Unit 4

Distributed Network Programming 8hrs

4.1 TCP, UDP, IP Address and Ports

Introduction to TCP

Transmission Control Protocol (TCP) is a communications standard that enables application programs and computing devices to exchange messages over a network. It is designed to send packets across the internet and ensure the successful delivery of data and messages over networks.

TCP is one of the basic standards that define the rules of the internet and is included within the standards defined by the Internet Engineering Task Force (IETF). It is one of the most commonly used protocols within digital network communications and ensures end-to-end data delivery.

TCP organizes data so that it can be transmitted between a server and a client. It guarantees the integrity of the data being communicated over a network. Before it transmits data, TCP establishes a connection between a source and its destination, which it ensures remains live until communication begins. It then breaks large amounts of data into smaller packets, while ensuring data integrity is in place throughout the process.

Introduction to UDP

User Datagram Protocol (UDP) is a communications protocol for time-sensitive applications like gaming, playing videos, or Domain Name System (DNS) lookups. UDP results in speedier communication because it does not spend time forming a firm connection with the destination before transferring the data. Because establishing the connection takes time, eliminating this step results in faster data transfer speeds.

However, UDP can also cause data packets to get lost as they go from the source to the destination. It can also make it relatively easy for a hacker to execute a distributed denial-of-service (DDoS) attack.

In many cases, particularly with Transmission Control Protocol (TCP), when data is transferred across the internet, it not only has to be sent from the destination but also the receiving end has to signal that it is ready for the data to arrive. Once both of these aspects of the communication are fulfilled, the

transmission can begin. However, with UDP, the data is sent before a connection has been firmly established. This can result in problems with the data transfer, and it also presents an opportunity for hackers who seek to execute DDoS attacks.

IP Address

An Internet Protocol (IP) address is the unique identifying number assigned to every device connected to the internet. An IP address definition is a numeric label assigned to devices that use the internet to communicate. Computers that communicate over the internet or via local networks share information to a specific location using IP addresses.

IP addresses have two distinct versions or standards. The Internet Protocol version 4 (IPv4) address is the older of the two, which has space for up to 4 billion IP addresses and is assigned to all computers. The more recent Internet Protocol version 6 (IPv6) has space for trillions of IP addresses, which accounts for the new breed of devices in addition to computers. There are also several types of IP addresses, including public, private, static, and dynamic IP addresses.

There are five classes of IP address:

Name	First octet	Description
Class A	1 to 126	Many hosts per network.
Class B	128 to 191	Many hosts per network.
Class C	192 to 223	Many networks with fewer hosts per network.
Class D	224 to 239	Multicasting.

Understanding Port:

Clients connect to servers via objects known as ports. A port serves as a channel through which several clients can exchange data with the same server or with different servers. Ports are usually specified by numbers. Some ports are dedicated to special servers or tasks. For example, many computers reserve port number 13 for the day/time server, which allows clients to obtain the date and time. Port number 80 is reserved for a Web server, and so forth. Most computers also have hundreds or even thousands of free ports available for use by network applications.

java.net- Networking Classes and Interfaces

Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model.

The classes contained in the java.net package are as follows:

Authenticator	Inet6Address	ServerSocket
CacheRequest	InetAddress	Socket
CacheResponse	InetSocketAddress	SocketAddress
ContentHandler	InterfaceAddress (Added by Java SE 6.)	SocketImpl
CookieHandler	JarURLConnection	SocketPermission
CookieManager (Added by Java SE 6.)	MulticastSocket	URI
DatagramPacket	NetPermission	URL
DatagramSocket	NetworkInterface	URLClassLoader
DatagramSocketImpl	PasswordAuthentication	URLConnection
HttpCookie (Added by Java SE 6.)	Proxy	URLDecoder
HttpURLConnection	ProxySelector	URLEncoder
IDN (Added by Java SE 6.)	ResponseCache	URLStreamHandler
Inet4Address	SecureCacheResponse	

The **java.net** package's interfaces are as follows:

ContentHandlerFactory	DatagramSocketImplFactory	SocketOptions
CookiePolicy (Added by Java SE 6.)	FileNameMap	URLStreamHandlerFactory
CookieStore (Added by Java SE 6)	SocketImplFactory	

Example to display the IP address of current host:

```
import java.net.InetAddress;
public class HostInfo{
     public static void main(String args[]){
          try{
               InetAddress ipaddress=InetAddress.getLocalHost();
               System.out.println("IP address:\n"+ipaddress);
          }catch(Exception e){
               System.out.println("Unknown Host");
          }
```

```
	}
}
```

Note: We can also get the IP address of any machine by name as follows:

```
import java.net.InetAddress;
import java.util.Scanner;
public class HostInfo{
	public static void main(String args[]){
		try{
			Scanner sc=new Scanner(System.in);
			String name=sc.next();
			InetAddress ipaddress=InetAddress.getByName(name);
			System.out.println("IP address:\n"+ipaddress);
			sc.close();
		}catch(Exception e){
			System.out.println("Unknown Host");
		}
	}
}
```

4.2 Socket Programming using TCP and UDP

Understanding Socket

Socket is an object which establishes a communication link between two ports. A socket is an endpoint of a two-way communication link between two programs running on the network. Socket is bound to a port number so that the TCP layer can identify the application that Data is destined to be sent. The working mechanism of the socket is as shown in the figure below:

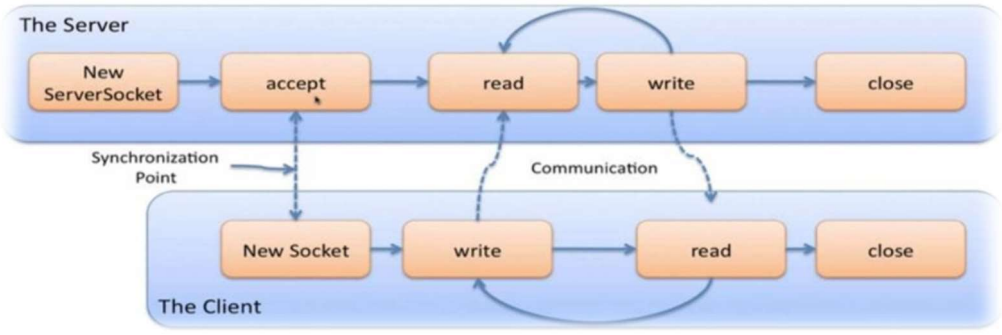
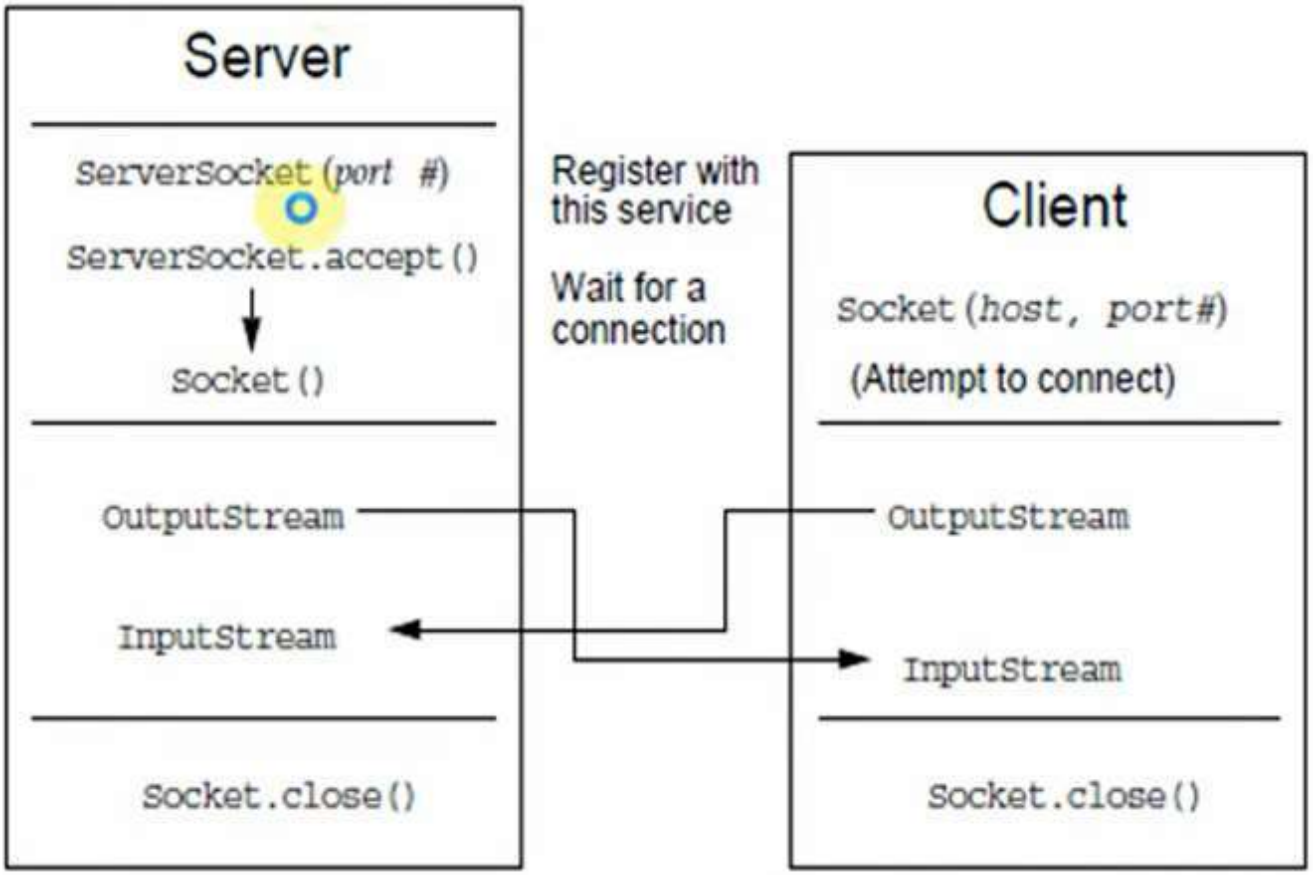


Fig: overview of java socket



Working mechanism of the socket is as shown in the table below by the example of the day/time server:

SN	Server Program	Client Program
1	Informs the user that it has started	
2	Sets up a server to listen for client on port; certain port says 1234	
3	Waits for a client to make a connection	
4		Asks the user for the IP address of the available server for example day/time server
5		After the user inputs the day/time server, connections on port 1234
6		Establishes the input stream on the Connection
7	Establishes the output stream on the connection.	
8	Writes the day/time to the stream.	
9		Reads the day/time from the stream
10	Closes the connection	Closes the connection

Implementing TCP/IP based Server and Client

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based

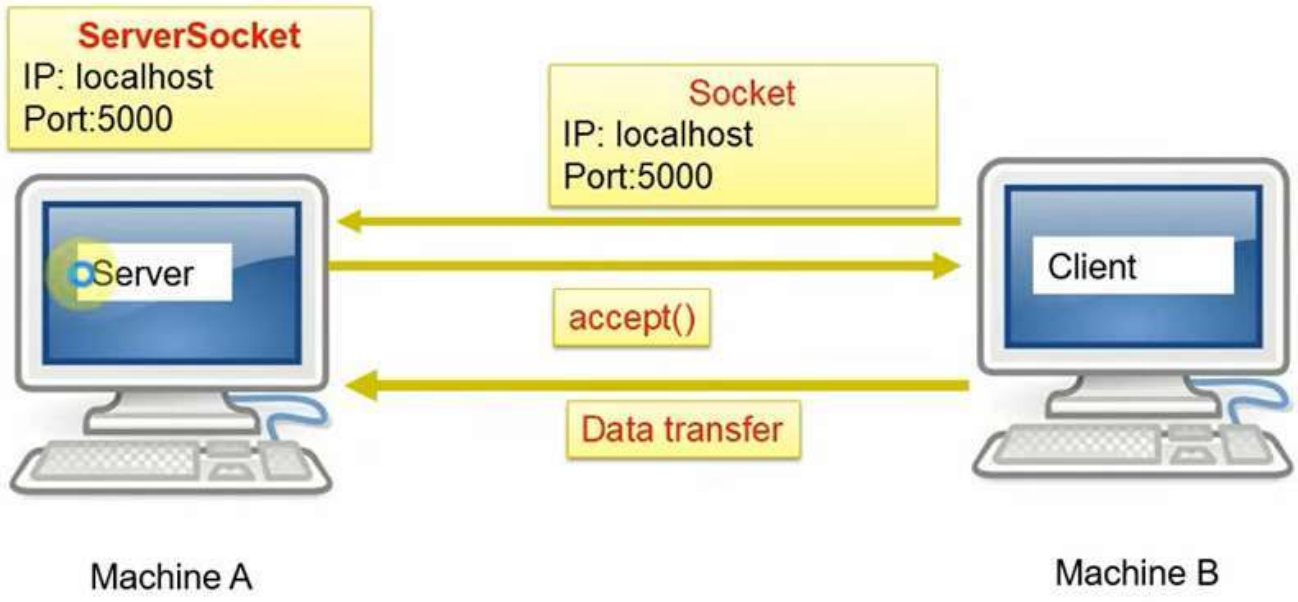
connections between hosts on the Internet. There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients.

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

1. The server initializes a ServerSocket object, denoting in which port number communication is to start.
2. The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
3. After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.
4. The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.
5. 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, so data can be sent across both streams at the same time. There are following useful classes providing complete set of methods to implement sockets.

Write two-way TCP/IP client Server program where client and server can communicate each other



TCP/IP Server Socket

The TCP/IP server socket is used to communicate with the client machine. The steps to get a server up and running are shown below:

1. Create a server socket and name the socket

2. Wait for a request to connect, a new client socket is created here
3. Read/Write data sent from/to client and send data back to client
4. Close client socket
5. Loop back if not told to exit
6. Close server socket

Server.java

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

public class Server {
public static void main(String[] args) {
    try {
        ServerSocket serverSocket = new ServerSocket(1234);
        System.out.println("server ready...");
        while (true) {
            Socket s = serverSocket.accept();
            System.out.println("client connected");

            //io stream
            DataInputStream din = new DataInputStream(s.getInputStream());
            DataOutputStream dos = new DataOutputStream(s.getOutputStream());

            Scanner in = new Scanner(System.in);
            System.out.println("write message");
            while (true) {
                //reading
                System.out.print("Client: " + din.readUTF() + "\n");

                //writing
                dos.writeUTF(in.nextLine());
            }
        }

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

TCP/IP Client socket

The TCP/IP client socket is used to communicate with the server. Following are the steps client needs to take in order to communicate with the server.

1. Create a socket with the server IP address
2. Connect to the server, this step also names the socket
3. Send data to the server
4. Read data returned (echoed) back from the server
5. Close the socket

Client.java

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

public class Client {
public static void main(String[] args) {
    try {
        Socket s = new Socket("localhost", 1234);
        System.out.println("client ready");

        //io stream
        DataOutputStream dos = new DataOutputStream(s.getOutputStream());
        DataInputStream din = new DataInputStream(s.getInputStream());

        Scanner in = new Scanner(System.in);
        System.out.println("write message");
        while (true) {
            //writing
            dos.writeUTF(in.nextLine());
            dos.flush();

            //reading
            System.out.println("Server: " + din.readUTF() + "\n");
        }
    } catch (Exception e) {
        System.out.println("client error: " + e);
    }
}
}
```

Datagrams- Datagram Packet, Datagram Server and Client

DatagramPacket class represents a datagram packet. Datagram packets are used to implement a connectionless packet delivery service. Each message is routed from one machine to another, based solely on information contained within that packet. Multiple packets sent from one machine to another might be routed differently and might arrive in any order. Packet delivery is not guaranteed.

Datagrams are bundles of information passed between machines. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.
Java implements datagrams on top of the UDP protocol by using two classes: the **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the DatagramPackets.

DatagramSocket:

It defines four public constructors. They are as shown below:

Constructor	Meaning
DatagramSocket( ) throws SocketException	It creates a DatagramSocket bound to any unused port on the local computer.
DatagramSocket(int port) throws SocketException	It creates a DatagramSocket bound to the port specified by port.
DatagramSocket(int port, InetAddress ipAddress) throws SocketException	It constructs a DatagramSocket bound to the specified port and InetAddress.
DatagramSocket(SocketAddress address) throws SocketException	It constructs a DatagramSocket bound to the specified SocketAddress.

SocketAddress is an abstract class that is implemented by the concrete class InetSocketAddress. InetSocketAddress encapsulates an IP address with a port number. All can throw a SocketException if an error occurs while creating the socket.

DatagramSocket defines many methods. Two of the most important are send( ) and receive( ), which are shown here:

- void send(DatagramPacket packet) throws IOException
- void receive(DatagramPacket packet) throws IOException

The send( ) method sends packet to the port specified by packet. The receive method waits for a packet to be received from the port specified by packet and returns the result. Other methods give us access to various attributes associated with a DatagramSocket. Some of the methods are:

Methods	Meaning
InetAddress getInetAddress( )	If the socket is connected, then the address is returned. Otherwise, null is returned.
int getLocalPort( )	Returns the number of the local port.
int getPort( )	Returns the number of the port to which the socket is connected. It returns –1 if the socket is not connected to a port.
boolean isBound( )	Returns true if the socket is bound to an address. Returns false otherwise.
boolean isConnected( )	Returns true if the socket is connected to a server. Returns false otherwise.
void setSoTimeout(int millis) throws SocketException	Sets the time-out period to the number of milliseconds passed in millis.

DatagramPacket

DatagramPacket defines several constructors. Four are shown here:

Constructor	Meaning
DatagramPacket(byte data[ ], int size)	It specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a DatagramSocket.

DatagramPacket(byte data[ ], int offset, int size)	It allows us to specify an offset into the buffer at which data will be stored.
DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port)	It specifies a target address and port, which are used by a DatagramSocket to determine where the data in the packet will be sent.
DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress, int port)	It transmits packets beginning at the specified offset into the data.

DatagramPacket defines several methods. In general, the get methods are used on packets that are received and the set methods are used on packets that will be sent.

Methods	**Meaning**
InetAddress getAddress( )	Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
byte[ ] getData( )	Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
int getLength( )	Returns the length of the valid data contained in the byte array that would be returned from the getData( ) method. This may not equal the length of the whole byte array.
int getOffset( )	Returns the starting index of the data.
int getPort( )	Returns the port number.
void setAddress(InetAddress ipAddress)	Sets the address to which a packet will be sent. The address is specified by ipAddress.
void setData(byte[ ] data)	Sets the data to data, the offset to zero, and the length to number of bytes in data.
void setData(byte[ ] data, int idx, int size)	Sets the data to data, the offset to idx, and the length to size.
void setLength(int size)	Sets the length of the packet to size.
void setPort(int port)	Sets the port to port.

A Datagram Example

The following example implements a very simple networked communications client and server. Messages are typed into the window at the client and written across the network to the server, where capitalize the message and return back to the client and are displayed in client machine.

UDPServer.java

```
import java.net.*;

public class UDPServer {
public static void main(String args[]) throws Exception {

DatagramSocket ds = new DatagramSocket(9999);
System.out.println("server ready at port: 9999");
while (true) {
    byte[] dReceive = new byte[1024];
```

```
    DatagramPacket dpReceive = new DatagramPacket(dReceive, dReceive.length);
    ds.receive(dpReceive);
    String message = new String(dpReceive.getData(), 0, dpReceive.getLength());
    System.out.println("Client sent: " + message);

    byte[] dSend = message.toUpperCase().getBytes();
    InetAddress ia = InetAddress.getLocalHost();
    DatagramPacket dpSend = new DatagramPacket(dSend, dSend.length, ia,
dpReceive.getPort());
    ds.send(dpSend);
}
}
}
```

UDPClient.java

```
import java.io.*;
import java.net.*;
import java.util.Scanner;

public class UDPClient {
public static void main(String args[]) throws Exception {
DatagramSocket ds = new DatagramSocket();

Scanner in = new Scanner(System.in);
System.out.println("enter message");
while (true) {
    byte[] dSend = in.nextLine().getBytes();
    InetAddress ia = InetAddress.getLocalHost();
    DatagramPacket dpSend = new DatagramPacket(dSend, dSend.length, ia, 9999);
    ds.send(dpSend);

    byte[] dReceive = new byte[1024];
    DatagramPacket dpReceive = new DatagramPacket(dReceive, dReceive.length);
    ds.receive(dpReceive);

    String message = new String(dpReceive.getData(), 0, dpReceive.getLength());
    System.out.println("Server Sent: " + message);
}
}
}
```

4.3 Working with URLs and URL Connection Class

Internet Addressing URL

The URL allows uniquely identifying or addressing information on the Internet. Every browser uses them to identify information on the Web. Within Java's network class library, the URL class provides a simple, concise API to access information across the Internet using URLs. All URLs share the same basic format, although some variation is allowed.

A URL specification is based on four components. The first is the protocol to use, separated from the

rest of the locator by a colon (:). Common protocols are HTTP, FTP, gopher, etc. The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:). The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). It defaults to port 80, the predefined HTTP port. The fourth part is the actual file path.

Example showing the use of URL:

```
import java.net.*;

class HostInfo {
       public static void main(String args[]) throws IOException{
           URL hp = new URL("https://nast.edu.np/computer-
           engineering/");
           System.out.println("Protocol: " + hp.getProtocol());
           System.out.println("Port: " + hp.getPort());
           System.out.println("Host: " + hp.getHost());
           System.out.println("File: " + hp.getFile());
           System.out.println("Part: "+hp.getPath());
           System.out.println("Default Port: "+hp.getDefaultPort());
           System.out.println("Content: "+hp.getContent().toString());
           System.out.println("Authority: "+hp.getAuthority());
           System.out.println("Ref: "+hp.getRef());
      }
}
```

When we run this, you will get the following output:

Protocol: https
Port: -1
Host: nast.edu.np
File: /computer-engineering/
Part: /computer-engineering/
Default Port: 443
Content: sun.net.www.protocol.http.HttpURLConnection$HttpInputStream@551bdc27
Authority: nast.edu.np
Ref: null

Notice that the port is -1; this means that a port was not explicitly set. Given a URL object, we can retrieve the data associated with it.

URL Connections

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once

we make a connection to a remote server, we can use URLConnection to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and only make sense for URL objects that are using the HTTP protocol. URLConnection defines several methods. Some are as follows:

int getContentLength( )	Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned.
String getContentType( )	Returns the type of content found in the resource. This is the value of the content-type header field. Returns null if the content type is not available.
long getDate( )	Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT.
long getExpiration( )	Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable.
String getHeaderField(int idx)	Returns the value of the header field at index idx (Header field indexes begin at 0). Returns null if the value of idx exceeds the number of fields.
String getHeaderField(String fieldName)	Returns the value of header field whose name is specified by fieldName. Returns null if the specified name is not found.
String getHeaderFieldKey(int idx)	Returns the header field key at index idx. (Header field indexes begin at 0.) Returns null if the value of idx exceeds the number of fields.
Map<String, List<String>> getHeaderFields( )	Returns a map that contains all of the header fields and values.
long getLastModified( )	Returns the time and date, represented in terms of milliseconds since January 1, 1970 GMT, of the last modification of the resource. Zero is returned if the last-modified date is unavailable.
InputStream getInputStream( ) throws IOException	Returns an InputStream that is linked to the resource. This stream can be used to obtain the content of the resource.

To access the actual bits or content information of a URL, create a URLConnection object from it, using its openConnection( ) method. The following example creates a URLConnection using the openConnection( ) method of a URL object .

Write a program to read content from website;

ReadWebContent.java

```
public class ReadWebContent {
    public static void main(String[] args) throws IOException {
        URL url = new URL("https://nast.edu.np/computer-engineering/");
        URLConnection conn = url.openConnection();
        InputStream is = conn.getInputStream();
        int c;
        while ((c = is.read()) != -1) {
            System.out.print((char) c);
        }
        System.out.println("Total Length: " + is.available());
        is.close();
    }
}
```

4.4 Email Handling using Java Mail API

Sending an Email Using JavaMail

Step1: Download and include activation.jar and mail.jar file to Java Build Path

Step 2: Design a Sender class as:

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;

public class Sender {

final static String EMAIL = "javatrainingnepal@gmail.com";
final static String PASSWORD = "BE@2018_nast";
private static int RESULT = 0;

public static int send(String to, String subject, String body){
      try {
            Properties props = System.getProperties();
            props.setProperty("mail.transport.protocol", "smtp");
            props.setProperty("mail.host", "smtp.gmail.com");
            props.put("mail.smtp.auth", "true");
            props.put("mail.smtp.port", "25");
            props.put("mail.debug", "true");
            props.put("mail.smtp.socketFactory.port", "25");
            props.put("mail.smtp.socketFactory.class",
                        "javax.net.ssl.SSLSocketFactory");
            props.put("mail.smtp.socketFactory.fallback", "false");
            Session emailSession = Session.getInstance(props,
                        new javax.mail.Authenticator() {
                  protected PasswordAuthentication getPasswordAuthentication() {
                        return new PasswordAuthentication(EMAIL,PASSWORD);
                  }
            });
```

```
            emailSession.setDebug(true);
            Message message = new MimeMessage(emailSession);
            message.setFrom(new InternetAddress(EMAIL));

            message.setRecipients(Message.RecipientType.TO,
                        InternetAddress.parse(to));
            message.setSubject(subject);
            message.setText(body);

            Transport transport = emailSession.getTransport("smtps");
            transport.connect("smtp.gmail.com", EMAIL, PASSWORD);
                  transport.sendMessage(message, message.getAllRecipients());
                  RESULT = 1;

            } catch (MessagingException e) {
                  RESULT = 0;
            }
            return RESULT;
      }
}
```

Step3: Design a client page

```
<!DOCTYPE html>
<html>
<head>
<title>Simple Email Client Application</title>
</head>
<body>
      <h2>Email Client Application</h2>
      <h4>
            <% if (request.getParameter("status") != null) { String status =
            request.getParameter("status"); if (status.equals(new String("0"))) {
            out.print("unable to send"); } else { out.print("sent"); out.print("
            <meta http-equiv='refresh' content='2;url=send-email.jsp' />
            "); } } %>
      </h4>
      <form method="post" action="send.jsp">
            To: <input type="email" name="txtTo" /><br /> Subject: <input
                  type="text" name="txtSubject" /><br /> Message:
            <textarea name="txtMessage"></textarea>
            <br />   <input type="submit" name="btnSend" value="SEND" />
      </form>
</body>
</html>
```

Step4: Design and Implement Email Send Logic

```
<%@page import="email.Sender"%>
<%
      String to = request.getParameter("txtTo");
      String subject = request.getParameter("txtSubject");
```

```
        String body = request.getParameter("txtMessage");
        int status = Sender.send(to, subject, body);
        response.sendRedirect("send-email.jsp?status=" + status);
%>
```

4.5 Architecture of RMI

Remote Objects

The RMI (Java Remote Method Invocation) system is a mechanism that enables an object on one Java virtual machine to invoke methods on an object in another Java virtual machine. Any object whose methods can be invoked in this way must implement the java.rmi.Remote interface. When such an object is invoked, its arguments are marshalled and sent from the local virtual machine to the remote one, where the arguments are unmarshalled and used. When the method terminates, the results are marshalled from the remote machineand sent to the caller's virtual machine. To make a remote object accessible to other virtual machines, a program typically registers it with the RMI registry.

RMI (Remote Method Invocation)

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM. The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object. A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stuband skeleton objects:

stub

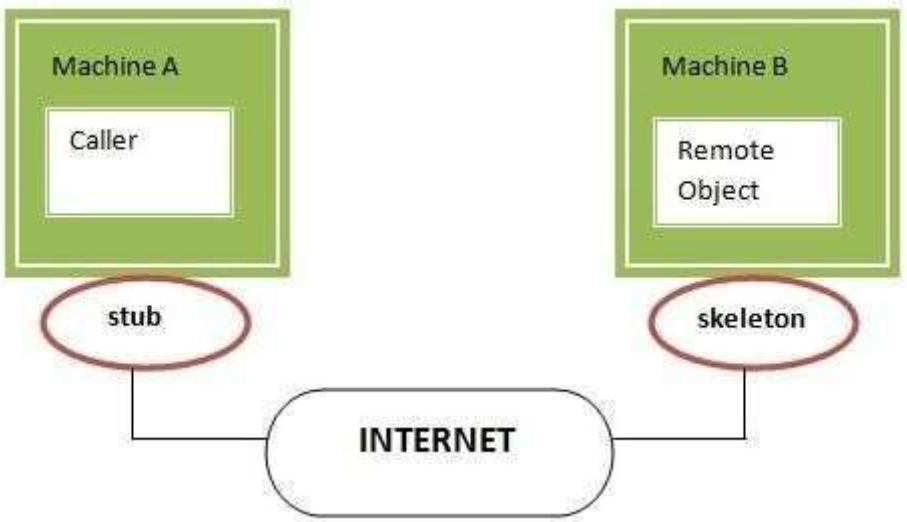
The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stubobject, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

The skeleton is an object, acts as a gateway for the server-side object. All the incoming requests are routedthrough it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.



Understanding requirements for the distributed applications

If any application performs these tasks, it can be distributed application.

1. The application needs to locate the remote method
2. It needs to provide the communication with the remote objects, and
3. The application needs to load the class definitions for the objects.

The RMI application has all these features, so it is called the distributed application.

Steps to write the RMI program

There are 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmi registry tool
5. Create and start the remote application
6. Create and start the client application

[Note: Step 3 and 4 are automatically created in the latest version of Java RMI Application)

The RMI Architecture is very simple involving a client program, a server program, a stub and skeleton. In RMI, the client and server do not communicate directly; instead communicates through **stub** and **skeleton** (a special concept of RMI and this is how designers achieved distributed computing in Java). They are nothing but special programs generated by **RMI compiler**. They can be treated as **proxies** for the client and server. Stub program resides on client side and skeleton program resides on server. That is, the client sends a method call with appropriate parameters to stub. The stub in turn calls the skeleton on the server. The skeleton passes the stub request to the server to execute the remote method. The return value of the method is sent to the skeletonby the server. The skeleton, as you expect, sends back to the stub. Finally, the return value reaches client throughstub.

Note: The method existing on the server is a remote method to the client (because client and server may be far away anywhere in the globe); but for the server, it is local method only. So, for the client, the server is remote server (okay, in server point of view, the client is remote client, but in RMI, we do not call client as remote client), a program on the remote server is remote program and finally the method in the remote program is remote method. Know the meaning of remote server, remote program, remote method; this must be very clear tothe beginner else the RMI concept is very confusing at the outset. So, anything residing on server is remote to client.

The RMI Architecture process is explained figure actively as under.
As the figure illustrates, there comes three layers in the RMI Architecture – Application layer, Proxy layer and Remote reference layer (Transport layer is part of Remote reference layer).

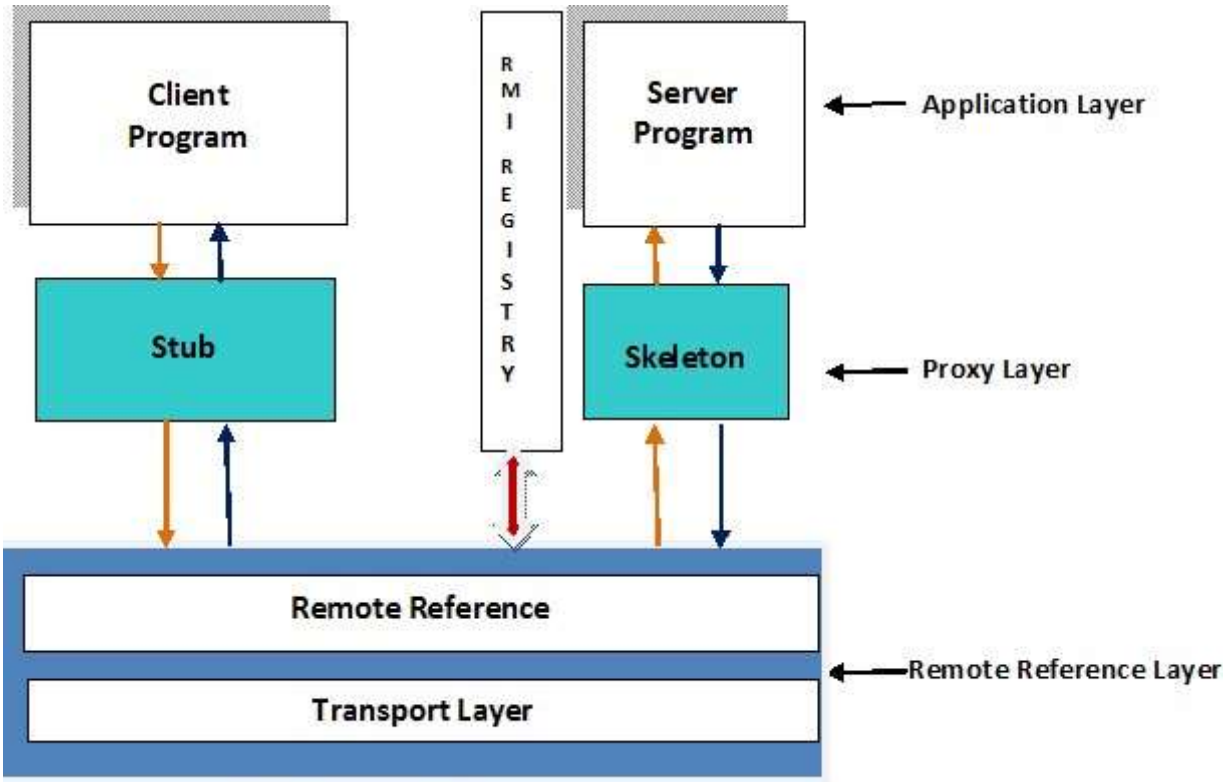


Fig: RMI Architecture

Application Layer

This layer is nothing but the actual systems (client and server) involved in communication. A client Java program communicates with the other Java program on the server side. RMI is nothing but a communication between two JVMs placed on different systems.

Proxy Layer

The proxy layer consists of **proxies** (named as **stub** and **skeleton** by designers) for client and server. **Stub is client-side proxy** and **Skeleton is server-side proxy**. Stub and skeleton, as many people confuse, are not separate systems but they are after all programs placed on client and server. The stub and skeleton are not hard coded by the Programmer but they are generated by RMI compiler (this is shown in execution part later). Stubis placed on client side and skeleton is placed on server side. The client server communication goes through these proxies. Client sends its request of method invocation (to be executed on remote server) to stub. Stub inturn sends the request to skeleton. Skeleton passes the request to the server program. Server executes the method and sends the return value to the skeleton (to route to client). Skeleton sends to stub and stub to client program.

Marshaling and Marshaling

One more job of proxies is marshaling and marshaling. **Marshaling** is nothing but converting data into a special format suitable to pass through the distributed environment without losing **object persistence**. For this reason, the RMI mechanism implicitly serialize the objects involved in communication. The stub marshals the data of client and then sends to the skeleton. As the format is not understood by the server program, it is marshaled by the skeleton into the original format and passed to server. Similarly, from server to client also.

A marshal stream includes a stream of objects that are used to transport parameters, exceptions, and errors needed for these streams for communicating with each other. Marshaling and marshaling are done by the RMI runtime mechanism implicitly without any programmers extra coding.

Remote Reference Layer

Proxies are implicitly connected to RMI mechanism through Remote reference layer, the layer responsible for object communication and transfer of objects between client and server. It is responsible for dealing with semantics of remote invocations and implementation – specific tasks with remote objects. In this layer, actual implementation of communication protocols is handled.

Transport layer does not exist separately but is a part of Remote reference layer. Transport layer is responsible for actually setting up connections and handling the transport of data from one machine to another.It can be modified to handle encrypted streams, compression algorithms and a number of othersecurity/performance related enhancements.

The marshaled stream is passed to **RRL** (Remote Reference Layer). Task of RRL is to identify the host, that is, remote machine. The marshaled stream is passed to Transport layer which performs routing.

4.6 Creating and Executing RMI Application

The RMI program can be developed and executed using the following four steps:

Step 1: Create an interface that will extends remote interface

Step 2: Write down the implementation class of the remote interface class

Step 3: Write down the server program

Step 4: Write down the client program.

- Run server program
- Run client program

Q. Create an RMI Client/Server program when the client program call the ***getMessage()*** method of server it should returns **Hello Client!** And display it on the client screen.

HelloInterface.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloInterface extends Remote{
	public String getMessage() throws RemoteException;
}
```

HelloInterfaceImpl.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloInterfaceImpl extends UnicastRemoteObject implements HelloInterface {
	public static final String MESSAGE = "Hello Client!";

	public HelloInterfaceImpl() throws RemoteException {
		super(0);	// required to avoid the 'rmic' step, see below
	}

	public String getMessage() {
		return MESSAGE;
	}
}
```

HelloServer.java

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

public class HelloServer {
	public static void main(String args[]) throws Exception {
		try {
			//special exception handler for registry creation
			Registry registry = LocateRegistry.createRegistry(1099);
			System.out.println("java RMI registry created.");

			//Instantiate HelloInterface
			HelloInterface obj = new HelloInterfaceImpl();

			// Bind this object instance to the name "HelloServer"
			registry.rebind("hello", obj);
			System.out.println("RMI server started");
```

```
		} catch (Exception e) {
			//do nothing, error means registry already exists
			System.out.println("java RMI registry already
			exists."+e.getMessage());
		}
	}
}
```

HelloClient.java

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

public class HelloClient {
	public static void main(String args[]) {
		try {
			Registry registry= LocateRegistry.getRegistry("192.168.1.91",1099);

			HelloInterface obj= (HelloInterface)registry.lookup("hello");


			System.out.println(obj.getMessage());

		}catch(Exception e) {
			System.out.println("Client Error: "+e.getMessage()+e);
		}
	}
}
```

[Note: The interface is same for both client and server program when hosting to different computer]

Q. Write an RMI Client server program to invoke remote method ***calculate (double, double, double)*** as parameters and which will return simple interest. Now create a client program to call that method and print the interest calculated by server.

InterestInterface.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface InterestInterface extends Remote {

	public double calculate(double p, double t, double r) throws RemoteException;
}
```

InterestInterfaceImpl.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
```

```
public class InterestInterfaceImpl extends UnicastRemoteObject implements
InterestInterface {

	public InterestInterfaceImpl() throws RemoteException {
		super(0);	// required to avoid the 'rmic' step, see below
	}

	public double calculate(double p, double t, double r) throws RemoteException {
		return (p*t*r/100);
	}
}
```

InterestServer.java

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

public class InterestServer {
	public static void main(String args[]) throws Exception {
		try {
			//special exception handler for registry creation
			Registry registry = LocateRegistry.createRegistry(1099);
			System.out.println("java RMI registry created.");

			//Instantiate InterestInterface
			InterestInterface obj = new InterestInterfaceImpl();

			// Bind this object instance to the name "si"
			registry.rebind("si", obj);
			System.out.println("RMI server started");

		} catch (Exception e) {
			//do nothing, error means registry already exists
			System.out.println("java RMI registry already
			exists."+e.getMessage());
		}
	}
}
```

[Note: The interface is same for both client and server program when hosting to different computer]

Q) Write an RMI Client/Server Program to calculate Profit or Loss through the help of Selling Price and CostPrice send by Client.

PLInterface.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface PLInterface extends Remote {
	public double calculatePL (double c, double s)throws RemoteException;
}
```

PLInterfaceImpl.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
```

```
public class PLInterfaceImpl extends UnicastRemoteObject implements PLInterface {

	public PLInterfaceImpl() throws RemoteException{
		super(0);
	}

	@Override
	public double calculatePL(double cp,double sp){
		return (sp-cp);
	}
}
```

PLServer.java

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

public class PLServer {
	public static void main(String args[]) throws Exception {
		try {
			//special exception handler for registry creation
			Registry registry = LocateRegistry.createRegistry(1099);
			System.out.println("java RMI registry created.");

			//Instantiate PLInterface
			PLInterface obj = new PLInterfaceImpl();

			// Bind this object instance to the name "pl"
			registry.rebind("pl", obj);
			System.out.println("RMI server started");

		} catch (Exception e) {
			//do nothing, error means registry already exists
			System.out.println("java RMI registry already
			exists."+e.getMessage());
		}
	}
}
```

PLClient.java

```
mport java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class PLClient {
	public static void main(String args[]) {
		try {
			Registry registry= LocateRegistry.getRegistry("192.168.1.91",1099);

			PLInterface obj= (PLInterface)registry.lookup("pl");


			double pl = obj.calculatePL(32000.33, 90000.23);

			if(pl>0){
				System.out.println("Profit: "+pl);
			}
```

```
                else{
                        System.out.println("Loss: "+Math.abs(pl));
                }

            }catch(Exception e) {
                System.out.println("Client Error: "+e.getMessage()+e);
            }
        }
}
```

[Note: The interface is same for both client and server program when hosting to different computer]

4.7 Architecture of CORBA

Need of CORBA (Why CORBA)

- Incompatibility of systems during data transfer
- Collaboration between system on different OS, programming languages or computing hardware was not possible.

History

- In 1989, 29 big MNCs took a decision against this issue
- No matter what on which technology stack a program or software in developed it should be compatible with each system.
- Created a platform called CORBA
- Group/Organization of these MNCs is called OMG (Object Management Group)
- First Version of CORBA was released in 1992 as Object Management Architecture Guide

What is CORBA?

- CORBA is a platform which supports multiple programming language work together successfully
- It is a mechanism in software for normalizing the method call semantics between application object residing either in the same address space or remote address space
- It is am middleware neither 2-tier or 3-tier architecture
- The object request broker (ORB) enables clients to invoke method in a remote object
- It is a technology to connect to objects of heterogeneous types.

Types of Object (CORBA)

- There are two types of Objects in CORBA
    - Service provider object
    - Client object

    #Service provider object: - Object that includes functionalities that can be used by another object

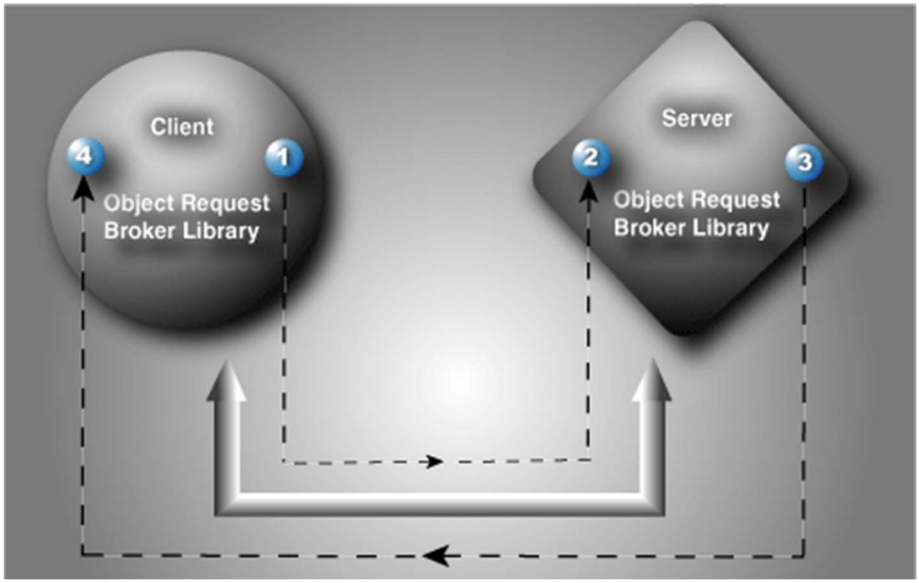
client object: - Object that requires services of other objects

How CORBA Works?

What is IDL?

- IDL (Interface Definition Language)
- Used to specify the interface which objects represent to the outer world.
- CORBA specifying a mapping from IDL to specific implementation like C++, or Java
- IDL Compiler is a tool to implement CORBA

Working



1. Client sends request to the server
2. Server receives a request from the client
3. Server sends reply to the client
4. Client gets reply from the server

#CORBA Key Elements
#ORB Core – It carries out the request – reply protocol between the client and server
#Object Adapter (Server) – Bridges gap between the CORBA object and programming language interface of the slave class
#skeletons (Server) IDL compiler generates skeleton class in the server's language
#Clients Proxies (Stubs)- Generated by IDL compiler in client language
#implementaiton repositories - Activities register server on demand and locates servers that are currently running
#Interface repository – Provides information about registers IDL interfaces to the client and server that request it

Applications

- Used as RMI (Remote Method Invocation) of a distributed client server system
- Enable communication between client and server on different OS programming language or hardware.
- Used in distributed applications
- Ideal for heterogenous distributed system like internet

Features of CORBA

- Language independence – CORBA provides its user the flexibility for programming language
- OS independence – CORBA is available in Java (OS independent), Linux/Uni, Windows, Mac etc.
- Freedom from technologies – C++ legacy code can take to C/Fortran legacy code and Java/Database code and can provide data to a web interface
- String data typing – CORBA provides flexible data typing
- Freedom from data transfer details – CORBA provides a high level of detail in error conditions.
- Compression – CORBA compresses its data in binary form

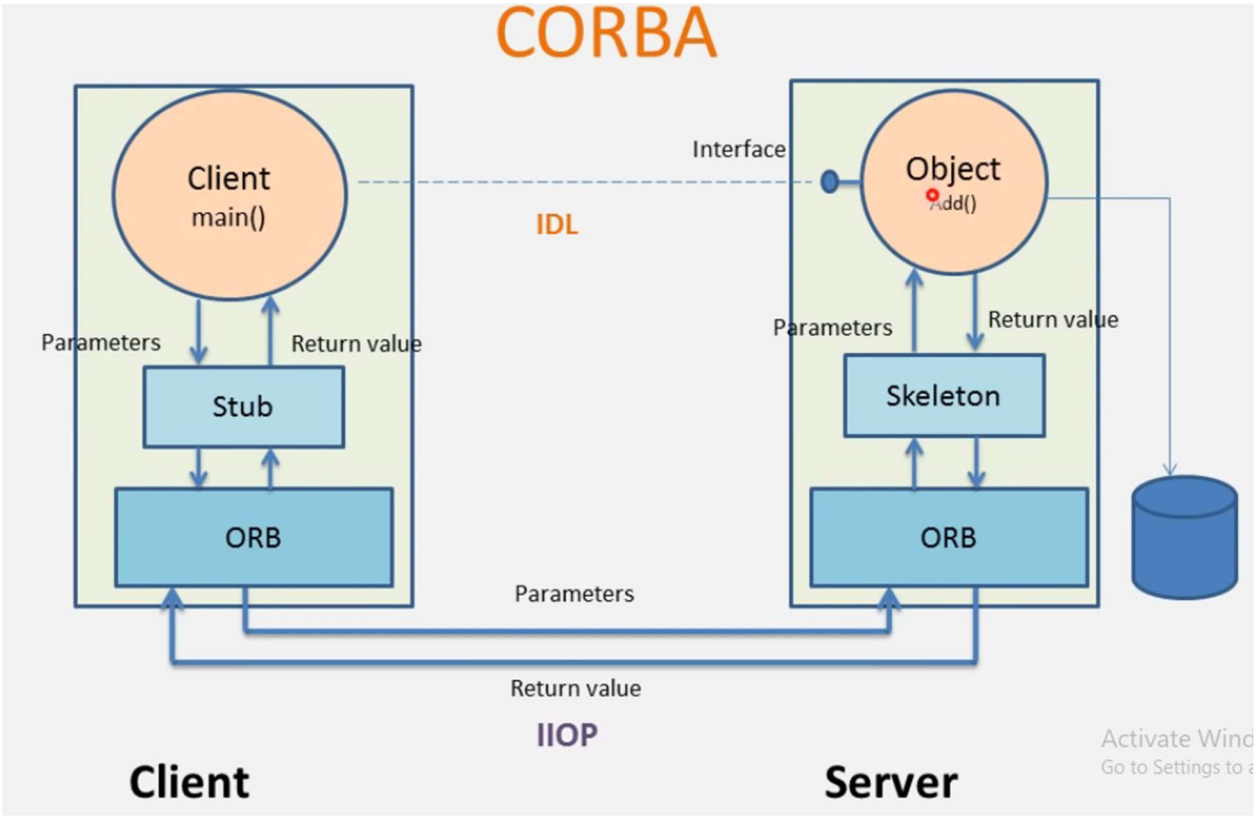


Fig: CORBA Architecture

4.8 RMI vs CORBA

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

4.9 IDL and Simple CORBA Program

Please write the Simple CORBA program and Submit as Assignment