JMS.**

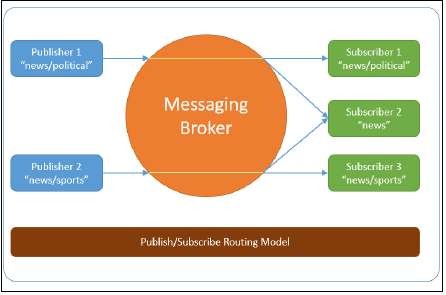
JMS allows distributed Java applications to communicate with applications developed in any other technology that understands messaging through asynchronous messages. JMS applications contain a provider, clients, messages, and administrated objects.

JMS providing a standard, portable way for Java programs to send/receive messages through a MOM product. Any application written in JMS can be executed on any MOM that implements the JMS API standards. The JMS API is specified as a set of interfaces as part of the Java API. Hence, all the products that intend to provide JMS behavior will have to deliver the provider to implement JMS-defined interfaces. With programming patterns that allow a program to interface, you should be able to construct a Java application in line with the JMS standards by defining the messaging programs with client applications to exchange information through JMS messaging.

## The publish/subscribe messaging paradigm:

The publish/subscribe messaging paradigm is built with the concept of a topic, which behaves like an announcement board. Consumers subscribe to receiving messages that belong to a topic, and publishers report messages to a topic. The JMS provider retains the responsibility for distributing the messages that it receives from multiple publishers to many other subscribers based on the topic they subscribe to. A subscriber receives messages that it subscribes to based on the rules it defines and the messages that are published after the subscription is registered; they do not receive any messages that are already published, as shown in the following diagram:





## JMS interfaces

JMS defines a set of high-level interfaces that encapsulate several messaging concepts. These high-level interfaces are further extended for the Point-To-Point and publish/subscribe messaging domains:

ConnectionFactory: This is an administered object with the ability to create a connection.

Connection: This is an active connection handle to the provider.

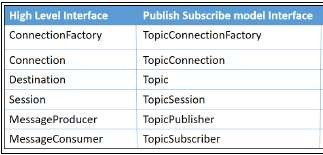
Destination: This is an administered object that encapsulates the identity of a message destination where messages are sent to/received from.

Session: This is a single-threaded context for sending/receiving messages. To ensure a simple session-based transaction, concurrent access to a message by multiple threads is restricted. We can use multiple sessions for a multithreaded application.

MessageProducer: This is used to send messages.

MessageConsumer: This is used to receive messages.

The following table shows interfaces specific to publish/subscribe paradigms enhanced from their corresponding high-level interface:

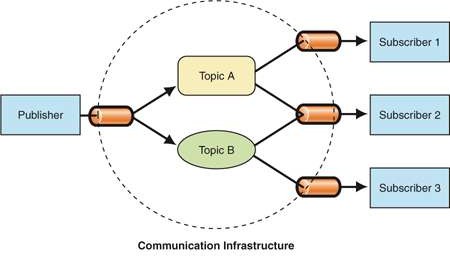


## Designing the solution:

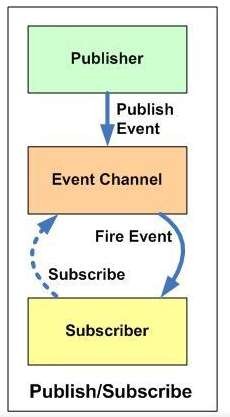
In ‘Publisher-Subscriber’ pattern, senders of messages, called **publishers**, do not program the messages to be sent directly to specific receivers, called **subscribers.**

For example, consider there is a publisher publishes news (topics) related to politics and sports; they publish to the Messaging Broker, as shown in the following diagram. While Subscriber 1 receives news related to politics and Subscriber 3 receives news related to sports, Subscriber 2 will receive both political and sports news as it subscribed to the common topics.

In designing our solution, we have created one publisher and subscriber wherein the publisher creates topic.



The **Publisher/Subscriber** pattern is mostly implemented in an ***asynchronous*** way (using message queue).

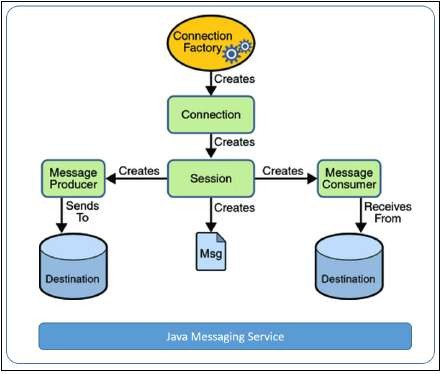


Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

JMS is a Java API that allows applications to create, send, receive, and read messages. The JMS API enables communication that is loosely coupled, asynchronous and reliable.

To use JMS, we need to have a JMS provider that can manage the sessions, queues, and topics. Some examples of known JMS providers are Apache ActiveMQ, WebSphere MQ from IBM or SonicMQ from Aurea Software. Starting from Java EE version 1.4, a JMS provider has to be contained in all Java EE application servers.

Refer to the following diagram:



A JMS provider is a messaging server that supports the creation of connections (multithreaded virtual links to the provider) and sessions (single-threaded contexts for producing and consuming messages). A JMS client is a Java program that either produces or consumes messages.

JMS messages are objects that communicate information between JMS clients and are composed of a header, some optional properties, and an optional body.

**Administered objects** are preconfigured JMS objects, such as a connection factory (the object a client uses to create a connection to a provider) and a destination (the object a client uses to specify a target for its messages).

JMS applications are usually developed in either the publish/subscribe or Point-To-Point paradigm.

The following are the objectives of JMS, as highlighted in its specification:

* Defining a common collection of messaging concepts and features
* Minimizing the number of concepts a developer should learn to develop
* applications as EMS's
* Improving the application messaging portability
* Reducing the effort involved in implementing a provider
* Providing client interfaces for both Point-To-Point and pub/sub domains

## Implementing the solution:

1. To execute the pub-sub programs, you need the message queue environment.

The Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. It is a Java API that allows applications to create, send, receive, and read messages. The JMS API enables communication that is loosely coupled, asynchronous and reliable.

To use JMS, we need to have a JMS provider that can manage the sessions, queues, and topics. Some examples of known JMS providers are Apache ActiveMQ, WebSphere MQ from IBM or SonicMQ from Aurea Software. Starting from Java EE version 1.4, a JMS provider has to be contained in all Java EE application servers.

Here we are implementing the JMS concepts and illustrates them with a JMS Hello World example using ActiveMQ.

Interfaces extending core JMS interfaces for Topic help build publish-subscribe components.

1. The Publisher.java program to publish messages to the Publish-Subscribe topic. The code for which is shown in the below section.
2. While the preceding program helps publish messages to the Publish-Subscribe Topic, the Subscribe.java program is used to subscribe to the Publish-Subscribe Topic, which keeps receiving messages related to the Topic until the quit command is given.

**Compilation and Executing the solution:**

**For Setting up an environment:**

1. Download the 2 Jar files javax.jms.jar for JMS and apache-activemq-4.1.1.jar for Apache ActiveMQ.
2. Download Apache MQ and Install it using the Apache MQ Installation Link

Links for Download and installation instruction:

* 1. Jms <http://www.java2s.com/Code/Jar/j/Downloadjavaxjmsjar.htm>……… [Jar file]
  2. Apache <http://www.java2s.com/Code/Jar/a/Downloadapacheactivemq411jar.htm>[Jar file]
  3. Download - <http://activemq.apache.org/activemq-5158-release.html> [ApacheMQ Download link]
  4. Install - https://docs.wso2.com/display/BAM200/Installing+Apache+ActiveMQ+on+Linux

…………. [Apache MQ Installation Instructions]

* 1. Concept - https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c

## Steps to execute:

* + 1. Create a Publisher.java file and copy paste the Publisher code
    2. Create a Subscriber.java file and copy paste the Subscriber code

1. Add external jars
   1. Right Click on Project in eclipse package explorer
   2. Go to Build Path
   3. Select Configure Build Path
   4. Add external jars
   5. Select both the downloaded jars from the first step
2. Run activemq with the following command:

## sudo sh active start

1. Run the publisher code and pin console for publisher
2. Run Subscriber

**Conclusion:** This assignment includes study of Publish-Subscribe model of Communication which is implemented using JMS and Apache ActiveMQ. The topic based filtering requires the messages to be broadcasted into logical channels, the subscribers only receives messages from logic channels they are subscribed.

## ASSIGNMENT NO. 7

**Problem Statement:**

To develop micro services framework based distributed application.

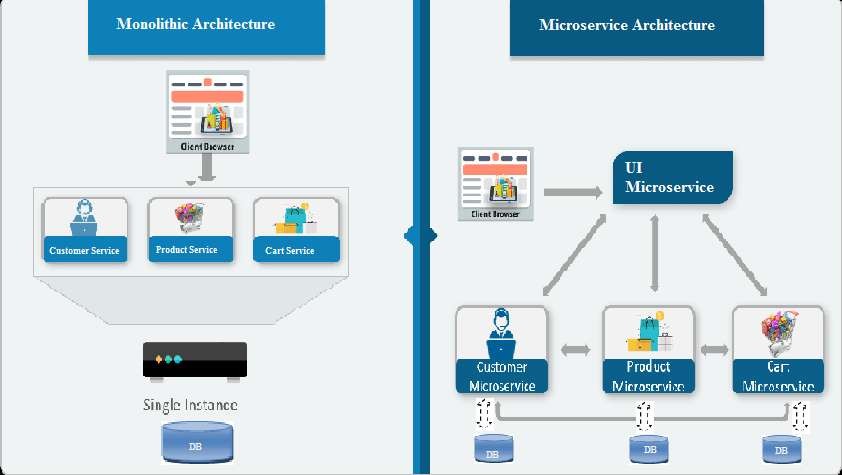
## Tools / Environment:

Python 3.6.0 using Flask framework.

## Related Theory:

1. **Microservices:**

Traditional application design is often called “monolithic” because the whole thing is developed in one piece. Even if the logic of the application is modular it’s deployed as one group, like a Java application as a JAR file for example. This monolith eventually becomes so difficult to manage as the larger applications require longer and longer deployment timeframes. In contrast with the monolith type application, here’s what an app developed with a microservices focus might look like:



A team designing micro services architecture for their application will split all of the major functions of an application into independent services. Each independent service is usually packaged as an API so it can interact with the rest of the application elements.

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are:

* + Highly maintainable and testable
  + Loosely coupled
  + Independently deployable
  + Organized around business capabilities.

The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack.

1. **Web frameworks** encapsulate what developers have learned over the past twenty years while programming sites and applications for the web. Frameworks make it easier to reuse code for common HTTP operations and to structure projects so other developers with knowledge of the framework can quickly build and maintain the application.

**Common web framework functionality:** Frameworks provide functionality in their code or through extensions to perform common operations required to run web applications. These common operations include:

1. URL routing
2. Input form handling and validation
3. [HTML,](https://www.fullstackpython.com/hypertext-markup-language-html.html) XML, JSON, and other output formats with a [templating engine](https://www.fullstackpython.com/template-engines.html)
4. Database connection configuration and persistent data manipulation through an [object- relational mapper (ORM)](https://www.fullstackpython.com/object-relational-mappers-orms.html)
5. [Web security](https://www.fullstackpython.com/web-application-security.html) against Cross-site request forgery (CSRF), SQL Injection, Cross-site Scripting (XSS) and other common malicious attacks
6. Session storage and retrieval.
7. **Flask** (source code) is a Python web framework built with a small core and easy-to-extend philosophy. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine.
8. **WSGI:** Web Server Gateway Interface (WSGI) has been adopted as a standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications.
9. **Werkzeug :**It is a WSGI toolkit, which implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.
10. **Virtual Environment:**

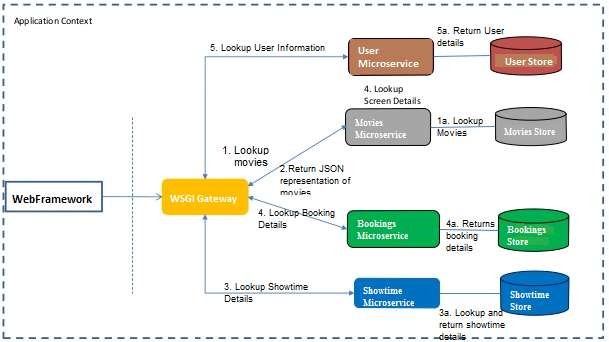
In Python, by default, every project on the system will use the same directories to store and

retrieve **site packages** (third party libraries). and **system packages** (packages that are part of the standard Python library). Consider the a scenario where there are two projects: *ProjectA* and *ProjectB*, both of which have a dependency on the same library, *ProjectC*. The problem becomes apparent when we start requiring different versions of *ProjectC*. Maybe *ProjectA* needs v1.0.0, while *ProjectB* requires the newer v2.0.0, for example.

Since projects are stored in site-packages directory according to just their name and can't differentiate between versions, both projects, *ProjectA* and *ProjectB*, would be required to use the same version which is unacceptable in many cases and hence the virtual environment. The

main purpose of Python virtual environments is to create an isolated environment for Python projects. This means that each project can have its own dependencies, regardless of what dependencies every other project has. There are no limits to the number of environments you can have since they’re just directories containing a few scripts. Plus, they’re easily created using the virtualenv or pyenv command line tools.

## Designing the solution:



Here, we are attempting to develop an microservice based architecture for Movie ticket Booking web application. The services are being implemented using python and JSON is used as for Data Store.

## Implementing the solution:

1. **Using Virtual Environments:** Install virtualenv for development environment. virtualenv is a virtual Python environment builder. It helps a user to create multiple Python environments side-by-side. Thereby, it can avoid compatibility issues between the different versions of the libraries.

The following command installs virtualenv: Sudo apt-get install virtualenv

## Flask Module:

Importing flask module in the project is mandatory. An object of Flask class is our WSGI application. Flask constructor takes the name of current module ( name ) as argument. The route() function of the Flask class is a decorator, which tells the application which URL should call the associated function.

## Route decorator:

The route() decorator in Flask is used to bind URL to a function. For example −

@app.route(‘/hello’) def hello\_world():

return ‘hello world’

Here, URL ‘/hello’ rule is bound to the hello\_world() function. As a result, if a user visits http://localhost:5000/hello URL, the output of the hello\_world() function will be rendered in the browser.

1. **Writing the subroutine for the four microservices:** There are four microservices viz., user, Showtimes, Bookings and Movies for which microservices are to be implemented.

**Conclusion:** With microservices, modules within software can be independently deployable. In a microservices architecture, each service runs a unique process and usually manages its own database. This not only provides development teams with a more decentralized approach to

building software, it also allows each service to be deployed, rebuilt, redeployed and managed independently. Netflix, eBay, Amazon, the UK Government Digital Service, Twitter, PayPal, The Guardian, and many other large-scale websites and applications have all evolved from

monolithic to microservices architecture.