

APPENDIX

I. Java.io Package

Provides for system input and output through data streams, serialization and the file system.

A) DataInputStream

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.

1) public DataInputStream(InputStream in)

Creates a DataInputStream that uses the specified underlying InputStream.

Parameters:

in - the specified input stream

2) public static final String readUTF(DataInput in) throws IOException

Reads from the stream in a representation of a Unicode character string encoded in modified UTF-8 format; this string of characters is then returned as a String.

Parameters:

in - a data input stream.

Returns:

a Unicode string.

B) DataOutputStream

A data output stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

1) public DataOutputStream(OutputStream out)

Creates a new data output stream to write data to the specified underlying output stream. The counter written is set to zero.

Parameters:

out - the underlying output stream, to be saved for later use.

2) public final void writeUTF(String str) throws IOException

Writes a string to the underlying output stream using modified UTF-8 encoding in a machine-independent manner.

Parameters:

str - a string to be written.

II. Java.net Package

Provides the classes for implementing networking applications.

A) InetAddress

This class represents an Internet Protocol (IP) address.

1) public static InetAddress getByName(String host)

throws UnknownHostException

Determines the IP address of a host, given the host's name.

Parameters:

host - the specified host, or null.

Returns:

an IP address for the given host name.

B) Socket

This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.

1) public Socket(String host, int port, InetAddress localAddr, int localPort) throws IOException

Creates a socket and connects it to the specified remote host on the specified remote port. The Socket will also bind() to the local address and port supplied.

Parameters:

host - the name of the remote host, or null for the loopback address.

port - the remote port

localAddr - the local address the socket is bound to, or null for the anyLocal address.

localPort - the local port the socket is bound to, or zero for a system selected free port.

2) public InputStream getInputStream() throws IOException

Returns an input stream for this socket.

Returns:

an input stream for reading bytes from this socket.

3) public OutputStream getOutputStream() throws IOException

Returns an output stream for this socket.

Returns:

an output stream for writing bytes to this socket.

4) public void close() throws IOException

Closes this socket.

C) ServerSocket

This class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.

1) public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException

Create a server with the specified port, listen backlog, and local IP address to bind to.

Parameters:

port - the port number, or 0 to use a port number that is automatically allocated.

backlog - requested maximum length of the queue of incoming connections.

bindAddr - the local InetAddress the server will bind to

2) public Socket accept() throws IOException

Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.

Returns:

the new Socket

3) public void close() throws IOException

Closes this socket. Any thread currently blocked in accept() will throw a SocketException.

If this socket has an associated channel then the channel is closed as well.

Throws:

IOException - if an I/O error occurs when closing the socket.

D) DatagramPacket

This 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.

1) **public DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)**

Constructs a datagram packet for sending packets of length with the offset to the specified port number on the specified host. The length argument must be less than or equal to buf.length.

Parameters:

buf - the packet data.

offset - the packet data offset.

length - the packet data length.

address - the destination address.

port - the destination port number.

2) **public byte[] getData()**

Returns the data buffer. The data received or the data to be sent starts from the offset in the buffer, and runs for length long.

Returns:

the buffer used to receive or send data

3) **public InetAddress getAddress()**

Returns the IP address of the machine to which this datagram is being sent or from which the datagram was received.

Returns:

the IP address of the machine to which this datagram is being sent or from which the datagram was received.

4) **public int getPort()**

Returns the port number on the remote host to which this datagram is being sent or from which the datagram was received.

Returns:

the port number on the remote host to which this datagram is being sent or from which the datagram was received.

E) DatagramSocket

This class represents a socket for sending and receiving datagram packets.

A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may be routed differently, and may arrive in any order.

1) public DatagramSocket(int port, InetAddress laddr) throws SocketException

Creates a datagram socket, bound to the specified local address..

Parameters:

port - local port to use

laddr - local address to bind

2) public void send(DatagramPacket p) throws IOException

Sends a datagram packet from this socket. The DatagramPacket includes information indicating the data to be sent, its length, the IP address of the remote host, and the port number on the remote host.

Parameters:

p - the DatagramPacket to be sent.

3) public void receive(DatagramPacket p) throws IOException

Receives a datagram packet from this socket. When this method returns, the DatagramPacket's buffer is filled with the data received. The datagram packet also contains the sender's IP address, and the port number on the sender's machine.

Parameters:

p - the DatagramPacket into which to place the incoming data.

F) MalformedURLException

Thrown to indicate that a malformed URL has occurred. Either no legal protocol could be found in a specification string or the string could not be parsed.

III. Java.rmi Package

Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism.

A) Remote (Interface)

The Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a "remote interface", an interface that extends `java.rmi.Remote` are available remotely.

B) RemoteException

A `RemoteException` is the common superclass for a number of communication-related exceptions that may occur during the execution of a remote method call. Each method of a remote interface, an interface that extends `java.rmi.Remote`, must list `RemoteException` in its throws clause

C) java.rmi.server.UnicastRemoteObject

Used for exporting a remote object with JRMP and obtaining a stub that communicates to the remote object. Stubs are either generated at runtime using dynamic proxy objects, or they are generated statically at build time, typically using the `rmic` tool.

D) Naming

The `Naming` class provides methods for storing and obtaining references to remote objects in a remote object registry. Each method of the `Naming` class takes as one of its arguments a name that is a `java.lang.String` in URL format (without the scheme component) of the form:

`//host:port/name`

where `host` is the host (remote or local) where the registry is located, `port` is the port number on which the registry accepts calls, and where `name` is a simple string uninterpreted by the registry.

1) public static void bind(String name, Remote obj)
throws AlreadyBoundException,
MalformedURLException,
RemoteException

Binds the specified name to a remote object.

Parameters:

`name` - a name in URL format (without the scheme component)

`obj` - a reference for the remote object (usually a stub)

2) **public static Remote lookup(String name)**

**throws NotBoundException,
MalformedURLException,
RemoteException**

Returns a reference, a stub, for the remote object associated with the specified name.

Parameters:

name - a name in URL format (without the scheme component)

Returns:

a reference for a remote object

E) java.rmi.registry.LocateRegistry

LocateRegistry is used to obtain a reference to a bootstrap remote object registry on a particular host (including the local host), or to create a remote object registry that accepts calls on a specific port.

1)public static Registry createRegistry(int port) throws RemoteException

Creates and exports a Registry instance on the local host that accepts requests on the specified port.

Parameters:

port - the port on which the registry accepts requests

Returns:

the registry

IV. org.apache.commons.net.ftp Package

Contains FTP and FTPS support classes

A) FTPClient

FTPClient encapsulates all the functionality necessary to store and retrieve files from an FTP server. This class takes care of all low level details of interacting with an FTP server and provides a convenient higher level interface.

1) public FTPClient()

Default FTPClient constructor.

2) public void connect(InetAddress host) throws SocketException, IOException

Opens a Socket connected to a remote host at the current default port and originating from the current host at a system assigned port. Before returning, `_connectAction_()` is called to perform connection initialization actions.

Parameters:

host - The remote host.

3) public int getReplyCode()

Returns the integer value of the reply code of the last FTP reply. You will usually only use this method after you connect to the FTP server to check that the connection was successful since connect is of type void.

Returns:

The integer value of the reply code of the last FTP reply.

4) public boolean login(String username, String password) throws IOException

Login to the FTP server using the provided username and password.

Parameters:

username - The username to login under.

password - The password to use.

Returns:

True if successfully completed, false if not.

5) public boolean logout() throws IOException

Logout of the FTP server by sending the QUIT command.

Returns:

True if successfully completed, false if not.

6) public void disconnect() throws IOException

Closes the connection to the FTP server and restores connection parameters to the default values.

7) public FTPFile[] listFiles() throws IOException

Using the default system autodetect mechanism, obtain a list of file information for the current working directory.

Returns:

The list of file information contained in the current directory in the format determined by the autodetection mechanism.

8) public boolean storeFile(String remote, InputStream local) throws IOException

Stores a file on the server using the given name and taking input from the given InputStream.

Parameters:

remote - The name to give the remote file.

local - The local InputStream from which to read the file.

Returns:

True if successfully completed, false if not.

**9) public boolean retrieveFile(String remote, OutputStream local)
throws IOException**

Retrieves a named file from the server and writes it to the given OutputStream.

Parameters:

remote - The name of the remote file.

local - The local OutputStream to which to write the file.

Returns:

True if successfully completed, false if not.

B) FTPReply

FTPReply stores a set of constants for FTP reply codes.

1) public static boolean isPositiveCompletion(int reply)

Determine if a reply code is a positive completion response. All codes beginning with a 2 are positive completion responses. The FTP server will send a positive completion response on the final successful completion of a command.

Parameters:

reply - The reply code to test.

Returns:

True if a reply code is a positive completion response, false if not.

C) FTPFile

The FTPFile class is used to represent information about files stored on an FTP server.

1) public String getName()

Return the name of the file.

Returns:

The name of the file.

V. java.sql Package

Provides the API for accessing and processing data stored in a data source (usually a relational database) using the JavaTM programming language.

A) Connection (Interface)

A connection (session) with a specific database. SQL statements are executed and results are returned within the context of a connection.

1) Statement createStatement() throws SQLException

Creates a Statement object for sending SQL statements to the database. SQL statements without parameters are normally executed using Statement objects.

Returns:

a new default Statement object

2) void close() throws SQLException

Releases this Connection object's database and JDBC resources immediately instead of waiting for them to be automatically released.

B) DriverManager

The basic service for managing a set of JDBC drivers. As part of its initialization, the DriverManager class will attempt to load the driver classes referenced in the "jdbc.drivers" system property. This allows a user to customize the JDBC Drivers used by their applications.

1) public static Connection getConnection(String url,String user,String password) throws SQLException

Attempts to establish a connection to the given database URL. The DriverManager attempts to select an appropriate driver from the set of registered JDBC drivers.

Parameters:

url - a database url of the form jdbc:subprotocol:subname

user - the database user on whose behalf the connection is being made

password - the user's password

Returns:

a connection to the URL

C) Statement (Interface)

The object used for executing a static SQL statement and returning the results it produces.

1) ResultSet executeQuery(String sql) throws SQLException

Executes the given SQL statement, which returns a single ResultSet object.

Note: This method cannot be called on a PreparedStatement or CallableStatement.

Parameters:

sql - an SQL statement to be sent to the database, typically a static SQL SELECT statement

Returns:

a ResultSet object that contains the data produced by the given query; never null

D) ResultSet (Interface)

A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

1) ResultSetMetaData getMetaData() throws SQLException

Retrieves the number, types and properties of this ResultSet object's columns.

Returns:

the description of this ResultSet object's columns

2) boolean next() throws SQLException

Moves the cursor forward one row from its current position. A ResultSet cursor is initially positioned before the first row; the first call to the method next makes the first row the current row; the second call makes the second row the current row, and so on.

Returns:

true if the new current row is valid; false if there are no more rows

3) String getString(int columnIndex) throws SQLException

Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.

Parameters:

columnIndex - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is null

E) ResultSetMetaData

An object that can be used to get information about the types and properties of the columns in a ResultSet object.

1) int getColumnCount() throws SQLException

Returns the number of columns in this ResultSet object.

Returns:

the number of columns

Throws:

SQLException - if a database access error occurs

2) String getColumnName(int column) throws SQLException

Get the designated column's name.

Parameters:

column - the first column is 1, the second is 2, ...

Returns:

column name

Throws:

SQLException - if a database access error occurs

PROGRAM - 1

Program Statement:

Implement a GUI based UDP Client program for Well Known Service – Echo.

Description of the program:

This is a UDP Client Program that uses the User Datagram Protocol (UDP) to send a message to a server at a specified IP address and port number (7). The client takes the input message from a text field, creates a DatagramSocket, and uses it to send the message as a DatagramPacket to the server. It then waits for a response from the server and receives it as a DatagramPacket, which is then displayed in a text area.

An Echo Service is a Service which always returns back the same message that it receives from the client.

Input:

IP Address – 10.2.0.5

Port No. -- 7

Message -- XYZ

Output:

The message sent to the server is sent back as a response.

Packages Used:

- java.io.*;
- java.net.*;

IDE Used: Netbeans

Classes and Methods Used:

- DatagramSocket
 - send()
 - receive()
 - close()
- DatagramPacket
 - DatagramPacket()
- InetAddress
 - getByName()

Program:

```
package udpwellknown42;
import java.io.*;
import java.net.*;

public class UDPClientWK extends javax.swing.JFrame {
    private void SendButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        String ipAddress = IPField.getText();
        int portNo = Integer.parseInt(portField.getText());
        String message = MessageField.getText();
        try {
            DatagramSocket ds = new DatagramSocket();
            byte[] sndmsg = message.getBytes();

            DatagramPacket sndPacket = new DatagramPacket(sndmsg,
sndmsg.length, InetAddress.getByName(ipAddress), portNo);

            ds.send(sndPacket);

            byte[] rcvmsg = new byte[40];
            DatagramPacket rcvPacket = new DatagramPacket(rcvmsg, 40);
            ds.receive(rcvPacket);
            String response = new String(rcvPacket.getData());

            ServerResponseArea.append("Received: " + response + "\n");

            ds.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Output Images:



The image shows a window titled "Client Echo" with a light gray background. It contains several input fields and a button. The "IP Address" field contains "10.2.0.5". The "Port Number" field contains "7". The "Message" field contains "UDP Well Known". Below these fields is a "Submit" button. At the bottom, the "Response" field also contains "UDP Well Known". The window has standard OS window controls (minimize, maximize, close) in the top right corner.

Field	Value
IP Address	10.2.0.5
Port Number	7
Message	UDP Well Known
Submit	Submit
Response	UDP Well Known

PROGRAM - 2

Program Statement:

Implement a GUI based TCP Client program for Well Known Service – Echo.

Description of the program:

This is a TCP Client Program that uses the Transmission Control Protocol (TCP) to send a message to a server at a specified IP address and port number (7). The client takes the input message from a text field, creates a Socket, and uses it to send the message on its DataOutputStream to the server. It then waits for a response from the server and receives it in DataInputStream, which is then displayed in a text area.

An Echo Service is a Service which always returns back the same message that it receives from the client.

Input:

IP Address – 10.2.0.5

Port No. -- 7

Message -- XYZ

Output:

The message sent to the server is sent back as a response.

Packages Used:

- java.io.*;
- java.net.*;

IDE Used: Netbeans

Classes and Methods Used:

- Socket
 - Socket()
 - getInputStream()
 - getOutputStream()
 - close()
- DataInputStream
 - readUTF()
- DataOutputStream
 - writeUTF()
- InetAddress
 - getByName()

Program:

```
package tcpwellknown42;

import java.net.*;
import java.io.*;

public class TCPEchoClientWK extends javax.swing.JFrame {
    private void SendButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        String ip = IPAddressField.getText();
        int port = Integer.parseInt(PortNumber.getText());
        String msg = RequestMessage.getText();

        try{
            Socket s = new Socket(InetAddress.getByName(ip), port);
            DataInputStream dis = new DataInputStream(s.getInputStream());

            DataOutputStream dos = new DataOutputStream(s.getOutputStream());

            dos.writeUTF(msg);
            String response = dis.readUTF();
            ServerResponse.append("Echo: " + msg + "\n");
            s.close();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Output Images:



Client Echo

IP Address: 10.2.0.5

Port Number: 7

Message: well known

Submit

Response: meee
well known

PROGRAM - 3

Program Statement:

Implement a GUI based UDP Client – Server program for Echo Service.

Description of the program:

This is a UDP Client-Server Echo Service Program that creates a Client Application that uses the User Datagram Protocol (UDP) to send a message to a server at a specified IP address and port number. The client takes the input message from a text field, the IP address and port number from text fields as well, creates a DatagramSocket, and uses it to send the message as a DatagramPacket to the server. It then waits for a response from the server and receives it as a DatagramPacket, which is then displayed in a text area.

An Echo Service is a Service which always returns back the same message that it receives from the client.

Input:

IP Address – localhost

Port No. – 50xx

Message -- XYZ

Output:

The message sent to the server is sent back as a response.

Packages Used:

- java.io.*;
- java.net.*;

IDE Used: Netbeans

Classes and Methods Used:

- DatagramSocket
 - send()
 - receive()
 - close()
- DatagramPacket
 - DatagramPacket()
- InetAddress
 - getByName()

Program:**Client:**

```
package udpapplication42;

import java.io.*;
import java.net.*;

public class UDPClient extends javax.swing.JFrame {
    private void SendButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        String ipAddress = IPField.getText();
        int portNo = Integer.parseInt(portField.getText());
        String message = MessageField.getText();

        try {
            DatagramSocket ds = new DatagramSocket();

            byte[] sndmsg = message.getBytes();
            DatagramPacket sndPacket = new DatagramPacket(sndmsg,
sndmsg.length, InetAddress.getByName(ipAddress), portNo);
            ds.send(sndPacket);

            byte[] rcvmsg = new byte[40];
            DatagramPacket rcvPacket = new DatagramPacket(rcvmsg, 40);
            ds.receive(rcvPacket);
            String response = new String(rcvPacket.getData());

            ServerResponseArea.append("Received: " + response + "\n");

            ds.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Server:

```
package udpapplication42;
import java.io.*;
import java.net.*;
public class UDPServer extends javax.swing.JFrame implements Runnable{
    private void StartButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        Thread thread = new Thread(this, "server");
        StartButton.setEnabled(false);
        thread.start();
        MessageArea.append("Server Started");
    }
    @Override
    public void run() {
        String ipAddress = IPField.getText();
        int portNo = Integer.parseInt(PortField.getText());

        try {
            DatagramSocket ds = new DatagramSocket(portNo,
            InetAddress.getByName(ipAddress));

            while(true) {
                byte[] rcvMessage = new byte[40];
                DatagramPacket rcvPacket = new DatagramPacket(rcvMessage,
                40);

                ds.receive(rcvPacket);

                InetAddress clientIP = rcvPacket.getAddress();
                int clientPort = rcvPacket.getPort();

                String received = new String(rcvPacket.getData());
                MessageArea.append("\nClient " + clientIP + ": " +
                received + "\n");

                byte[] sndmsg = received.getBytes();
                DatagramPacket sndPacket = new DatagramPacket(sndmsg,
                sndmsg.length, clientIP, clientPort);
                ds.send(sndPacket);
            }

            //ds.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Output Images:

The image displays two side-by-side screenshots of a network application interface, likely a Java Swing or Qt-based GUI.

Left Window: UDP Client

- IP Address:** A text input field containing "localhost".
- Port No:** A text input field containing "5041".
- Message:** A text input field containing "udpcs".
- Send Request:** A button with a blue border and a small icon on the left.
- Output:** A text area containing "udpcs".

Right Window: UDP Server

- IP Address:** A text input field containing "localhost".
- Port no:** A text input field containing "5041".
- Start Server:** A button with a grey background and a small icon on the left.
- Message:** A text area containing the text "Server listening..." and "Client [/127.0.0.1] : udpcs".

PROGRAM - 4

Program Statement:

Implement a GUI based TCP Client – Server program for Echo Service.

Description of the program:

The is a Java Program that creates a Client Application that uses the Transmission Control Protocol (TCP) to send a message to a server at a specified IP address and port number. The client takes the input message from a text field, the IP address and port number from text fields as well, creates a Socket and connects it to the server using the IP address and port number. Then the client creates DataInputStream and DataOutputStream to read and write the data on the socket respectively. It sends the message to the server by writing it to the DataOutputStream, then reads the response from the server by reading it from the DataInputStream and displays it in a text area. The client then closes the socket.

An Echo Service is a Service which always returns back the same message that it receives from the client.

Input:

IP Address – localhost

Port No. – 50xx

Message -- XYZ

Output:

The message sent to the server is sent back as a response.

Packages Used:

- java.io.*;
- java.net.*;

IDE Used: Netbeans

Classes and Methods Used:

- Socket
 - Socket()
 - getInputStream()
 - getOutputStream()
 - close()
- DataInputStream
 - readUTF()
- DataOutputStream
 - writeUTF()

- ServerSocket
 - ServerSocket()
 - accept()
- InetAddress
 - getByName()

Program:

Client:

```
package tcpclientserverby42;

import java.net.*;
import java.io.*;

public class TCPEchoClient extends javax.swing.JFrame {
    private void SendButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        String ip = IPAddressField.getText();
        int port = Integer.parseInt(PortNumber.getText());
        String msg = RequestMessage.getText();

        try{
            Socket s = new Socket(InetAddress.getByName(ip), port);
            DataInputStream dis = new DataInputStream(s.getInputStream());

            DataOutputStream dos = new
DataOutputStream(s.getOutputStream());

            dos.writeUTF(msg);
            String response = dis.readUTF();
            ServerResponse.append(msg + "\n");
            s.close();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```


Server:

```
package tcpclientserverby42;

import java.net.*;
import java.io.*;
import java.lang.Runnable;

public class TCPEchoServer extends javax.swing.JFrame implements Runnable
{
    private void StartButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        Thread t = new Thread(this, "Server");
        t.start();
        StartButton.setEnabled(false);
        ComWindow.append("Server Listening\n");
    }
    @Override
    public void run() {
        String ip = IPAddressField.getText();
        int port = Integer.parseInt(PortNumber.getText());

        try{

            ServerSocket ss = new ServerSocket(port, 5,
InetAddress.getByAddress(ip));

            while(true){
                Socket s = ss.accept();

                DataInputStream dis = new
DataInputStream(s.getInputStream());

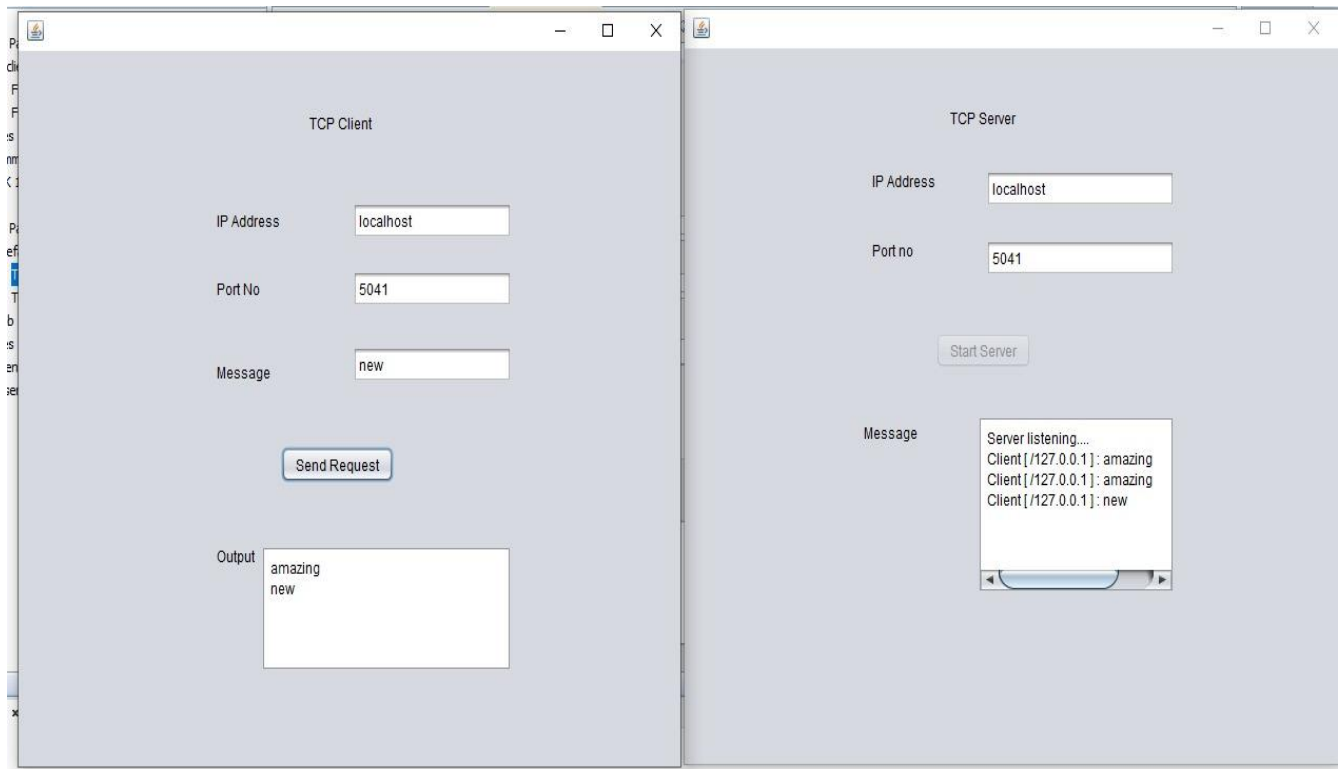
                DataOutputStream dos = new
DataOutputStream(s.getOutputStream());

                String req = dis.readUTF();
                ComWindow.append(req + "\n");
                dos.writeUTF(req);
                s.close();
            }
        }

        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

```
}  
}
```

Output Images:



PROGRAM - 5

Program Statement:

Implement a GUI based Client – Server program for Bulletin Board Application.

Description of the program:

The is a Java Program that creates a Client Application that connects to a Bulletin Board Server using the Transmission Control Protocol (TCP) at a specified IP address and port number. The client takes the input message from a text field, the IP address and port number from text fields as well, creates a Socket and connects it to the server using the IP address and port number. Then the client creates DataOutputStream to write the data on the socket. It sends the message to the server by writing it to the DataOutputStream, the server reads the message from the client and displays it in a text area. The client then closes the socket.

A Bulletin Board Application is an application dedicated to the sharing or exchange of messages on a network.

Input:

IP Address – localhost

Port No. – 50xx

Message -- XYZ

Output:

The messages sent by multiple clients to bulletin server are displayed at the server.

Packages Used:

- java.io.*;
- java.net.*;

IDE Used: Netbeans

Classes and Methods Used:

- Socket
 - Socket()
 - getInputStream()
 - getOutputStream()
 - close()
- DataInputStream
 - readUTF()
- DataOutputStream
 - writeUTF()
- ServerSocket
 - ServerSocket()
 - accept()

- InetAddress
 - getByName()

Program:

Client:

```
package bulletinboardby42;

import java.net.*;
import java.io.*;

public class BulletinBoardClient extends javax.swing.JFrame {
    private void SendButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        String ip = IPField.getText();
        int port = Integer.parseInt(PortField.getText());
        String msg = MessageField.getText();

        try{
            Socket s = new Socket(InetAddress.getByName(ip), port);

            DataInputStream dis = new DataInputStream(s.getInputStream());

            DataOutputStream dos = new
            DataOutputStream(s.getOutputStream());

            dos.writeUTF(msg);
            String res = dis.readUTF();
            ResponseArea.append(res + "\n");
            s.close();

        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Server:

```
package bulletinboardby42;

import java.net.*;
import java.io.*;
import java.lang.Runnable;
public class BulletinBoardServer extends javax.swing.JFrame implements
Runnable{
private void StartServerButtonActionPerformed(java.awt.event.ActionEvent
evt) {
    // TODO add your handling code here:

    Thread t = new Thread(this, "Server");
    t.start();
    StartServerButton.setEnabled(false);
    ResponseArea.append("Server Listening\n");
}

@Override
public void run(){
    String ip = IPField.getText();
    int port = Integer.parseInt(PortField.getText());

    try{

        ServerSocket ss = new ServerSocket(port, 5,
InetAddress.getByIp(ip));

        while(true){
            Socket s = ss.accept();

            DataInputStream dis = new
DataInputStream(s.getInputStream());

            DataOutputStream dos = new
DataOutputStream(s.getOutputStream());

            String announcement = dis.readUTF();

            ResponseArea.append("\n" + announcement);

            dos.writeUTF("Received Message");
            s.close();
        }

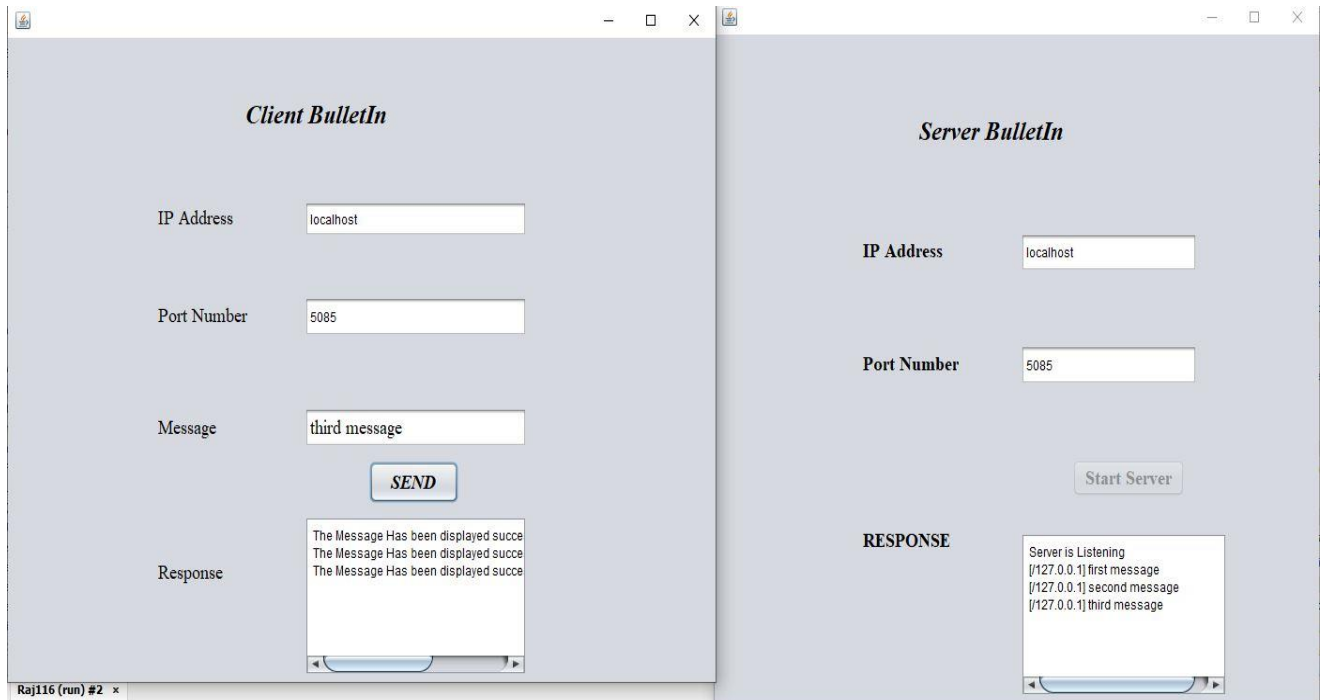
    }
}
```

```

        catch(Exception e){
            e.printStackTrace();
        }
    }
}
}

```

Output Images:



PROGRAM - 6

Program Statement:

Implement a GUI based Client – Server program for Name Service/DNS Application.

Description of the program:

The is a Java Program that creates a Client Application that uses the Transmission Control Protocol (TCP) to send a URL to a DNS (Domain Name Service) Server at a specified IP address and port number. The client takes the input message from a text field, the IP address and port number from text fields as well, creates a Socket and connects it to the server using the IP address and port number. Then the client creates DataInputStream and DataOutputStream to read and write the data on the socket respectively. It sends the message to the server by writing it to the DataOutputStream, then reads the response from the server by reading it from the DataInputStream and displays it in a text area. The client then closes the socket.

A DNS server is a computer server that contains a database of public IP addresses and their associated hostnames, and in most cases serves to resolve, or translate, those names to IP addresses as requested.

Input:

IP Address – localhost

Port No. – 50xx

URL – www.abc.com

Output:

The Client sends a URL to the server and gets its Ip address as response.

Packages Used:

- java.io.*
- java.net.*

Classes and Methods Used:

- Socket
 - Socket()
 - getInputStream()
 - getOutputStream()
 - close()
- DataInputStream
 - readUTF()
- DataOutputStream
 - writeUTF()
- ServerSocket
 - ServerSocket()
 - accept()

- InetAddress
 - getByName()

Program:

Server:

```
package nameserver42;
import java.io.*;
import java.net.*;
import java.util.Scanner;
import java.lang.Runnable;

public class NameServer extends javax.swing.JFrame implements Runnable {
    private void startButtonActionPerformed(java.awt.event.ActionEvent
    evt) {
        // TODO add your handling code here:
        Thread t = new Thread(this, "server");
        startButton.setEnabled(false);
        t.start();
    }
    @Override
    public void run() {
        String ipAddress = ipField.getText();
        int portNo = Integer.parseInt(portField.getText());

        try{
            ServerSocket ss = new ServerSocket(portNo, 5,
            InetAddress.getByAddress(ipAddress));

            while(true){
                try(
                    Socket s = ss.accept();

                    DataOutputStream dos = new
DataOutputStream(s.getOutputStream());

                    DataInputStream dis = new
DataInputStream(s.getInputStream());){

                    String url = dis.readUTF();
                    boolean found = false;

                    File f = new
File("C:\\Users\\admin\\Desktop\\NameServer42\\src\\nameserver42\\data.txt
");

                    Scanner scanner = new Scanner(f);
```



```

        while(scanner.hasNextLine()){
            String[] tokens = scanner.nextLine().split(" ");
            if(url.equals(tokens[1])){
                dos.writeUTF(tokens[0]);
                found = true;
            }
        }
        if(!found){
            dos.writeUTF("Not Found");
        }
    } catch(Exception e) {
        e.printStackTrace();
    }
}

} catch(Exception e) {
    e.printStackTrace();
} //To change body of generated methods, choose Tools | Templates.
}
}

```

Client:

```

package nameserver42;
import java.net.*;
import java.io.*;

public class NameClient extends javax.swing.JFrame {
    private void sendButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        String ipAddress = ipField.getText();
        int portNo = Integer.parseInt(portField.getText());
        String url = urlField.getText();

        try{
            Socket s = new Socket(InetAddress.getByName(ipAddress),
            portNo);

            DataOutputStream dos = new
            DataOutputStream(s.getOutputStream());

            DataInputStream dis = new
            DataInputStream(s.getInputStream()); {

                dos.writeUTF(url);
                serverResponseArea.append(url);
            }
        }
    }
}

```

```

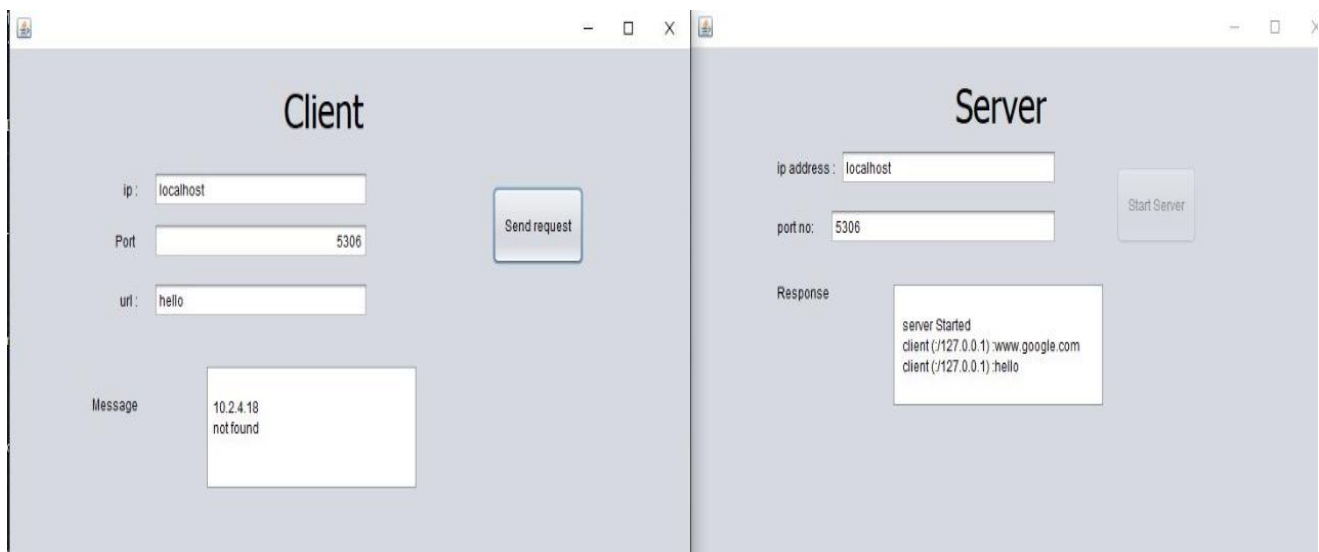
        String response = dis.readUTF();
        serverResponseArea.append(": " + response + "\n");
        s.close();

    } catch (Exception e) {
        e.printStackTrace();
    }

}
}

```

Output Images:



PROGRAM - 7

Program Statement:

Implement a GUI based Client – Server program for Chat application.

Description of the program:

This is a Java Program that implements a Chat Server and Client using sockets. The program has two main parts: the Server and the Client. A Chat Application allows two users to send messages to each other and read the responses as well.

The server listens for incoming connections from clients on a specified IP address and port number. When a client connects, the server creates a new socket for the client, reads messages sent by the client using a `DataInputStream`, and writes responses to the client using a `DataOutputStream`. The server also displays the received messages in a text area on the GUI.

The client connects to the server using the specified IP address and port number. The client also uses a `DataInputStream` and `DataOutputStream` to read and write messages to the server. The client also has a text field on the GUI where the user can enter a message to send to the server. Additionally, the client displays the sent messages in a text area on the GUI.

The Start button on the GUI starts the server thread, and the Send button sends messages from the client to the server. The Exit button closes the connection between the client and server and closes the program.

Chat Application refers to an Application which allows the process of communicating, interacting and/or exchanging messages over the Internet. It involves two or more individuals that communicate with each other in real time.

Input:

Client1

IP Address – localhost

Port No. – 5043

Message – ABC

Server Port – 5023

Client2

IP Address – localhost

Port No. – 5023

Message – XYZ

Server Port – 5043

Output:

The message sent by client 1 is received by server of client2 and the message sent by client 2 is received by server of client 1.

Packages Used:

- `java.io.*`
- `java.net.*`

Classes and Methods Used:

- Socket
 - Socket()
 - getInputStream()
 - getOutputStream()
 - close()
- DataInputStream
 - readUTF()
- DataOutputStream
 - writeUTF()
- ServerSocket
 - ServerSocket()
 - accept()
- InetAddress
 - getByName()

Program:

```
package chatserverclientby42;
```

```
import java.net.*;
import java.io.*;
import java.lang.Runnable;
```

```
public class ChatClientServer extends javax.swing.JFrame implements
Runnable{
    private void StartButtonActionPerformed(java.awt.event.ActionEvent
    evt) {
        // TODO add your handling code here:
        Thread t = new Thread(this, "Server");
        t.start();
        StartButton.setEnabled(false);
        MessagePane.append("ServerListening\n");
    }
}
```

```
@Override
    public void run(){
        String ip = ServerIPField.getText();
        int port = Integer.parseInt(ServerPortField.getText());
        try{
            ServerSocket ss = new ServerSocket(port, 5,
            InetAddress.getByName(ip));

            while(true){
                Socket s = ss.accept();
                DataInputStream dis = new DataInputStream(s.getInputStream());
                DataOutputStream dos = new
                DataOutputStream(s.getOutputStream());
```

```

        String receivedMessage = dis.readUTF();

        MessagePane.append("Received: " +receivedMessage + "\n");
        ServerResponseArea.append(s.getInetAddress()+":
"+receivedMessage+"\n");

        s.close();
    }

    }catch(Exception e){
        e.printStackTrace();
    }

    }

    private void SendButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        String ip = ClientIPField.getText();
        int port = Integer.parseInt(ClientPortField.getText());

        try{

            Socket s = new Socket(InetAddress.getByName(ip), port);
            DataInputStream dis = new DataInputStream(s.getInputStream());
            DataOutputStream dos = new
DataOutputStream(s.getOutputStream());
            while(true){
                String msg = MessageField.getText();
                dos.writeUTF(msg);
                String response = dis.readUTF();
                ServerResponseArea.append("You: " + msg + "\n");
                if(msg.equals("/Exit")){
                    s.close();
                    break;
                }
            }

        } catch(Exception e){
            e.printStackTrace();
        }
    }
}

```

Output Images:

The image displays two side-by-side screenshots of a web-based 'Chat Application' interface. Both windows have a light blue background and a title bar with standard window controls.

Left Window:

- IP Address:** localhost
- Port Number:** 5085
- Message:** (Empty text input field)
- SEND:** (Button)
- SERVER:** (Button)
- Port Number:** 5086
- Response:** Server listening...
[me] : message sent to 5085
/127.0.0.1 : message sent to 5086

Right Window:

- IP Address:** localhost
- Port Number:** 5086
- Message:** (Empty text input field)
- SEND:** (Button)
- SERVER:** (Button)
- Port Number:** 5085
- Response:** Server listening...
/127.0.0.1 : message sent to 5085
[me] : message sent to 5086

PROGRAM - 8 (a)

Program Statement:

Implement a RPC program for Echo Service.

Description of the program:

RPC (Remote Procedure Call) allows programs to call procedures located on other machines. When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing at all is visible to the programmer. This method is known as Remote Procedure Call, or often just RPC. The steps of RPC are,

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's as sends the message to the remote as.
4. The remote as gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local as.
8. The server's as sends the message to the client's as.
9. The client's as gives the message to the client stub.
10. The stub unpacks the result and returns to the client

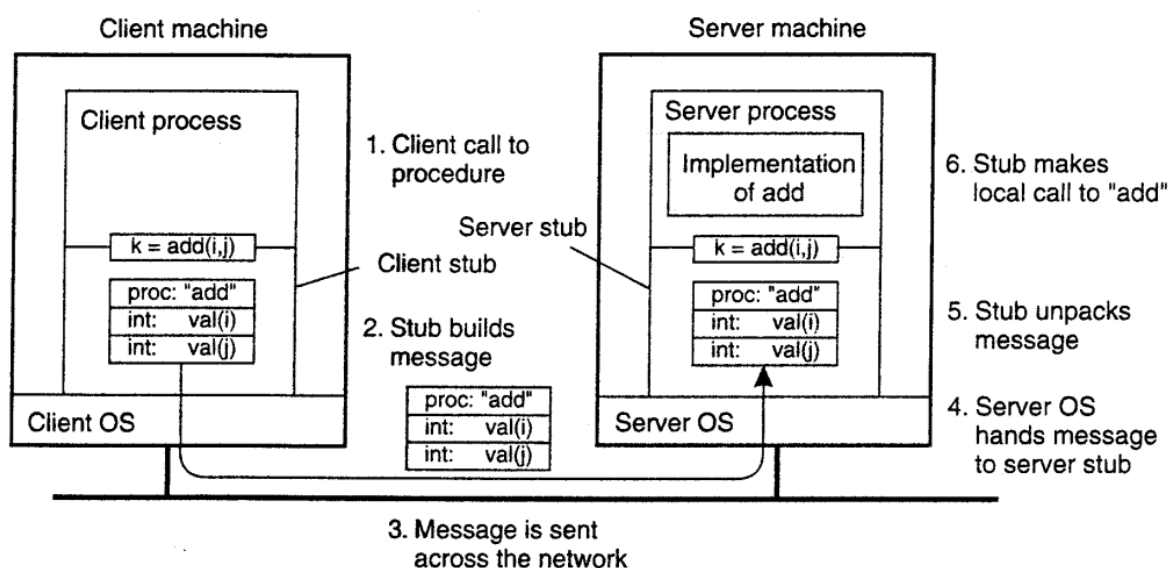


Figure 4-7. The steps involved in a doing a remote computation through RPC.

This is a program in c which uses Sun-RPC (Remote Procedure Call) library to implement Echo Service. Here the client sends a message by making a function call to a remote procedure present on the server. As an Echo Service, the server sends back the same message as a response.

Input:

A message to send to the server.

Output:

The message sent to server is sent back as response.

Program:

Specification file

```
program echo_prg{
version echo_ver{
    string echo(string)=1;
    }=1;
}=0x20000021;
```

Client

```
#include "echo.h"
void
echo_prg_1(char *host)
{
    CLIENT *clnt;
    char * *result_1;
    char * echo_1_arg;

#ifdef    DEBUG
    clnt = clnt_create (host, echo_prg, echo_ver, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif    /* DEBUG */
    echo_1_arg=(char*)malloc(sizeof(char)*25);
    printf("\nEnter a message:");
    scanf("%s",echo_1_arg);
    *result_1=(char*)malloc(sizeof(char)*25);
    result_1 = echo_1(&echo_1_arg, clnt);
    if (result_1 == (char **) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("result=%s",*result_1);
#ifdef    DEBUG
    clnt_destroy (clnt);
#endif    /* DEBUG */
}
```



```

int
main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    echo_prg_1 (host);
    exit (0);
}

```

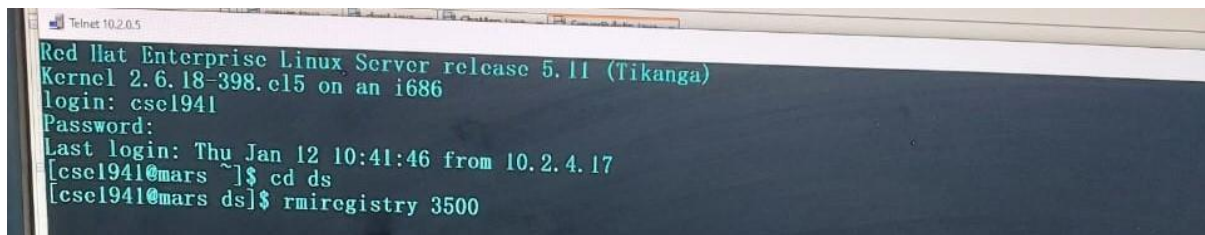
Server

```

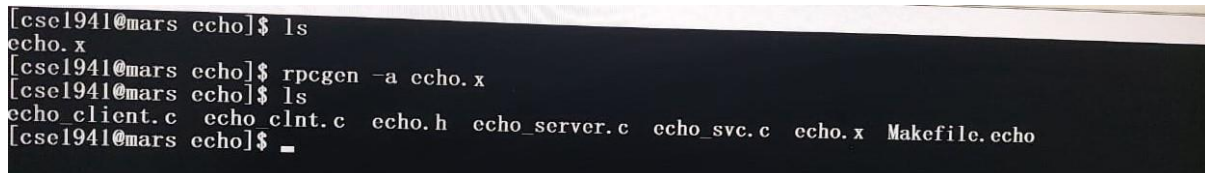
#include "echo.h"
#include<string.h>
char **
echo_1_svc(char **argp, struct svc_req *rqstp)
{
    static char * result;
    result=(char*)malloc(sizeof(char)*25);
    strcpy(result,*argp);
    return &result;
}

```

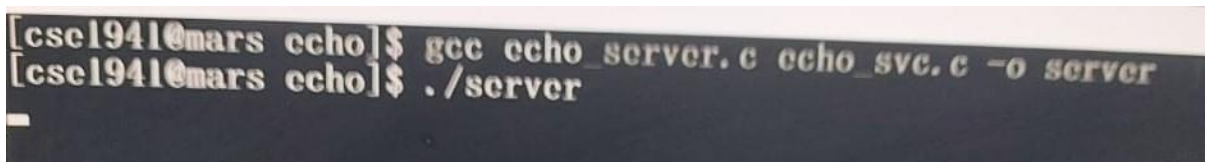
Output Images:



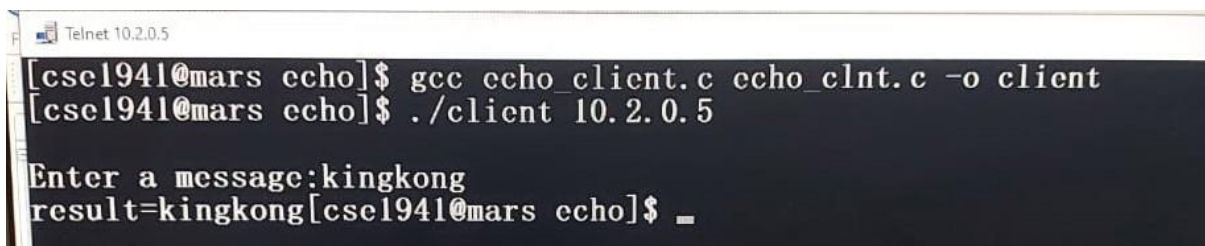
```
Telnet 10.2.0.5
Red Hat Enterprise Linux Server release 5.11 (Tikanga)
Kernel 2.6.18-398.el5 on an i686
login: cse1941
Password:
Last login: Thu Jan 12 10:41:46 from 10.2.4.17
[cse1941@mars ~]$ cd ds
[cse1941@mars ds]$ rmiregistry 3500
```



```
[cse1941@mars echo]$ ls
echo.x
[cse1941@mars echo]$ rpcgen -a echo.x
[cse1941@mars echo]$ ls
echo_client.c  echo_clnt.c  echo.h  echo_server.c  echo_svc.c  echo.x  Makefile.echo
[cse1941@mars echo]$ _
```



```
[cse1941@mars echo]$ gcc echo_server.c echo_svc.c -o server
[cse1941@mars echo]$ ./server
_
```



```
Telnet 10.2.0.5
[cse1941@mars echo]$ gcc echo_client.c echo_clnt.c -o client
[cse1941@mars echo]$ ./client 10.2.0.5

Enter a message:kingkong
result=kingkong[cse1941@mars echo]$ _
```

PROGRAM - 8 (b)

Program Statement:

Implement a RPC program for Reversing a given String.

Description of the program:

This is a program in c which uses Sun-RPC (Remote Procedure Call) library to implement Reverse of a Given String. Here the client sends a message by making a function call to a Remote Procedure present on the server. The server sends back the Reverse of the String as a Response.

Input:

A message to send to the server.

Output:

The message sent to server is reversed and sent back as response.

Program:

Specification file

```
program rev_prog{
version rev_ver{
    string revstr(string)=1;
    }=1;
}=0x20000022;
```

Client

```
#include "rev.h"
void
rev_prog_1(char *host)
{
    CLIENT *clnt;
    char * *result_1;
    char * revstr_1_arg;

#ifdef    DEBUG
    clnt = clnt_create (host, rev_prog, rev_ver, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif    /* DEBUG */
    revstr_1_arg=(char*)malloc(sizeof(char)*20);
    printf("\nEnter a string:");
    scanf("%s",revstr_1_arg);
    *result_1=(char*)malloc(sizeof(char)*20);
    result_1 = revstr_1(&revstr_1_arg, clnt);
```

```

        if (result_1 == (char **) NULL) {
            clnt_perror (clnt, "call failed");
        }
        printf("result=%s",*result_1);
#ifdef      DEBUG
        clnt_destroy (clnt);
#endif      /* DEBUG */
    }
    int
    main (int argc, char *argv[])
    {
        char *host;

        if (argc < 2) {
            printf ("usage: %s server_host\n", argv[0]);
            exit (1);
        }
        host = argv[1];
        rev_prog_1 (host);
    exit (0);
    }

```

Server

```

#include "rev.h"
#include<string.h>
#include<stdlib.h>
char **
revstr_1_svc(char **argp, struct svc_req *rqstp)
{
    static char * result;
    int i,j;
    result=(char*)malloc(sizeof(char)*20);
    for(j=0,i=strlen(*argp)-1;i>=0;i--,j++)
        *(result+j)=*(*argp+i);
    *(result+j)='\0';
    return &result;
}

```

Output Images:

```
Telnet 10.2.0.5
Red Hat Enterprise Linux Server release 5.11 (Tikanga)
Kernel 2.6.18-398.el5 on an i686
login: cse1941
Password:
Last login: Thu Jan 12 10:41:46 from 10.2.4.17
[cse1941@mars ~]$ cd ds
[cse1941@mars ds]$ rmiregistry 3500
```

```
Telnet 10.2.0.5
[cse1941@mars rev]$ ls
rev.x
[cse1941@mars rev]$ rpcgen -a rev.x
[cse1941@mars rev]$ ls
Makefile.rev rev_client.c rev_clnt.c rev.h rev_server.c rev_svc.c rev.x
[cse1941@mars rev]$
```

```
Telnet 10.2.0.5
[cse1941@mars rev]$ gcc rev_client.c rev_clnt.c -o client
[cse1941@mars rev]$ ./client 10.2.0.5

Enter a string:kingkong
result=gnokgnik[cse1941@mars rev]$
```

```
Telnet 10.2.0.5
[cse1941@mars rev]$ gcc rev_server.c rev_svc.c -o server
[cse1941@mars rev]$ ./server
```

PROGRAM - 8 (c)

Program Statement:

Implement a RPC program for Concatenating two given Strings.

Description of the program:

This is a program in c which uses Sun-RPC (Remote Procedure Call) library to implement Concatenation of Two Given Strings. Here the client sends two strings by making a function call to a Remote Procedure present on the server. The server sends back the Concatenated String as a response. Concatenation means joining the end of one String to start of the other String.

Input:

Two strings to be concatenated.

Output:

Concatenated String is sent back as response.

Program:

Specification file

```
struct str{
    string str1<>;
    string str2<>;
};
program concat_prog{
version concat_ver{
    string concat(strs)=1;
    }=1;
}=0x20000201;
```

Client

```
#include "concat.h"
void
concat_prog_1(char *host)
{
    CLIENT *clnt;
    char * *result_1;
    strs concat_1_arg;

#ifdef    DEBUG
    clnt = clnt_create (host, concat_prog, concat_ver, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif    /* DEBUG */
```

```

        concat_1_arg.str1=(char*)malloc(sizeof(char)*32);
        concat_1_arg.str2=(char*)malloc(sizeof(char)*32);
        printf("\nEnter two string to concatenate:");
        scanf("%s%s",concat_1_arg.str1,concat_1_arg.str2);
        *result_1=(char*)malloc(sizeof(char)*100);
        result_1 = concat_1(&concat_1_arg, clnt);
        if (result_1 == (char **) NULL) {
            clnt_perror (clnt, "call failed");
        }
        printf("\nresult=%s",*result_1);
#ifdef      DEBUG
        clnt_destroy (clnt);
#endif      /* DEBUG */
    }

int
main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    concat_prog_1 (host);
    exit (0);
}

```

Server

```

#include "concat.h"
#include<string.h>
char **
concat_1_svc(strs *argp, struct svc_req *rqstp)
{
    static char * result;
    result=(char*)malloc(sizeof(char)*100);
    strcpy(result,argp->str1);
    strcat(result,argp->str2);
    return &result;
}

```

Output Images:

```
Telnet 10.2.0.5
Red Hat Enterprise Linux Server release 5.11 (Tikanga)
Kernel 2.6.18-398.el5 on an i686
login: cse1941
Password:
Last login: Thu Jan 12 10:41:46 from 10.2.4.17
[cse1941@mars ~]$ cd ds
[cse1941@mars ds]$ rmiregistry 3500
```

```
Telnet 10.2.0.5
[cse1941@mars concat]$ ls
concat.x
[cse1941@mars concat]$ rpcgen -a concat.x
[cse1941@mars concat]$ ls
concat_client.c concat_clnt.c concat.h concat_server.c concat_svc.c concat.x concat_xdr.c Makefile.concat
[cse1941@mars concat]$
```

```
Telnet 10.2.0.5
[cse1941@mars concat]$ gcc concat_server.c concat_svc.c concat_xdr.c -o server
[cse1941@mars concat]$ ./server
```

```
Telnet 10.2.0.5
[cse1941@mars concat]$ gcc concat_client.c concat_clnt.c concat_xdr.c -o client
[cse1941@mars concat]$ ./client 10.2.0.5

Enter two string to concatenate:king
kong

result=kingkong[cse1941@mars concat]$
```


PROGRAM - 8 (d)

Program Statement:

Implement a RPC program to find length of a given String.

Description of the program:

This is a program in c which uses Sun-RPC (Remote Procedure Call) library to implement Length of String Function. Here the client sends a message by making a function call to a Remote Procedure present on the server. The server sends back the Length of the String as a response.

Input:

A message to send to the server.

Output:

The length of the message is sent back as response.

Program:

Specification file

```
program strl_prog{
version strl_ver{
    int strl(string)=1;
    }=1;
}=0x20000022;
```

Client

```
#include "strl.h"
void
strl_prog_1(char *host)
{
    CLIENT *clnt;
    int *result_1;
    char * strl_1_arg;

#ifdef    DEBUG
    clnt = clnt_create (host, strl_prog, strl_ver, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif    /* DEBUG */
    strl_1_arg=(char*)malloc(sizeof(char)*40);
    printf("\nEnter a string:");
    scanf("%s",strl_1_arg);
    printf("\n input string is %s\n",strl_1_arg);
    result_1=(int*)malloc(sizeof(int));
    result_1 = strl_1(&strl_1_arg, clnt);
```

```

        if (result_1 == (int *) NULL) {
            clnt_perror (clnt, "call failed");
        }
        printf("length of %s is %d",strl_1_arg,*result_1);
#ifdef      DEBUG
        clnt_destroy (clnt);
#endif      /* DEBUG */
    }
    int
    main (int argc, char *argv[])
    {
        char *host;

        if (argc < 2) {
            printf ("usage: %s server_host\n", argv[0]);
            exit (1);
        }
        host = argv[1];
        strl_prog_1 (host);
    exit (0);
    }

```

Server

```

#include "strl.h"
#include<string.h>
int *
strl_1_svc(char **argp, struct svc_req *rqstp)
{
    static int  result;
    result=strlen(*argp);
    return &result;
}

```

Output Images:

```
Telnet 10.2.0.5
Red Hat Enterprise Linux Server release 5.11 (Tikanga)
Kernel 2.6.18-398.el5 on an i686
login: cse1941
Password:
Last login: Thu Jan 12 10:41:46 from 10.2.4.17
[cse1941@mars ~]$ cd ds
[cse1941@mars ds]$ rmiregistry 3500
```

```
Telnet 10.2.0.5
[cse1941@mars strlength]$ ls
strl.x
[cse1941@mars strlength]$ rpcgen -a strl.x
[cse1941@mars strlength]$ ls
Makefile.strl strl_client.c strl_clnt.c strl.h strl_server.c strl_svc.c strl.x
[cse1941@mars strlength]$
```

```
Telnet 10.2.0.5
[cse1941@mars strlength]$ gcc strl_client.c strl_clnt.c -o client
[cse1941@mars strlength]$ ./client 10.2.0.5

Enter a string:reverse

input string is reverse
length of reverse is 7[cse1941@mars strlength]$ _
```

```
Telnet 10.2.0.5
[cse1941@mars strlength]$ gcc strl_server.c strl_svc.c -o server
[cse1941@mars strlength]$ ./server
```

PROGRAM - 8 (e)

Program Statement:

Implement a RPC program to find GCD of two given numbers.

Description of the program:

This is a program in c which uses Sun-RPC (Remote Procedure Call) library to implement GCD (Greatest Common Divisor) of Two Numbers. Here the client sends two numbers by making a function call to a Remote Procedure present on the server. The server sends back GCD of the two numbers as a response.

Input:

Two numbers whose GCD has to be computed.

Output:

The GCD of the two numbers is sent back as response.

Program:

Specification file

```
struct num{
    int a;
    int b;
};
program gcd_prog{
    version gcd_ver{
        int gcd(num)=1;
    }=1;
}=0x20000021;
```

Client

```
#include "gcd.h"
void
gcd_prog_1(char *host)
{
    CLIENT *clnt;
    int *result_1;
    num gcd_1_arg;

#ifdef    DEBUG
    clnt = clnt_create (host, gcd_prog, gcd_ver, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif    /* DEBUG */
    printf("\nEnter two numbers:");
```

```

scanf("%d%d",&gcd_1_arg.a,&gcd_1_arg.b);
result_1 = gcd_1(&gcd_1_arg, clnt);
if (result_1 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
}

printf("\ngcd of %d and %d is
%d",gcd_1_arg.a,gcd_1_arg.b,*result_1);

#ifdef      DEBUG
    clnt_destroy (clnt);
#endif      /* DEBUG */
}

int
main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    gcd_prog_1 (host);
exit (0);
}

```

Server

```

#include "gcd.h"

int *
gcd_1_svc(num *argp, struct svc_req *rqstp)
{
    static int  result;
    int temp;
    while(argp->a!=0)
    {
        temp=argp->a;
        argp->a=argp->b%argp->a;
        argp->b=temp;
    }
    result=argp->b;
    return &result;
}

```

Output Images:

```
Telnet 10.2.0.5
Red Hat Enterprise Linux Server release 5.11 (Tikanga)
Kernel 2.6.18-398.el5 on an i686
login: csel941
Password:
Last login: Thu Jan 12 10:41:46 from 10.2.4.17
[csel941@mars ~]$ cd ds
[csel941@mars ds]$ rmiregistry 3500
```

```
Telnet 10.2.0.5
[csel941@mars gcd]$ ls
gcd.x
[csel941@mars gcd]$ rpcgen -a gcd.x
[csel941@mars gcd]$ ls
gcd_client.c  gcd_clnt.c  gcd.h  gcd_server.c  gcd_svc.c  gcd.x  gcd_xdr.c  Makefile.gcd
[csel941@mars gcd]$
```

```
Telnet 10.2.0.5
[csel941@mars gcd]$ gcc gcd_server.c gcd_svc.c gcd_xdr.c -o server
[csel941@mars gcd]$ ./server
```

```
Telnet 10.2.0.5
[csel941@mars gcd]$ gcc gcd_client.c gcd_clnt.c gcd_xdr.c -o client
[csel941@mars gcd]$ ./client 10.2.0.5

Enter two numbers:65
18

gcd of 65 and 18 is 1[csel941@mars gcd]$
```

PROGRAM - 8 (f)

Program Statement:

Implement a RPC program to find the sum of two given numbers.

Description of the program:

This is a program in c which uses Sun-RPC (Remote Procedure Call) library to implement Sum of Two Numbers. Here the client sends two numbers by making a function call to a Remote Procedure present on the server. The server sends back the Sum of Two Numbers as a response.

Input:

Two numbers whose sum has to be computed.

Output:

The sum of the two numbers is sent back as response.

Program:

Specification file

```
struct num{
    int a;
    int b;
};
program add_prog{
version add_ver{
    int add(num)=1;
    }=1;
}=0x20000022;
```

Client

```
#include "add.h"
void
add_prog_1(char *host)
{
    CLIENT *clnt;
    int *result_1;
    num add_1_arg;

#ifdef    DEBUG
    clnt = clnt_create (host, add_prog, add_ver, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif    /* DEBUG */
    printf("\nEnter two numbers:");
    scanf("%d%d",&add_1_arg.a,&add_1_arg.b);
```

```

        result_1=(int*)malloc(sizeof(int));
        result_1 = add_1(&add_1_arg, clnt);
        if (result_1 == (int *) NULL) {
            clnt_perror (clnt, "call failed");
        }
        printf("\n %d + %d = %d",add_1_arg.a,add_1_arg.b,*result_1);
#ifdef      DEBUG
        clnt_destroy (clnt);
#endif      /* DEBUG */
    }

```

```

int
main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    add_prog_1 (host);
    exit (0);
}

```

Server

```

#include "add.h"

int *
add_1_svc(num *argp, struct svc_req *rqstp)
{
    static int  result;
    result=argp->a+argp->b;
    return &result;
}

```


Output Images:

```
Telnet 10.2.0.5
Red Hat Enterprise Linux Server release 5.11 (Tikanga)
Kernel 2.6.18-398.el5 on an i686
login: csel1941
Password:
Last login: Thu Jan 12 10:41:46 from 10.2.4.17
[csel1941@mars ~]$ cd ds
[csel1941@mars ds]$ rmiregistry 3500
```

```
Telnet 10.2.0.5
[csel1941@mars add1]$ ls
add.x
[csel1941@mars add1]$ rpcgen -a add.x
[csel1941@mars add1]$ ls
add_client.c add_clnt.c add.h add_server.c add_svc.c add.x add_xdr.c Makefile.add
[csel1941@mars add1]$
```

```
Telnet 10.2.0.5
Red Hat Enterprise Linux Server release 5.11 (Tikanga)
Kernel 2.6.18-398.el5 on an i686
login: csel1941
Password:
Last login: Thu Jan 12 10:23:16 from 10.2.4.17
[csel1941@mars ~]$ cd ds
[csel1941@mars ds]$ cd rpc
[csel1941@mars rpc]$ cd add2
[csel1941@mars add2]$ ls
add_client.c add_clnt.c add.h add_server.c add_svc.c add.x add_xdr.c client Makefile.add server
[csel1941@mars add2]$ ./server
[csel1941@mars add2]$ ./server 10.2.0.5
[csel1941@mars add2]$ ./server
```

```
Telnet 10.2.0.5
[csel1941@mars add2]$ gcc add_client.c add_clnt.c add_xdr.c -o client
[csel1941@mars add2]$ gcc add_server.c add_svc.c add_xdr.c -o server
[csel1941@mars add2]$ ./client
usage: ./client server host
[csel1941@mars add2]$ ./client 10.2.0.5
Enter two numbers:3
4
3 + 4 = 7[csel1941@mars add2]$ ./client 10.2.0.5
Enter two numbers:6
7
6 + 7 = 13[csel1941@mars add2]$
```

PROGRAM - 9 (a)

Program Statement:

Implement a RMI program for Echo Service

Description of the program:

This program is a simple Remote Method Invocation (RMI) Application that allows a client to send a message to a server and receive the same message as a response identical to an Echo Service. The application consists of three parts: an interface, an implementation class, and a server and client class.

The EchoInterface interface defines a single Remote Method, echo(String s), which takes in a string as a parameter and returns the same string.

The EchoImpl class is the Implementation of the EchoInterface. The class extends UnicastRemoteObject, which is necessary for objects that are exported to use RMI. The echo method in this class takes in a string as a parameter, prints it out, and returns the same string.

The EchoServer class is the Server-Side Component of the Application. The main method of this class creates an instance of EchoImpl, creates an RMI registry on a specified port, and binds the EchoImpl object to a location in the registry.

The EchoClient class is the Client-Side Component of the Application. The main method of this class looks up the EchoImpl object in the RMI registry using the specified location, prompts the user for input and sends it to the echo method of the EchoImpl object, and prints out the response received from the server.

To run the application, the server must be started first, followed by the client. The server and client both take in three command-line arguments: the IP address of the server, the port number of the RMI registry, and the name to bind the EchoImpl object to in the registry.

Input:

A message to send to the server.

Output:

The message sent to server is sent back as response.

Packages Used:

- java.rmi.*
- java.io.*
- java.net.*
- java.rmi.server.*

Classes and Methods Used:

- LocateRegistry
 - createRegistry()
- Naming
 - bind()
 - lookup()
- Remote
- RemoteException
- MalformedURLException
- UnicastRemoteObject

Program:

Interface:

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

public interface EchoInterface extends Remote{
    String echo(String s) throws RemoteException;
}
```

Implementation:

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

public class EchoImpl extends UnicastRemoteObject implements
EchoInterface{
    public EchoImpl() throws RemoteException{
    }
    public String echo(String s1)
    {
        System.out.println("REMOTE SERVICE: Remote Client Request Message
is : "+s1);
        return s1;
    }
}
```

Server:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.Naming;
import java.net.MalformedURLException;
import java.rmi.registry.LocateRegistry;

public class EchoServer{
```

```

public EchoServer() throws RemoteException{
}
public static void main(String args[]) throws RemoteException
{
    EchoImpl echoObj=new EchoImpl();
    int port=Integer.parseInt(args[1]);
    try{
        LocateRegistry.createRegistry(port);
        System.out.println("\n RMI registry created \n");
        String host=args[0];
        String bindLocation="//"+host+": "+port+"/"+args[2];
        Naming.bind(bindLocation,echoObj);
        System.out.println("\nRMI server ready at "+bindLocation);
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
}

```

Client:

```

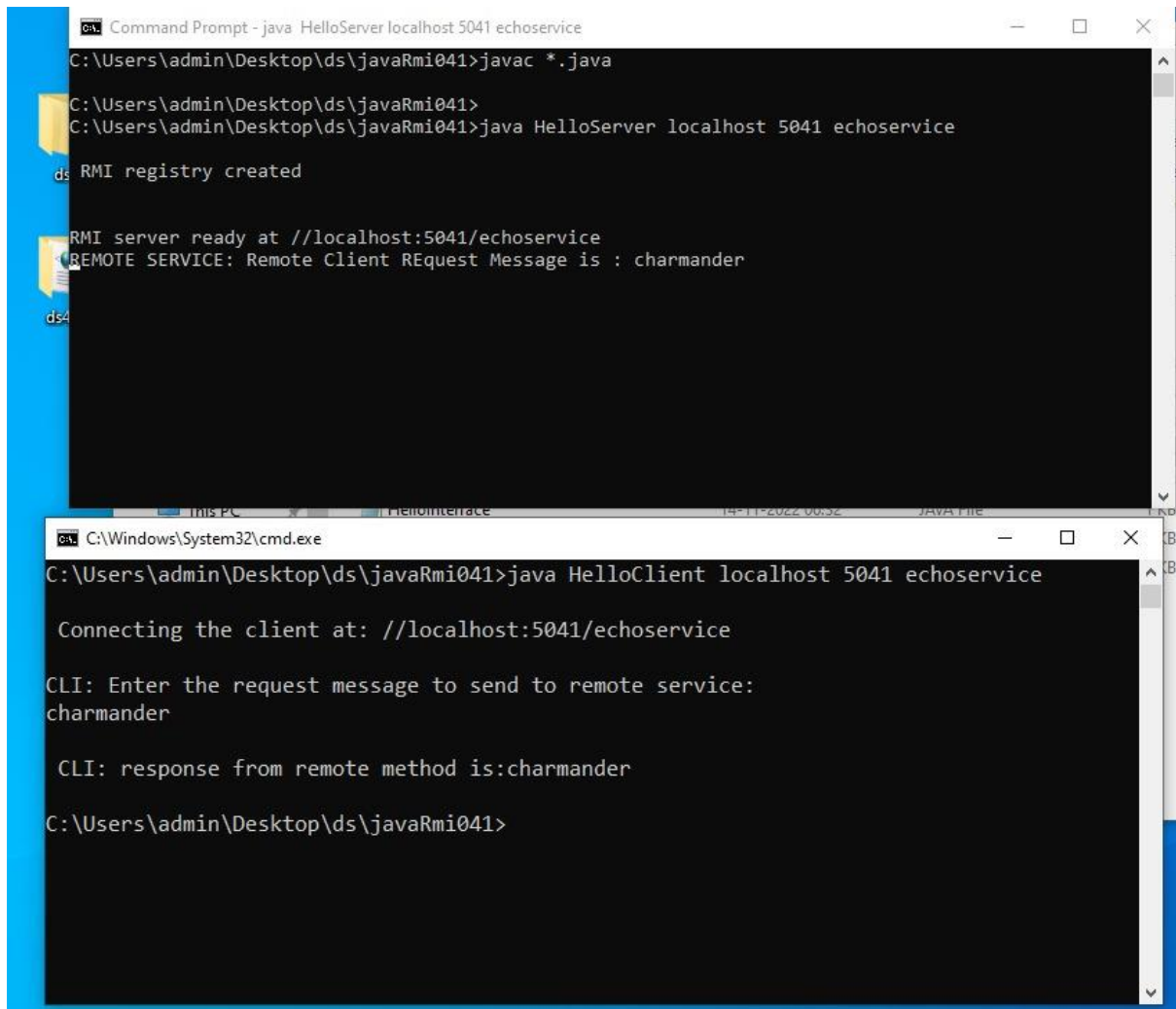
import java.rmi.*;
import java.io.*;
import java.net.MalformedURLException;

public class EchoClient {
    public static void main(String args[])
    {
        String
connectLocation="//"+args[0]+": "+Integer.parseInt(args[1])+"/"+args[2];
        EchoInterface eintf=null;
        try{
            System.out.println("\n Connecting the client at:
"+connectLocation);
            eintf=(EchoInterface)Naming.lookup(connectLocation);
            BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
            System.out.println("\nCLI: Enter the request message to send
to remote service:");
            String s=br.readLine();
            String response=eintf.echo(s);
            System.out.println("\n CLI: response from remote method
is:"+response);
        }
        catch(Exception e)

```

```
    {  
        e.printStackTrace();  
    }  
}  
}
```

Output Images:



```
Command Prompt - java HelloServer localhost 5041 echoservice  
C:\Users\admin\Desktop\ds\javaRmi041>javac *.java  
C:\Users\admin\Desktop\ds\javaRmi041>  
C:\Users\admin\Desktop\ds\javaRmi041>java HelloServer localhost 5041 echoservice  
ds RMI registry created  
RMI server ready at //localhost:5041/echoservice  
REMOTE SERVICE: Remote Client REquest Message is : charmander  
ds4  
C:\Windows\System32\cmd.exe  
C:\Users\admin\Desktop\ds\javaRmi041>java HelloClient localhost 5041 echoservice  
Connecting the client at: //localhost:5041/echoservice  
CLI: Enter the request message to send to remote service:  
charmander  
CLI: response from remote method is:charmander  
C:\Users\admin\Desktop\ds\javaRmi041>
```

PROGRAM - 9 (b)

Program Statement:

Implement a RMI program for Reversing a given String.

Description of the program:

The program consists of an interface, an implementation class, and a server and client class.

The ReverseInterface interface defines a Single Remote method, reverseStr(String s), which takes in a string as a parameter and returns the Reverse of the String.

The ReverseImpl class is the implementation of the ReverseInterface. The class extends UnicastRemoteObject, which is necessary for objects that are exported to use RMI. The reverseStr method in this class takes in a string as a parameter, creates a StringBuilder object with the input string, reverses the string using the StringBuilder's reverse() method, and returns the reversed string.

The ReverseServer class is the Server-Side Component of the Application. The main method of this class creates an instance of ReverseImpl, creates an RMI registry on the specified port, binds the object to the registry, and makes it available for remote clients to access.

The ReverseClient class is the Client-Side Component of the Application. The main method of this class looks up the remote object on the specified IP address, port and service name, prompts the user to enter a string, calls the reverseStr method on the remote object, and prints the response.

The client and server communicate using RMI, which allows the client to invoke methods on the remote object as if they were local methods. The client sends the string to the server, the server reverses the string and sends the reversed string back to the client, which then prints it.

Input:

A message to send to the server.

Output:

The message sent to server is reversed and sent back as response.

Packages Used:

- java.rmi.*
- java.io.*
- java.net.*
- java.rmi.server.*

Classes and Methods Used:

- LocateRegistry
 - createRegistry()
- Naming
 - bind()
 - lookup()
- Remote
- RemoteException
- MalformedURLException
- UnicastRemoteObject

Program:

Interface:

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

public interface ReverseInterface extends Remote{
    String reverseStr(String s) throws RemoteException;
}
```

Implementation:

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

public class ReverseImpl extends UnicastRemoteObject implements
ReverseInterface{
    public ReverseImpl() throws RemoteException{
    }
    public String reverseStr(String s1)
    {
        System.out.println("REMOTE SERVICE: Remote Client Request Message
is : "+s1);
        StringBuilder sb=new StringBuilder(s1);
        String response=sb.reverse().toString();
        return response;
    }
}
```

Server:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.Naming;
import java.net.MalformedURLException;
import java.rmi.registry.LocateRegistry;
```

```

public class ReverseServer{
    public ReverseServer() throws RemoteException{
    }
    public static void main(String args[]) throws RemoteException
    {
        ReverseImpl reverseObj=new ReverseImpl();
        int port=Integer.parseInt(args[1]);
        try{
            LocateRegistry.createRegistry(port);
            System.out.println("\n RMI registry created \n");
            String host=args[0];
            String bindLocation="//"+host+": "+port+"/"+args[2];
            Naming.bind(bindLocation,reverseObj);
            System.out.println("\nRMI server ready at "+bindLocation);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

Client:

```

import java.rmi.*;
import java.io.*;
import java.net.MalformedURLException;

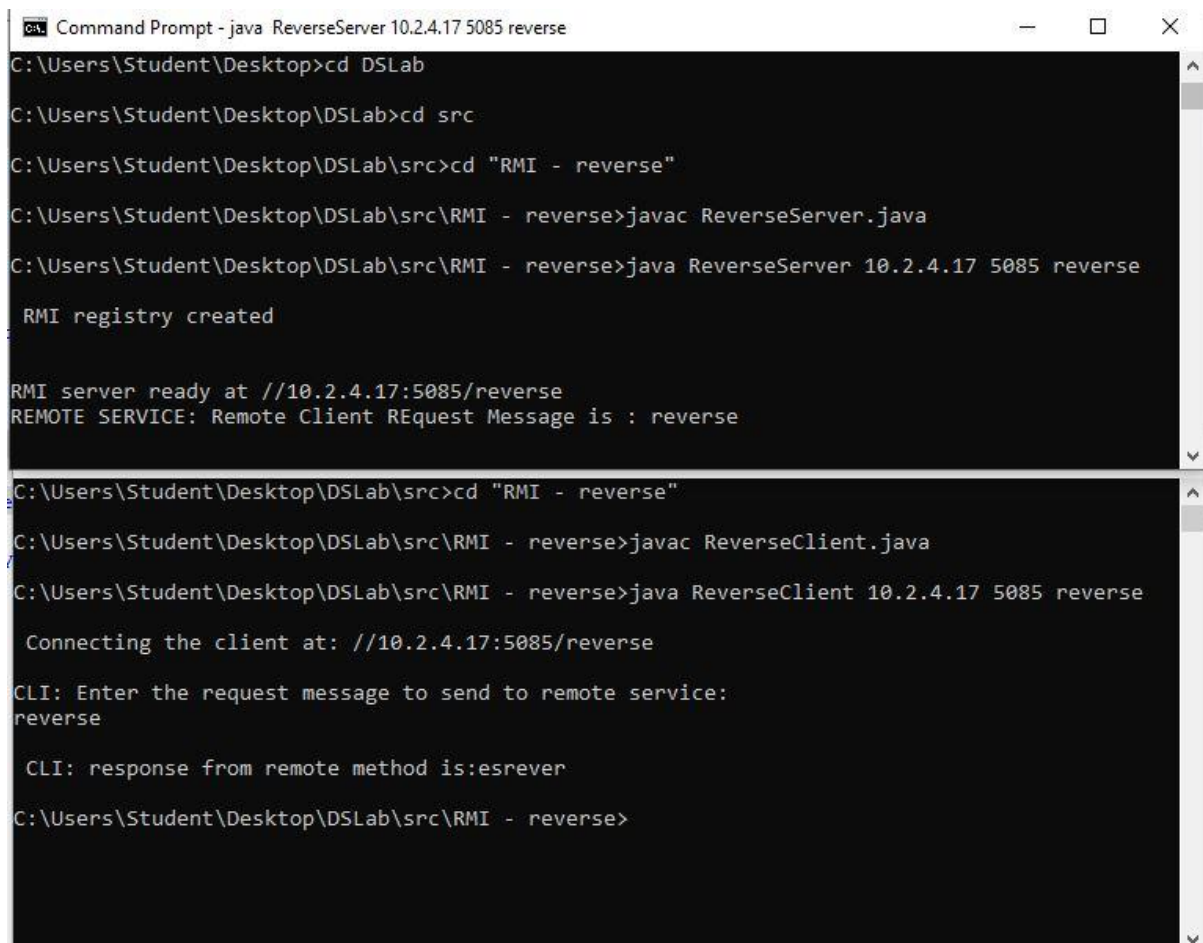
public class ReverseClient {
    public static void main(String args[])
    {
        String
connectLocation="//"+args[0]+": "+Integer.parseInt(args[1])+"/"+args[2];
        ReverseInterface revintf=null;
        try{
            System.out.println("\n Connecting the client at:
"+connectLocation);
            revintf=(ReverseInterface)Naming.lookup(connectLocation);
            BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
            System.out.println("\nCLI: Enter the request message to send
to remote service:");
            String s=br.readLine();
            String response=revintf.reverseStr(s);
            System.out.println("\n CLI: response from remote method
is:"+response);
        }
        catch(Exception e)

```



```
    {  
        e.printStackTrace();  
    }  
}  
}
```

Output Images:



```
Command Prompt - java ReverseServer 10.2.4.17 5085 reverse  
C:\Users\Student\Desktop>cd DSLab  
C:\Users\Student\Desktop\DSLAb>cd src  
C:\Users\Student\Desktop\DSLAb\src>cd "RMI - reverse"  
C:\Users\Student\Desktop\DSLAb\src\RMI - reverse>javac ReverseServer.java  
C:\Users\Student\Desktop\DSLAb\src\RMI - reverse>java ReverseServer 10.2.4.17 5085 reverse  
  
RMI registry created  
  
RMI server ready at //10.2.4.17:5085/reverse  
REMOTE SERVICE: Remote Client REquest Message is : reverse  
  
C:\Users\Student\Desktop\DSLAb\src>cd "RMI - reverse"  
C:\Users\Student\Desktop\DSLAb\src\RMI - reverse>javac ReverseClient.java  
C:\Users\Student\Desktop\DSLAb\src\RMI - reverse>java ReverseClient 10.2.4.17 5085 reverse  
  
Connecting the client at: //10.2.4.17:5085/reverse  
CLI: Enter the request message to send to remote service:  
reverse  
  
CLI: response from remote method is:esrever  
C:\Users\Student\Desktop\DSLAb\src\RMI - reverse>
```

PROGRAM - 9 (c)

Program Statement:

Implement a RMI program for Concatenating two given Strings.

Description of the program:

The RMI system consists of three main parts: the interface, the implementation, and the server and client.

The interface, ConcatInterface, defines the methods that can be called remotely by the client. In this case, the method concatenate(String s1, String s2) is defined.

The implementation, ConcatImpl, provides the actual implementation of the methods defined in the interface. ConcatImpl extends UnicastRemoteObject, and implements the ConcatInterface.

The server, ConcatServer, creates an instance of the ConcatImpl class and makes it available to the client through the RMI registry. The server also creates the RMI registry on the specified port and binds the ConcatImpl object to the location specified in the command line arguments.

The client, ConcatClient, looks up the ConcatImpl object on the RMI registry and invokes the concatenate method, passing two strings as arguments. It receives the concatenated string as the response from the server.

To run the RMI system, first, the server needs to be started, and then the client. The client connects to the server and invokes the remote method by passing the required arguments. The server send backs the concatenated string, i.e., end of first string joined to the start of the second string.

Input:

Two strings to send to the server.

Output:

The concatenated is sent back as response.

Packages Used:

- java.rmi.*
- java.io.*
- java.net.*
- java.rmi.server.*

Classes and Methods Used:

- LocateRegistry
 - createRegistry()
- Naming
 - bind()
 - lookup()
- Remote
- RemoteException
- MalformedURLException
- UnicastRemoteObject

Program:

Interface:

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

public interface ConcatInterface extends Remote{
    String concatenate(String s1,String s2) throws RemoteException;
}
```

Implementation:

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

public class ConcatImpl extends UnicastRemoteObject implements
ConcatInterface{
    public ConcatImpl() throws RemoteException{
    }
    public String concatenate(String s1,String s2)
    {
        System.out.println("REMOTE SERVICE: Remote Client Request Message
is : "+s1+" and "+s2);
        return s1+s2;
    }
}
```

Server:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.Naming;
import java.net.MalformedURLException;
import java.rmi.registry.LocateRegistry;

public class ConcatServer{
    public ConcatServer() throws RemoteException{
```

```

    }
    public static void main(String args[]) throws RemoteException
    {
        ConcatImpl concatObj=new ConcatImpl();
        int port=Integer.parseInt(args[1]);
        try{
            LocateRegistry.createRegistry(port);
            System.out.println("\n RMI registry created \n");
            String host=args[0];
            String bindLocation="//"+host+": "+port+"/"+args[2];
            Naming.bind(bindLocation,concatObj);
            System.out.println("\nRMI server ready at "+bindLocation);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

Client:

```

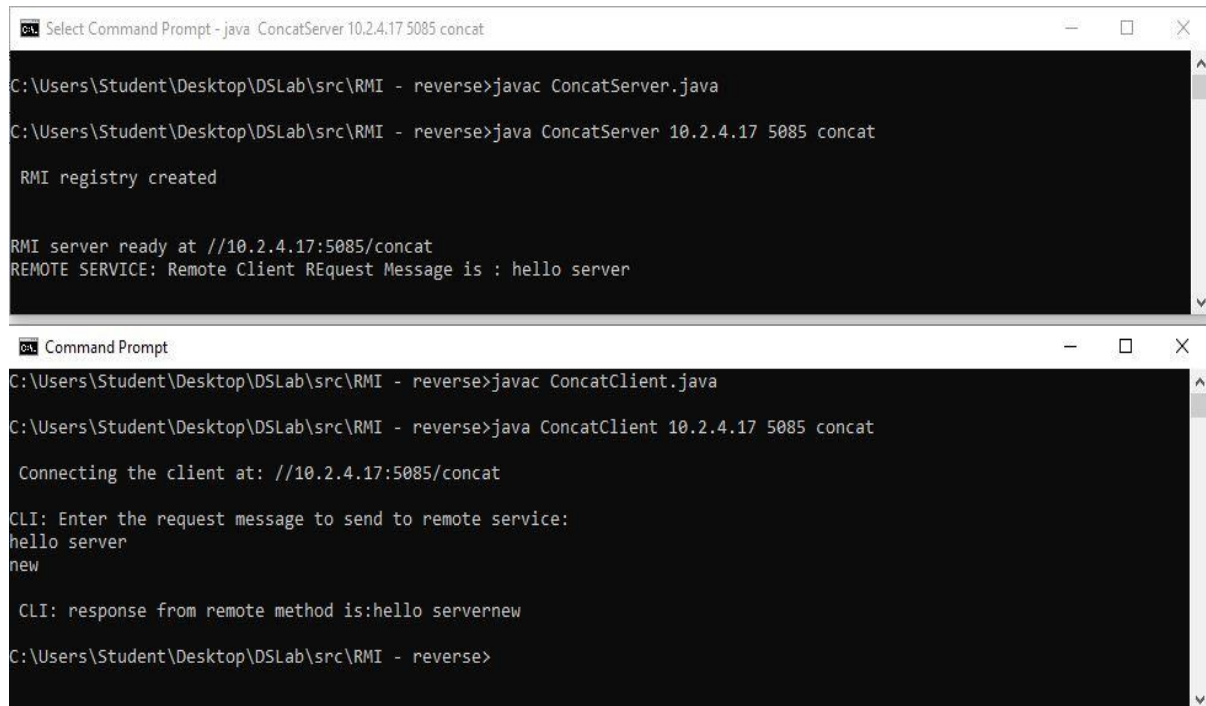
import java.rmi.*;
import java.io.*;
import java.net.MalformedURLException;

public class ConcatClient {
    public static void main(String args[])
    {
        String
connectLocation="//"+args[0]+": "+Integer.parseInt(args[1])+"/"+args[2];
        ConcatInterface cintf=null;
        try{
            System.out.println("\n Connecting the client at:
"+connectLocation);
            cintf=(ConcatInterface)Naming.lookup(connectLocation);
            BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
            System.out.println("\nCLI: Enter two strings to send to remote
service:");
            String s1=br.readLine();
            String s2=br.readLine();
            String response=cintf.concatenate(s1,s2);
            System.out.println("\n CLI: response from remote method
is:"+response);
        }
        catch(Exception e)
        {

```

```
        e.printStackTrace();
    }
}
```

Output Images:



The image contains two screenshots of Windows Command Prompts. The top screenshot shows the execution of a Java RMI server. The bottom screenshot shows the execution of a Java RMI client.

Top Screenshot: Select Command Prompt - java ConcatServer 10.2.4.17 5085 concat

```
C:\Users\Student\Desktop\DSLAb\src\RMI - reverse>javac ConcatServer.java
C:\Users\Student\Desktop\DSLAb\src\RMI - reverse>java ConcatServer 10.2.4.17 5085 concat

RMI registry created

RMI server ready at //10.2.4.17:5085/concat
REMOTE SERVICE: Remote Client REquest Message is : hello server
```

Bottom Screenshot: Command Prompt

```
C:\Users\Student\Desktop\DSLAb\src\RMI - reverse>javac ConcatClient.java
C:\Users\Student\Desktop\DSLAb\src\RMI - reverse>java ConcatClient 10.2.4.17 5085 concat

Connecting the client at: //10.2.4.17:5085/concat
CLI: Enter the request message to send to remote service:
hello server
new

CLI: response from remote method is:hello servernew
C:\Users\Student\Desktop\DSLAb\src\RMI - reverse>
```

PROGRAM - 9 (d)

Program Statement:

Implement a RMI program to find length of a given String.

Description of the program:

This is a Java RMI (Remote Method Invocation) program that consists of an interface, an implementation, a server, and a client. The interface "LengthInterface" defines a method "strLength" that takes a string as an input and returns its length. The class "LengthImpl" implements this interface and provides the implementation for the "strLength" method. The "LengthServer" class creates an instance of "LengthImpl" and starts the RMI registry on a specific port, binds the instance to a specific location and makes it available for remote clients to invoke the "strLength" method.

The "LengthClient" class connects to the RMI server at the specified location, gets the remote reference to the "LengthImpl" object and invokes the "strLength" method by passing a string as an argument. The response returned by the server is the length of the string, which is then printed on the client side.

Input:

A message to send to the server.

Output:

The length of the message is sent back as response.

Packages Used:

- java.rmi.*
- java.io.*
- java.net.*
- java.rmi.server.*

Classes and Methods Used:

- LocateRegistry
 - createRegistry()
- Naming
 - bind()
 - lookup()
- Remote
- RemoteException
- MalformedURLException
- UnicastRemoteObject

Program:

Interface:

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

public interface LengthInterface extends Remote{
    int strLength(String s) throws RemoteException;
}
```

Implementation:

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

public class LengthImpl extends UnicastRemoteObject implements
LengthInterface{
    public LengthImpl() throws RemoteException{
    }
    public int strLength(String s1)
    {
        System.out.println("REMOTE SERVICE: Remote Client Request Message
is : "+s1);
        return s1.length();
    }
}
```

Server:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.Naming;
import java.net.MalformedURLException;
import java.rmi.registry.LocateRegistry;

public class LengthServer{
    public LengthServer() throws RemoteException{
    }
    public static void main(String args[]) throws RemoteException
    {
        LengthImpl lengthObj=new LengthImpl();
        int port=Integer.parseInt(args[1]);
        try{
            LocateRegistry.createRegistry(port);
            System.out.println("\n RMI registry created \n");
            String host=args[0];
            String bindLocation="//"+host+": "+port+"/"+args[2];
            Naming.bind(bindLocation,lengthObj);
            System.out.println("\nRMI server ready at "+bindLocation);
        }
    }
}
```

```

    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
}

```

Client:

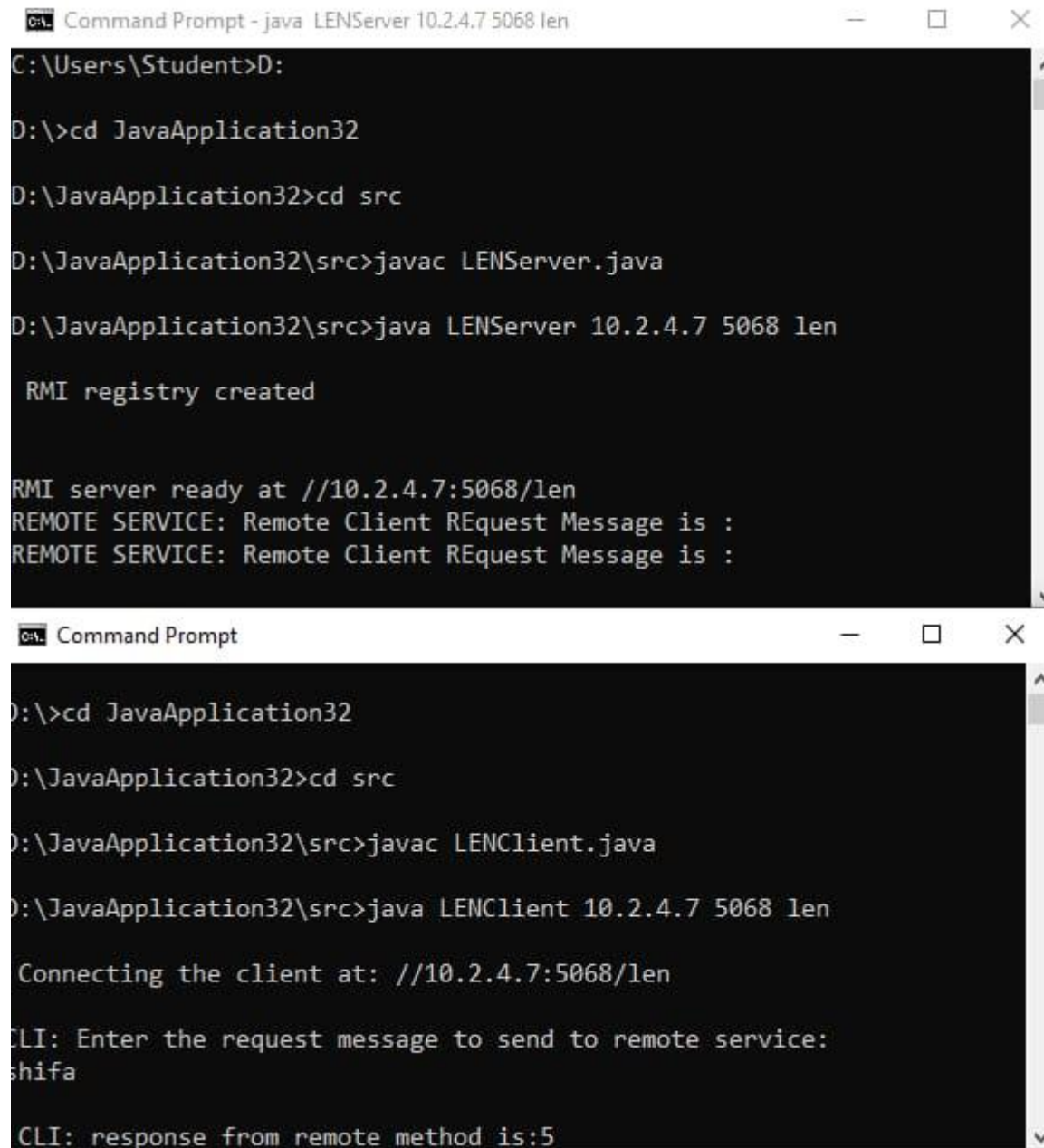
```

import java.rmi.*;
import java.io.*;
import java.net.MalformedURLException;

public class LengthClient {
    public static void main(String args[])
    {
        String
connectLocation="//"+args[0]+":"+Integer.parseInt(args[1])+"/"+args[2];
        LengthInterface lintf=null;
        try{
            System.out.println("\n Connecting the client at:
"+connectLocation);
            lintf=(LengthInterface)Naming.lookup(connectLocation);
            BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
            System.out.println("\nCLI: Enter the request message to send
to remote service:");
            String s=br.readLine();
            int response=lintf.strLength(s);
            System.out.println("\n CLI: response from remote method
is:"+response);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

```


Output Images:



The image contains two screenshots of Windows Command Prompts. The top screenshot shows the execution of a Java RMI server. The bottom screenshot shows the execution of a Java RMI client.

```
Command Prompt - java LENServer 10.2.4.7 5068 len
C:\Users\Student>D:
D:\>cd JavaApplication32
D:\JavaApplication32>cd src
D:\JavaApplication32\src>javac LENServer.java
D:\JavaApplication32\src>java LENServer 10.2.4.7 5068 len

RMI registry created

RMI server ready at //10.2.4.7:5068/len
REMOTE SERVICE: Remote Client RRequest Message is :
REMOTE SERVICE: Remote Client RRequest Message is :
```

```
Command Prompt
D:\>cd JavaApplication32
D:\JavaApplication32>cd src
D:\JavaApplication32\src>javac LENCClient.java
D:\JavaApplication32\src>java LENCClient 10.2.4.7 5068 len

Connecting the client at: //10.2.4.7:5068/len

CLI: Enter the request message to send to remote service:
shifa

CLI: response from remote method is:5
```

PROGRAM - 9 (e)

Program Statement:

Implement a RMI program to find GCD of two given numbers.

Description of the program:

This is Java code that uses the RMI (Remote Method Invocation) mechanism to find and invoke remote methods. The code defines an interface, `GcdInterface`, which declares a Single Remote Method, `gcd(int a, int b)`, that takes two integers as input and returns their greatest common divisor (GCD).

The implementation class, `GcdImpl`, extends `UnicastRemoteObject` and implements the `GcdInterface`. The implementation of the `gcd()` method in `GcdImpl` uses a simple algorithm to compute the GCD of the two input numbers.

The `GcdServer` class creates an instance of `GcdImpl` and binds it to a location in the RMI registry, so that it can be located by clients. The main method of `GcdServer` takes three command-line arguments: the hostname, port number and the service name, which are used to construct the bind location of the `GcdImpl` object in the RMI registry.

The `GcdClient` class looks up the `GcdImpl` object in the RMI registry using the same hostname, port number and service name, and invokes the `gcd()` method on it. The client takes two input numbers from the user and sends them to the remote method. The response from the remote method is the GCD of these two numbers, which is displayed to the user.

Input:

Two numbers to send to the server.

Output:

The GCD of the two numbers is sent back as response.

Packages Used:

- `java.rmi.*`
- `java.io.*`
- `java.net.*`
- `java.rmi.server.*`

Classes and Methods Used:

- `LocateRegistry`
 - `createRegistry()`
- `Naming`
 - `bind()`
 - `lookup()`
- `Remote`
- `RemoteException`
- `MalformedURLException`
- `UnicastRemoteObject`

Program:**Interface:**

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

public interface GcdInterface extends Remote{
    int gcd(int a,int b) throws RemoteException;
}
```

Implementation:

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

public class GcdImpl extends UnicastRemoteObject implements GcdInterface{
    public GcdImpl() throws RemoteException{
    }
    public int gcd(int a,int b)
    {
        System.out.println("REMOTE SERVICE: Remote Client Request Message
is : "+a+" and "+b);
        int temp;
        if(a>b)
        {
            temp=a;
            a=b;
            b=temp;
        }
        while(a!=0)
        {
            temp=a;
            a=b%a;
            b=temp;
        }
        return b;
    }
}
```

Server:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.Naming;
import java.net.MalformedURLException;
import java.rmi.registry.LocateRegistry;

public class GcdServer{
    public GcdServer() throws RemoteException{
    }
    public static void main(String args[]) throws RemoteException
    {
        GcdImpl gcdObj=new GcdImpl();
        int port=Integer.parseInt(args[1]);
        try{
            LocateRegistry.createRegistry(port);
            System.out.println("\n RMI registry created \n");
            String host=args[0];
            String bindLocation="//"+host+": "+port+"/"+args[2];
            Naming.bind(bindLocation,gcdObj);
            System.out.println("\nRMI server ready at "+bindLocation);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Client:

```
import java.rmi.*;
import java.io.*;
import java.net.MalformedURLException;

public class GcdClient {
    public static void main(String args[])
    {
        String
connectLocation="//"+args[0]+": "+Integer.parseInt(args[1])+"/"+args[2];
        GcdInterface gintf=null;
        try{
            System.out.println("\n Connecting the client at:
"+connectLocation);
            gintf=(GcdInterface)Naming.lookup(connectLocation);
            BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
```

```

        System.out.println("\nCLI: Enter two numbers to send to remote
service:");
        int a=Integer.parseInt(br.readLine().trim());
        int b=Integer.parseInt(br.readLine().trim());
        int response=gintf.gcd(a,b);
        System.out.println("\n CLI: response from remote method
is(gcd):"+response);
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
}

```

Output Images:

The image contains two screenshots of Windows Command Prompts. The top window, titled "Command Prompt - java GCDServer 10.2.4.7 5080 gcd", shows the compilation and execution of the GCDServer.java file. The output includes "RMI registry created", "RMI server ready at //10.2.4.7:5080/gcd", and "REMOTE SERVICE: Remote Client RRequest Message is :". The bottom window, titled "Command Prompt", shows the compilation and execution of the GCDClient.java file. The output includes "Connecting the client at: //10.2.4.7:5080/gcd", a prompt for the request message, the input "24" and "18", and the response "CLI: response from remote method is:6".

```

C:\> Command Prompt - java GCDServer 10.2.4.7 5080 gcd
D:\JavaApplication32\src>javac GCDServer.java
D:\JavaApplication32\src>java GCDServer 10.2.4.7 5080 gcd

RMI registry created

RMI server ready at //10.2.4.7:5080/gcd
REMOTE SERVICE: Remote Client RRequest Message is :

C:\> Command Prompt
D:\JavaApplication32\src>javac GCDClient.java
D:\JavaApplication32\src>java GCDClient 10.2.4.7 5080 gcd

Connecting the client at: //10.2.4.7:5080/gcd

CLI: Enter the request message to send to remote service:
24
18

CLI: response from remote method is:6
D:\JavaApplication32\src>

```

PROGRAM - 9 (f)

Program Statement:

Implement a RMI program to find the sum of two given numbers.

Description of the program:

This code defines an RMI interface named **AddInterface** that has a Single Method called add which takes in two integers as input and returns their sum as an integer. The implementation of this interface is done in the **AddImpl** class that extends the **UnicastRemoteObject** and implements the **AddInterface**. The **AddServer** class creates an object of the **AddImpl** class and starts the RMI registry on the specified port and binds the object to the registry. The **AddClient** class connects to the RMI server at the specified location, takes input from the user for two integers, calls the add method on the remote object and prints the result.

Input:

Two numbers to send to the server.

Output:

The sum of the two numbers is sent back as response.

Packages Used:

- java.rmi.*
- java.io.*
- java.net.*
- java.rmi.server.*

Classes and Methods Used:

- LocateRegistry
 - createRegistry()
- Naming
 - bind()
 - lookup()
- Remote
- RemoteException
- MalformedURLException
- UnicastRemoteObject

Program:

Interface:

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

public interface AddInterface extends Remote{
    int add(int a,int b) throws RemoteException;
}
```

Implementation:

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

public class AddImpl extends UnicastRemoteObject implements AddInterface{
    public AddImpl() throws RemoteException{
    }
    public int add(int a,int b)
    {
        System.out.println("REMOTE SERVICE: Remote Client Request Message
is : "+a+" and "+b);
        return a+b;
    }
}
```

Server:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.Naming;
import java.net.MalformedURLException;
import java.rmi.registry.LocateRegistry;

public class AddServer{
    public AddServer() throws RemoteException{
    }
    public static void main(String args[]) throws RemoteException
    {
        AddImpl addObj=new AddImpl();
        int port=Integer.parseInt(args[1]);
        try{
            LocateRegistry.createRegistry(port);
            System.out.println("\n RMI registry created \n");
            String host=args[0];
            String bindLocation="//"+host+": "+port+"/"+args[2];
            Naming.bind(bindLocation,addObj);
            System.out.println("\nRMI server ready at "+bindLocation);
        }
    }
}
```

```

    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
}

```

Client:

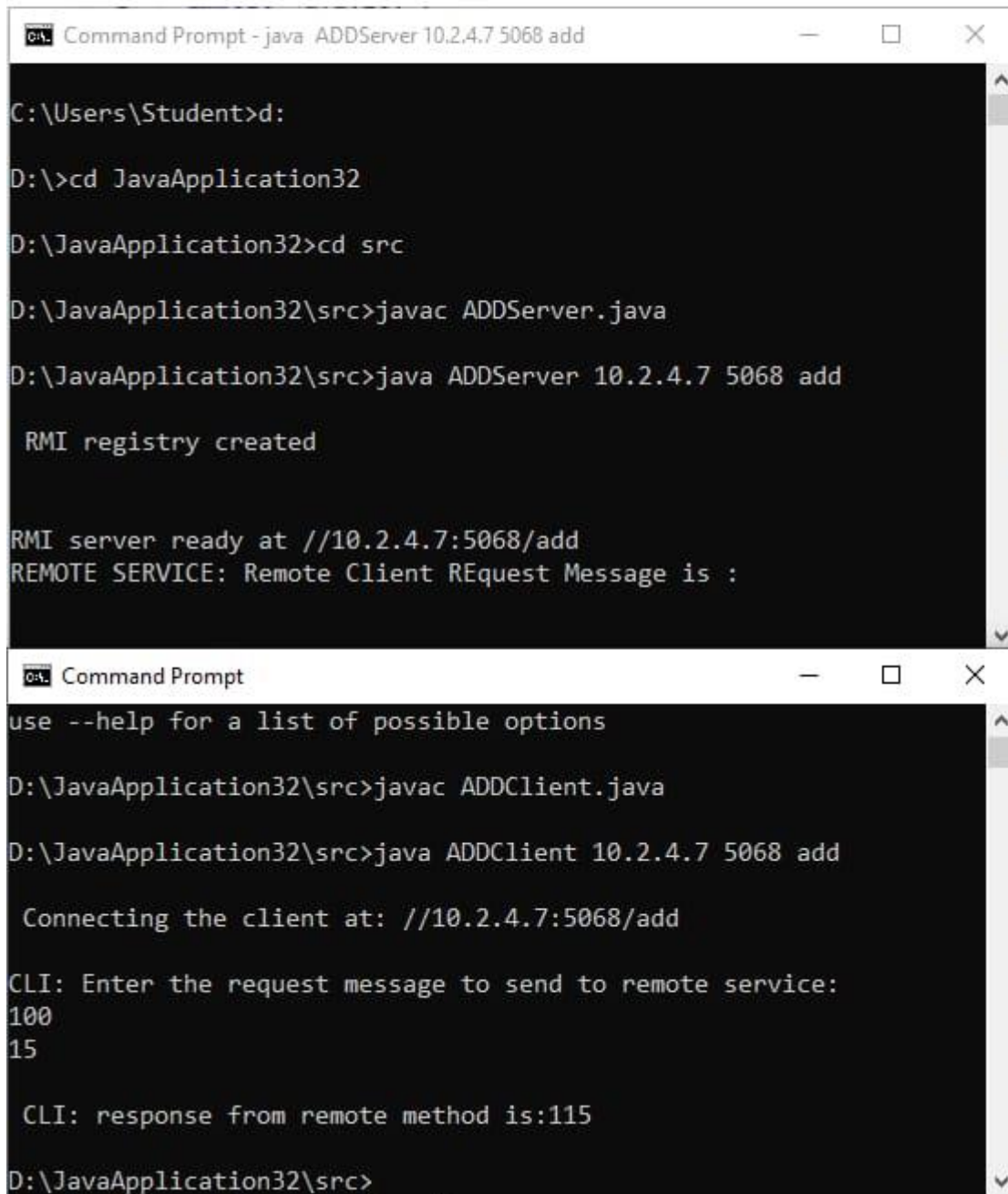
```

import java.rmi.*;
import java.io.*;
import java.net.MalformedURLException;

public class AddClient {
    public static void main(String args[])
    {
        String
connectLocation="//"+args[0]+":"+Integer.parseInt(args[1])+"/"+args[2];
        AddInterface aintf=null;
        try{
            System.out.println("\n Connecting the client at:
"+connectLocation);
            aintf=(AddInterface)Naming.lookup(connectLocation);
            BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
            System.out.println("\nCLI: Enter two numbers to send to remote
service:");
            int a=Integer.parseInt(br.readLine().trim());
            int b=Integer.parseInt(br.readLine().trim());
            int response=aintf.add(a,b);
            System.out.println("\n CLI: response from remote method
is(sum):"+response);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

```


Output Images:



The image displays two screenshots of a Windows Command Prompt window. The top screenshot shows the process of starting an RMI server. The user navigates to the directory `D:\JavaApplication32\src` and runs `javac ADDServer.java` and `java ADDServer 10.2.4.7 5068 add`. The output indicates that the RMI registry was created and the server is ready at `//10.2.4.7:5068/add`. The bottom screenshot shows the execution of the RMI client. The user runs `javac ADDClient.java` and `java ADDClient 10.2.4.7 5068 add`. The output shows the client connecting to the server and sending a request message of 100. The response from the remote method is 115.

```
Command Prompt - java ADDServer 10.2.4.7 5068 add
C:\Users\Student>d:
D:\>cd JavaApplication32
D:\JavaApplication32>cd src
D:\JavaApplication32\src>javac ADDServer.java
D:\JavaApplication32\src>java ADDServer 10.2.4.7 5068 add

RMI registry created

RMI server ready at //10.2.4.7:5068/add
REMOTE SERVICE: Remote Client REquest Message is :

Command Prompt
use --help for a list of possible options
D:\JavaApplication32\src>javac ADDClient.java
D:\JavaApplication32\src>java ADDClient 10.2.4.7 5068 add

Connecting the client at: //10.2.4.7:5068/add

CLI: Enter the request message to send to remote service:
100
15

CLI: response from remote method is:115
D:\JavaApplication32\src>
```

PROGRAM - 10 (a)

Program Statement:

Implement a GUI based FTP Client program to list directory contents/ list files.

Description of the program:

FTP (File Transfer Protocol) is a network protocol for transmitting files between computers over Transmission Control Protocol/Internet Protocol (TCP/IP) connections. Within the TCP/IP suite, FTP is considered an application layer protocol.

FTP File Listing Operation connects to an FTP Server and generates a list of files present in the current working directory of the FTP Login.

This code is for a Java Swing Application that connects to an FTP server using the Apache Commons Net library. When the user clicks the "List Files" button, the program retrieves the values entered in the IP, username, and password fields. It then creates an instance of the FTPClient class from the Apache Commons library, connects to the server, and attempts to log in with the provided credentials. If the login is successful, it retrieves a list of files in the current directory and displays them in a dropdown box. It also appends the file names and other status messages to a text area called "statusWindowArea". Finally, it logs out and disconnects from the server. Any exceptions that may occur during the process are caught and printed to the console.

Input:

IP Address – 10.2.0.5

Username – cse19xx

Password – xxxxxxxx

Output:

The files in the ftp server login are listed in the output window text area.

Packages Used:

- java.io.*
- org.apache.commons.net.ftp.*

Classes and Methods Used:

- FTPClient
 - FTPClient()
 - connect()
 - getReplyCode()
 - login()
 - listFiles()
 - logout()
 - disconnect()
- FTPReply
 - isPositiveCompletion()

- FTPFile
 - getName()

Program:

```
package ftpserverclientby42;

import java.io.*;
import javax.swing.JFileChooser;
import org.apache.commons.net.ftp.*;
import org.apache.commons.net.ftp.FTPClient;
public class FTPClientInterface extends javax.swing.JFrame {
    private void listFilesButtonActionPerformed(java.awt.event.ActionEvent evt)
    {

        String ip = IPField.getText();
        String username = usernameField.getText();
        String password = new String(passwordField.getPassword());

        try {
            FTPClient ftpc = new FTPClient();
            ftpc.connect(ip);
            int status = ftpc.getReplyCode();
            if(FTPReply.isPositiveCompletion(status)){
                statusWindowArea.append("Connected to " + ip + "\n");
                if(ftpc.login(username, password)){
                    statusWindowArea.append("User " +username+ " logged in
to " +ip+ "\n");
                    statusWindowArea.append("Current directory is
"+ftpc.printWorkingDirectory()+"\n");
                    FTPFile[] files = ftpc.listFiles();
                    dropdownBox.removeAllItems();
                    for(FTPFile file : files){
                        dropdownBox.addItem(file.getName());
                        statusWindowArea.append(">" + file.getName()+
"\n");
                    }
                    if(ftpc.logout()){
                        statusWindowArea.append("User " +username+ " logged
out. \n");
                    }
                    ftpc.disconnect();
                    statusWindowArea.append("Disconnected from " +ip +
"\n");
                }
            } else {
                statusWindowArea.append("Invalid Credentials");
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Output Images:



PROGRAM - 10 (b)

Program Statement:

Implement a GUI based FTP Client program to download a file.

Description of the program:

FTP File Download Operation connects to an FTP Server and downloads the specified file from the specified path on the remote system to the local system.

The code you provided is for a Java Swing based FTP client. The main functionalities are connecting to an FTP server, logging in, listing files, and downloading a file from the server. The "listFilesButtonActionPerformed" method is triggered when the "List Files" button is pressed, it connects to the FTP server using the IP, username and password provided by the user and lists all the files in the current directory. The "downloadButtonActionPerformed" method is triggered when the "Download" button is pressed, it allows the user to select a file from the list of files in the current directory and download it to the local machine.

Input:

IP Address – 10.2.0.5

Username – cse19xx

Password – xxxxxxxx

Output:

The files in the ftp server login are listed and download operation is performed for a file .

Packages Used:

- java.io.*
- org.apache.commons.net.ftp.*

Classes and Methods Used:

- FTPClient
 - FTPClient()
 - connect()
 - getReplyCode()
 - login()
 - listFiles()
 - logout()
 - disconnect()
 - retrieveFile()
- FTPReply
 - isPositiveCompletion()
- FTPFile
 - getName()

Program:

```
package ftpserverclientby42;

import java.io.*;
import javax.swing.JFileChooser;
import org.apache.commons.net.ftp.*;
import org.apache.commons.net.ftp.FTPClient;

public class FTPClientInterface extends javax.swing.JFrame {
    private void listFilesButtonActionPerformed(java.awt.event.ActionEvent evt)
    {

        String ip = IPField.getText();
        String username = usernameField.getText();
        String password = new String(passwordField.getPassword());

        try {
            FTPClient ftpc = new FTPClient();
            ftpc.connect(ip);
            int status = ftpc.getReplyCode();
            if(FTPReply.isPositiveCompletion(status)){
                statusWindowArea.append("Connected to " + ip + "\n");
                if(ftpc.login(username, password)){
                    statusWindowArea.append("User " +username+ " logged in
to " +ip+ "\n");
                    statusWindowArea.append("Current directory is
"+ftpc.printWorkingDirectory()+"\n");
                    FTPFile[] files = ftpc.listFiles();
                    dropdownBox.removeAllItems();
                    for(FTPFile file : files){
                        dropdownBox.addItem(file.getName());
                        statusWindowArea.append(">" + file.getName()+
"\n");
                    }
                } else {
                    statusWindowArea.append("Invalid Credentials");
                }
            } catch(Exception e) {
                e.printStackTrace();
            }
        }

        private void downloadButtonActionPerformed(java.awt.event.ActionEvent evt)
        {

            try {
                String filename =
                (String)dropdownBox.getSelectedItem();
                System.out.println(filename);
                File fname = new File(filename);
                OutputStream os = new BufferedOutputStream(new
FileOutputStream(fname));
                if(ftpc.retrieveFile(filename, os))
                {
```

```

        statusWindowArea.append("Succesfully downloaded
\n");
    }else{
        statusWindowArea.append("Couldn't download\n");
    }
    os.close();
    if(ftpc.logout()){
        statusWindowArea.append("User " +username+ " logged
out. \n");
    }
    ftpc.disconnect();
    statusWindowArea.append("Disconnected from " +ip +
\n");
    }
    } else {
        statusWindowArea.append("Invalid Credentials");
    }
    } catch(Exception e) {
        e.printStackTrace();
    }
}
}
}

```

Output Images:

Ftp Client

Ip address: 10.2.0.5

Username: cse1985

Password: *****

Connect Login Disconnect

Operations:

List Files List directories Change directory

Download Upload Go to Parent Directory

build.xml ▼

Status Window

- current working directory:/home/cse19/cse1985
- File download unsuccessful
- Current dir:/home/cse19/cse1985/BulletinBoard
- File downloaded successfully

PROGRAM - 10 (c)

Program Statement:

Implement a GUI based FTP Client program to upload a file

Description of the program:

FTP File Upload Operation connects to an FTP Server and uploads the specified file from the specified path on the local system to the remote system running the FTP Server.

This code is for a Java Swing Application that implements a basic FTP client interface. The interface includes a text field for entering the IP address of the FTP server, a text field for entering the username, a password field for entering the password, a button to list files on the server, a dropdown box to select a file to download or upload, a button to upload a file, and a button to download a file.

The listFilesButtonActionPerformed method is called when the "list files" button is pressed. It connects to the FTP server using the IP address, username, and password provided by the user. If the connection is successful and the credentials are valid, it lists the files on the server and displays them in the dropdown box.

The uploadButtonActionPerformed method is called when the "upload" button is pressed. It connects to the FTP server using the IP address, username, and password provided by the user. If the connection is successful and the credentials are valid, it opens a file chooser dialog for the user to select a file to upload. The selected file is then uploaded to the server.

Input:

IP Address – 10.2.0.5

Username – cse19xx

Password – xxxxxxx

Output:

The files in the ftp server login are listed in the output window text area.

Packages Used:

- java.io.*
- org.apache.commons.net.ftp.*

Classes and Methods Used:

- FTPClient
 - FTPClient()
 - connect()
 - getReplyCode()
 - login()
 - listFiles()
 - logout()
 - disconnect()
 - storeFile()
- FTPReply
 - isPositiveCompletion()
- FTPFile
 - getName()

Program:

```
package ftpserverclientby42;

import java.io.*;
import javax.swing.JFileChooser;
import org.apache.commons.net.ftp.*;
import org.apache.commons.net.ftp.FTPClient;

public class FTPClientInterface extends javax.swing.JFrame {
    private void listFilesButtonActionPerformed(java.awt.event.ActionEvent evt)
    {

        String ip = IPField.getText();
        String username = usernameField.getText();
        String password = new String(passwordField.getPassword());

        try {
            FTPClient ftpc = new FTPClient();
            ftpc.connect(ip);
            int status = ftpc.getReplyCode();
            if(FTPReply.isPositiveCompletion(status)){
                statusWindowArea.append("Connected to " + ip + "\n");
                if(ftpc.login(username, password)){
                    statusWindowArea.append("User " +username+ " logged in
to " +ip+ "\n");
                    statusWindowArea.append("Current directory is
"+ftpc.printWorkingDirectory()+"\n");
                    FTPFile[] files = ftpc.listFiles();
                    dropdownBox.removeAllItems();
                    for(FTPFile file : files){
                        dropdownBox.addItem(file.getName());
                        statusWindowArea.append(">" + file.getName()+
"\n");
                    }
                }
            }
        }
    }
}
```

```

        } else {
            statusWindowArea.append("Invalid Credentials");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private void uploadButtonActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    String ip = IPField.getText();
    String username = usernameField.getText();
    String password = new String(passwordField.getPassword());

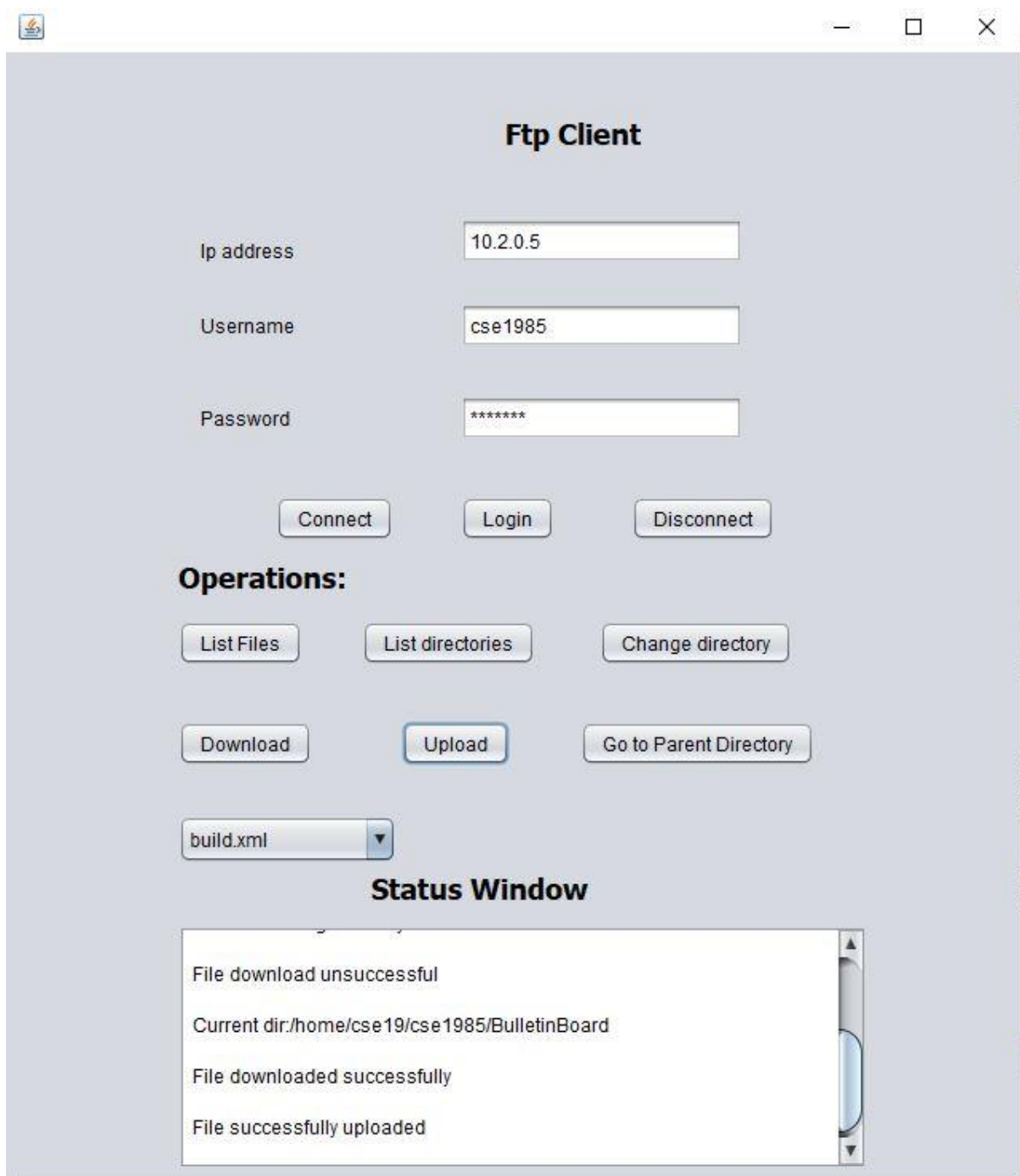
    try {
        ftpc = new FTPClient();
        ftpc.connect(ip);
        int status = ftpc.getReplyCode();
        if(FTPReply.isPositiveCompletion(status)){
            statusWindowArea.append("Connected to " + ip + "\n");
            if(ftpc.login(username, password)){
                statusWindowArea.append("User " +username+ " logged in
to " +ip+ "\n");
                statusWindowArea.append("Current directory is
"+ftpc.printWorkingDirectory()+"\n");
                fileChooser = new JFileChooser();
                fileChooser.showOpenDialog(null);
                File file = fileChooser.getSelectedFile();
                InputStream is = new BufferedInputStream(new
FileInputStream(file));
                if(ftpc.storeFile(file.getName(), is)){
                    statusWindowArea.append("Successfully uploaded\n");
                    is.close();
                } else {
                    statusWindowArea.append("Couldn't upload\n");
                }

            } else {
                statusWindowArea.append("Invalid Credentials");
            }

        } else {
            statusWindowArea.append("Connection Failed");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Output Images:



The image shows a screenshot of a Java-based FTP client application window. The window has a title bar with standard OS controls (minimize, maximize, close). The main area is titled "Ftp Client" and contains input fields for "Ip address" (10.2.0.5), "Username" (cse1985), and "Password" (masked with asterisks). Below these are buttons for "Connect", "Login", and "Disconnect". A section titled "Operations:" contains buttons for "List Files", "List directories", "Change directory", "Download", "Upload", and "Go to Parent Directory". A file list shows "build.xml" selected. At the bottom is a "Status Window" with a scrollable text area containing messages: "File download unsuccessful", "Current dir:/home/cse19/cse1985/BulletinBoard", "File downloaded successfully", and "File successfully uploaded".

Ftp Client

Ip address: 10.2.0.5

Username: cse1985

Password: *****

Connect Login Disconnect

Operations:

List Files List directories Change directory

Download Upload Go to Parent Directory

build.xml ▼

Status Window

File download unsuccessful

Current dir:/home/cse19/cse1985/BulletinBoard

File downloaded successfully

File successfully uploaded

PROGRAM – 11

Program Statement:

Implement a GUI/CUI based 3-tier application.

Description of the program:

N-tier denotes a software engineering concept used for the design and implementation of software systems using client/server architecture divided into multiple tiers. This decouples design and implementation complexity, thus allowing for the scalability of the deployed system.

A three-tier application is a specific type of n-tier architecture. In the case of three-tier architecture, the tiers are as follows:

- Presentation tier (also known as the user interface or the client application): -

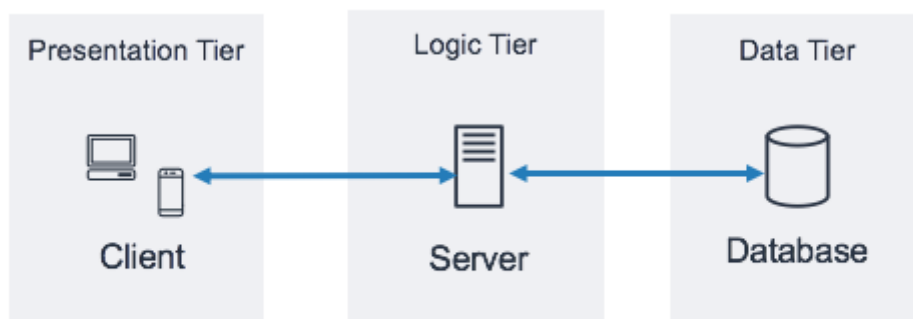
The user interaction is managed by the presentation tier, which provides an easy-to-operate front end. It communicates user requests to the application server and presents the results to the user.

- Business logic tier (also known as the application server): -

The business rules are managed by the business tier(also known as the middle tier), which controls and operates the entire application framework. The business rules are removed from the client and are executed in the application server. The application server ensures that the business rules are processed correctly. It also serves as an intermediary between the client application and database server.

- Data storage tier (also known as the database server): -

The underlying data is stored and served by the data storage tier, also known as data persistence. It serves Data whenever the application server sends a request to it.



Our 3-tier application consists of a database server, backend server in this case java, and a frontend User interface, which in this case is Java Swing Interface. The interface includes a text field for entering the username, a password field for entering the password, a text field to enter the fields/columns to be used in the query, a button to list tables in the database, a dropdown box to select a table to query, and a button to display the contents of table using the select query.

The listTablesBtnActionPerformed method is called when the "list tables" button is pressed. It connects to the MySQL server and lists the tables in the database on the server and displays them in the dropdown box.

The displayTableBtnActionPerformed method is called when the "Display Tables" button is pressed. It connects to the MySQL server and displays the content in the selected table of the dropdown box with the specified fields (default *).

Input:

Username – cse19xx

Password – xxxxxxxx

Fields – * or any column name(s)

Output:

The tables in the database are listed, and their content is displayed.

Packages Used:

- java.sql.*

Classes and Methods Used:

- Connection
 - createStatement()
 - close()
- DriverManager
 - getConnection()
- Statement
 - executeQuery()
- ResultSet
 - getMetaData()
 - next()
 - getString()
- ResultSetMetaData
 - getColumnCount()
 - getColumnName()

Program:

```
import java.sql.*;

public class NewJFrame extends javax.swing.JFrame {

    Connection con;

    private void connectBtnActionPerformed(java.awt.event.ActionEvent evt) {

        String user = jTextField1.getText();

        String password = new String(jPasswordField1.getPassword());

        try{

            Class.forName("com.mysql.cj.jdbc.Driver");

            con = DriverManager.getConnection("jdbc:mysql://10.2.4.6:3306/student", user, password);

            jTextArea1.append("connection Established \n");

            connectBtn.setEnabled(false);

            disconnectBtn.setEnabled(true);

        }

        catch(Exception e){

            e.printStackTrace();

        }

    }

    private void disconnectBtnActionPerformed(java.awt.event.ActionEvent evt)

    try{

        con.close();

        jTextArea1.append("\n Connection closed");

        connectBtn.setEnabled(true);

        disconnectBtn.setEnabled(false);

    }

    catch(Exception e){

        e.printStackTrace();

    }

}

    private void listTablesBtnActionPerformed(java.awt.event.ActionEvent evt) {

        try{
```

```

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("show tables");

jComboBox1.removeAllItems();

while(rs.next()){

    JTextArea1.append(rs.getString(1)+"\n");

    jComboBox1.addItem(rs.getString(1)+"\n");

}

}catch(Exception e){

    e.printStackTrace();

}

}

private void displayTableBtnActionPerformed(java.awt.event.ActionEvent evt) {

    String str = (String)jComboBox1.getSelectedItem();

    String fld = jTextField2.getText();

    try{

        Statement stmt = con.createStatement();

        ResultSet rs = stmt.executeQuery("select "+fld+" from " + str);

        JTextArea1.append("\n");

        ResultSetMetaData rsmd = rs.getMetaData();

        for(int i = 1; i <= rsmd.getColumnCount() ; i++)

            JTextArea1.append(rsmd.getColumnName(i) + "\t");

        JTextArea1.append("\n");

        while(rs.next()){

            for(int i = 1; i <= rsmd.getColumnCount() ; i++)

                JTextArea1.append(rs.getString(i) + "\t");

            JTextArea1.append("\n");

        }

    }catch(Exception e){

```



```
e.printStackTrace();  
}  
}
```

Output Images:

The screenshot shows a Java Swing window titled "three tier". The window has a light gray background and contains the following elements:

- Username:** A text field containing "cse19306".
- Password:** A text field.
- Fields:** A text field containing "*".
- Buttons:** "Connect", "Disconnect", "List Tables:", and "Display Tables:".
- Database Selection:** A dropdown menu showing "studdetails".
- Output Area:** A white rectangular area at the bottom containing the following text:
connection Established
studdetails

rollno	name1	branch	address
21	adams	cse	mjcet
22	turner	cse	mjcet
25	march	ece	mjcet
26	april	ece	mjcet
41	mongo	aids	mjcet

Connection closed

PROGRAM – 12

Program Statement:

NFS Case Study – mounting of specified directory from Linux server to Linux client, windows client.

Description of the program:

NFS, stands for **N**etwork **F**ile **S**ystem, is a server-client protocol used for sharing files between systems. NFS enables you to mount a remote share locally. You can then directly access any of the files on that remote share.

The basic idea behind NFS is that each file server provides a standardized view of its local file system. This approach has been adopted for other distributed files systems as well. NFS allows a heterogeneous collection of processes to run on different operating systems and machines, and share a common file system.

The model underlying NFS and similar systems is that of a remote file service. In this model, clients are offered transparent access to a file system that is managed by a remote server. However, clients are normally unaware of the actual location of files. Instead, they are offered an interface to a file system that is similar to the interface offered by a conventional local file system. In particular, the client is offered only an interface containing various file operations, but the server is responsible for implementing those operations. This model is therefore also referred to as the remote access model.

A client accesses the file system using the system calls provided by its local operating system. However, the local UNIX file system interface is replaced by an interface to the Virtual File System (VFS), which by now is a de facto standard for interfacing to different (distributed) file systems.

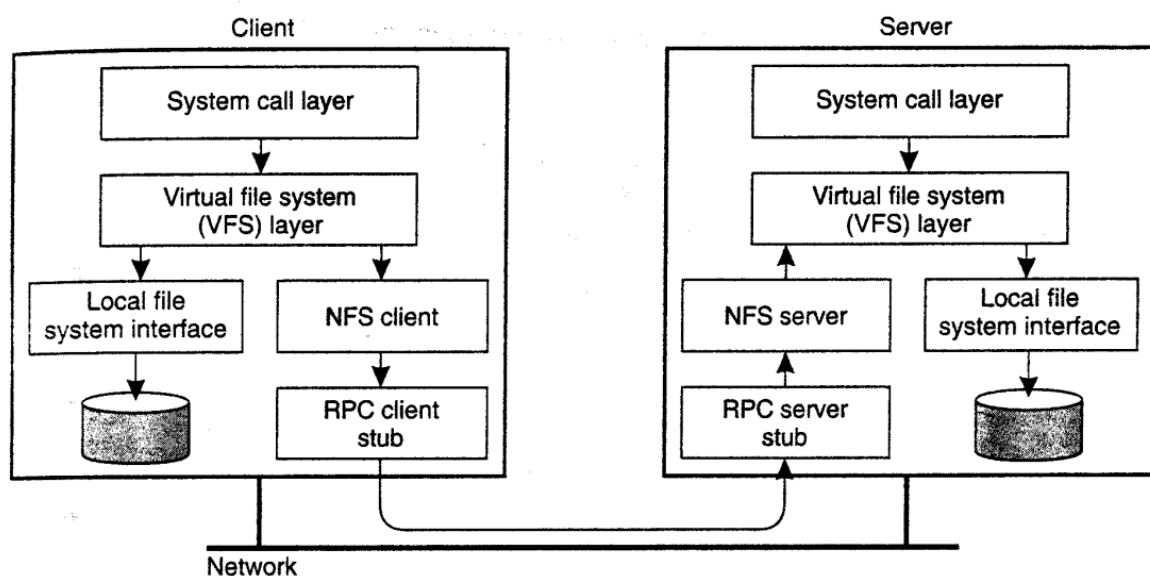
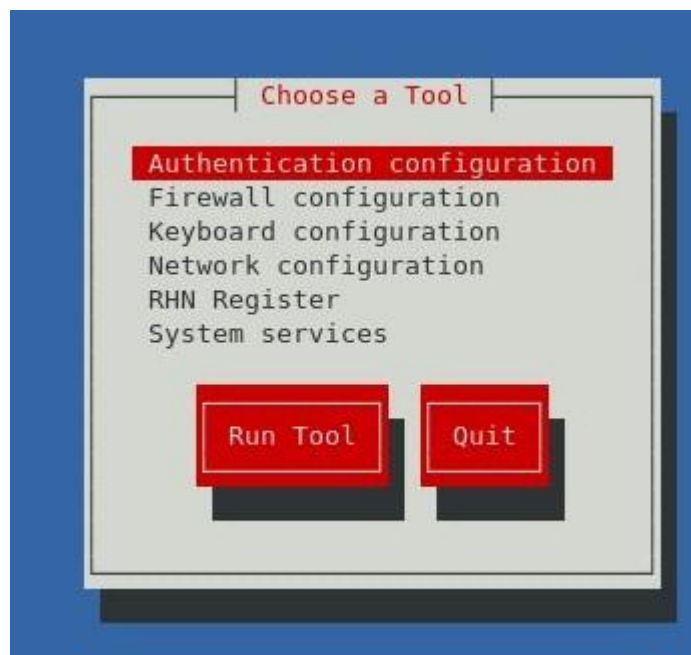


Figure 11-2. The basic NFS architecture for UNIX systems.

Procedure:

Linux Server:



- Install linux full-fledged – RHEL (Red hat enterprise linux)
- Open terminal, type
`>setup`
- In setup select system services
- Ensure NFS service is enabled
- Create a directory in root directory under var sub-directory
`>mkdir /var/mySharedFolder`
- Give permissions to the folder
`>chmod 777 mySharedFolder`
- Open exports file
`>vi /etc/exports`
- Mention the shared folder path , client Ip that can mount it and the different permissions by adding a line at the end file as below
`>/var/mySharedFolder 10.2.0.14 (rw, sync, no_root_squash, no_all_squash, insecure)`
- Save and exit the file
- Then restart the NFS service to apply the changes with the command
`>service nfs restart`

```

File Edit View Search Terminal Help
/var/nfsprgshare/ 10.2.4.8 (rw,sync,no_root_squash, no_all_squash,insecure)
/var/mysharefolder 10.2.4.18 (rw,sync,no_root_squash,no_all_squash,insecure)
~
~
~
~

```

```
Applications Places System root@localhost:/
File Edit View Search Terminal Help
[root@localhost ~]# setup
[root@localhost ~]# ls
'          del_exam.sh Desktop install.log Pictures Videos
anaconda-ks.cfg del_user.sh Documents install.log.syslog Public
create_user.sh del_user.sh Downloads Music Templates
[root@localhost ~]# cd ..
[root@localhost /]# ls
bin cgroup etc lib lost+found misc net proc sbin srv tmp usr
boot dev home lib64 media mnt opt root selinux sys usr
[root@localhost /]# mkdir /var/mysharefolder
[root@localhost /]# chmod 777 /var/mysharefolder
[root@localhost /]# vi /etc/exports
[root@localhost /]# service nfs restart
Shutting down NFS daemon: [ OK ]
Shutting down NFS mountd: [ OK ]
Shutting down NFS quotas: [ OK ]
Shutting down NFS services: [ OK ]
Shutting down RPC idmapd: [ OK ]
Starting NFS services: exportfs: No options for /var/nfsprgshare/ 10.2.4.8: suggest 10.2.4.8(sync) to avoid warning
exportfs: No host name given with /var/nfsprgshare/ (rw,sync,no_root_squash,, suggest *(rw,sync,no_root_squash, to avoid warning
exportfs: /etc/exports:1: syntax error: bad option list [FAILED]
Starting NFS quotas: [ OK ]
Starting NFS mountd: [ OK ]
Starting NFS daemon: [ OK ]
Starting RPC idmapd: [ OK ]
[root@localhost /]#
```

Linux-client: -

- Similar to server ensure that the NFS service is enabled.
- Make a directory
>mkdir myMountFolder
- Mount the shared folder of the server on the client machine

Syntax: -

>mount -t nfs server-IP:folder-path-on-server folder-path-on-client

Ex: -

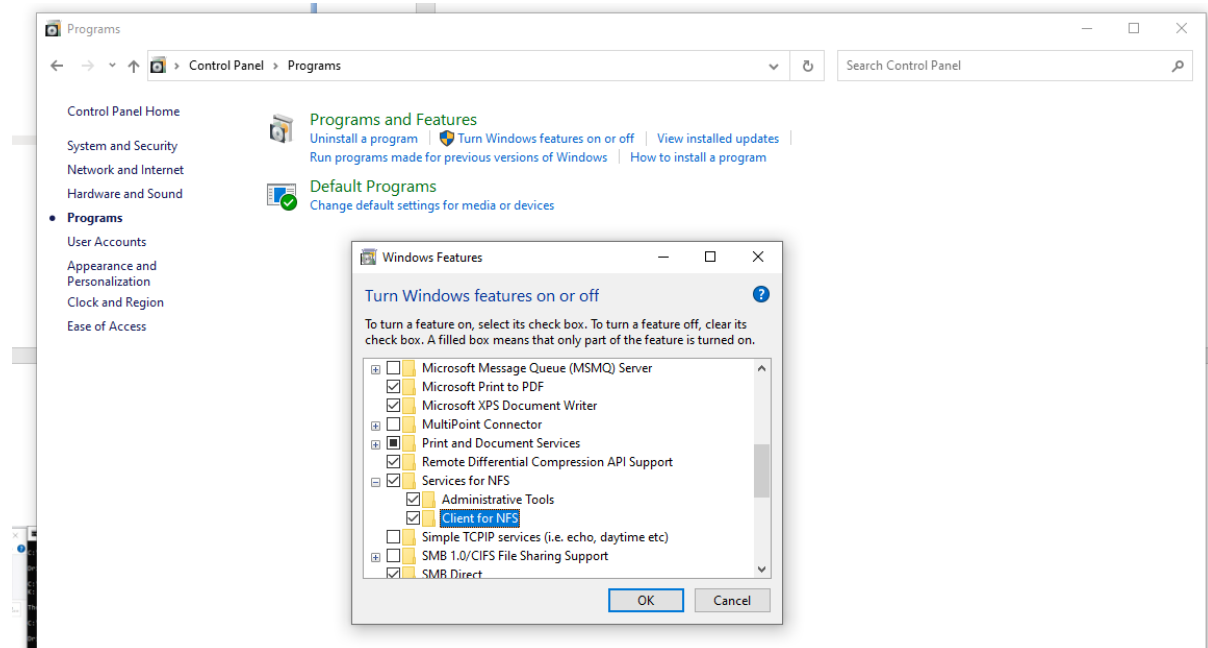
>mount -t nfs 10.2.0.5:/var/mySharedFolder /myMountFolder

- Now you can access the mounted folder by changing directory to myMountFolder

```
Applications Places System root@localhost:~/Desktop/my
File Edit View Search Terminal Help
[root@localhost Desktop]# ls
gnome-terminal.desktop Packages
[root@localhost Desktop]# mkdir mymount
[root@localhost Desktop]# mount -t nfs 10.2.0.5:/var/cshare2022 mymount
[root@localhost Desktop]# ls
gnome-terminal.desktop Packages
[root@localhost Desktop]# cd mymount
[root@localhost mymount]# ls
ChatServer1 commons-net-3.6.jar NewJFrame.java nfs.txt System Volume Information tcpechocs weka.ppt
[root@localhost mymount]#
```

Windows client:-

- Go to control panel>programs>turn windows features on or off
- Search for “services for NFS”
- Check “client for NFS” and “administrative tools” in it.
- Click ok, after changes apply, restart system



Windows client-CMD:

- Run CMD as Admin
- Put the below command to mount

Syntax:-

>mount -o anon \\server-ip\folder-path\ drive-letter

Ex:-

>mount -o anon \\10.2.0.5\var\mySharedFolder\ K:

- Then access the drive in CMD by typing,

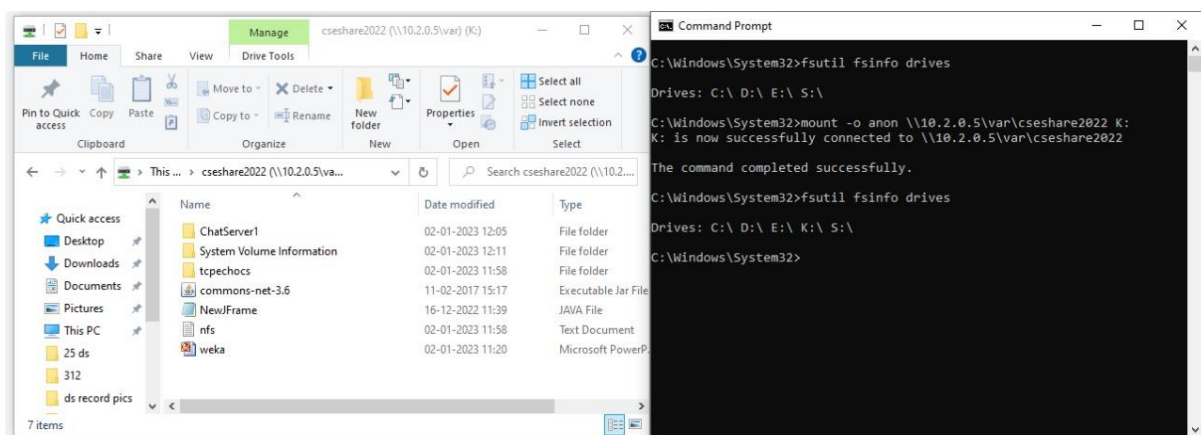
>K:

K:>dir

.....

.....

.....folder contents are displayed



Windows client-GUI:

- Click on This PC
- Above address bar, in the computer tab select “Map network drive”
- Select a drive letter which has not been already used.
- Specify folder on server which is to be shared as
[\\10.2.0.5\var\mySharedFolder](https://10.2.0.5/var/mySharedFolder)
- Check remount at sign-in
- Click finish

