THIAGARAJAR COLLEGE OF ENGINEERING



*(Govt. Aided & Autonomous Institution, Affiliated to Anna University, Chennai)*

Department of Electronics and communication

18EC560-Data communication networks Project Report.

By: SRIRAM VIGNESH V \_ 21D095

# PROBLEM STATEMENT :-

The problem is to develop a server-client chat application that allows users to exchange text messages and JPEG images. The server and client should be able to establish a connection and send/receive both text and image data seamlessly. The objective is to create a reliable and user-friendly chat application with image sharing capabilities.

# OBJECTIVE :-

The objective of this code is to create a server-client chat application that achieves the following goals:

1. Establish a connection: The code sets up a server and client that can establish a network connection over a local network or the internet.
2. Text Messaging: Users should be able to send and receive text messages between the server and client.
3. Image Sharing: Users can send and receive JPEG images between the server and client. The images are encoded in Base64 format for transmission.
4. User Interface: The code provides a user-friendly graphical user interface (GUI) for both the server and client, allowing users to type messages, send images, and display received images.
5. Error Handling: The code handles exceptions and errors gracefully, ensuring the application's stability and reliability.
6. Interaction: Users can interact with the application using the provided buttons and input fields.
7. File Format Validation: The code checks if the selected files for image transfer are in JPEG format and provides feedback if the selected file is not a valid image.
8. Save and Display: The received images can be saved to a local file system and displayed in the client's GUI.

Overall, the code aims to provide a functional and interactive chat application that facilitates communication between a server and a client, allowing the exchange of text and image data while maintaining a user-friendly interface and robust error handling.

**OOPS CONCEPTS USED:-**

This Java program involves several Object-Oriented Programming (OOP) concepts. Here are the main concepts and their explanations:

* 1. Classes and Objects:
* In this program, `Server` and `Client` are classes. They serve as blueprints for creating objects.
* `Server` and `Client` objects are created to represent the server and client applications, respectively.
  1. Inheritance:
     + Both `Server` and `Client` classes extend the `JFrame` class. They inherit properties and methods from `JFrame` to create GUI windows.
  2. Encapsulation:
     + The instance variables (e.g., `serverSocket`, `clientSocket`,

`input`, `output`, etc.) are declared as private to encapsulate their state within the class. They can be accessed and modified only through public methods.

* 1. Abstraction:
     + The details of the socket communication, GUI setup, and message handling are abstracted away from the user. The user interacts with the GUI and sends/receives messages without needing to understand the underlying implementation.
  2. Polymorphism:
     + There isn’t a direct use of polymorphism in this code. However, Java’s Swing framework internally uses polymorphism extensively. For example, `JFrame` can display different types of components (like text fields, buttons, labels) and handle various events.
  3. Interfaces (ActionListener):

- The `ActionListener` interface is used to handle events like button

clicks. Both `Server` and

`Client` implement `ActionListener` to define the behavior when the

“Send” button is clicked.

* 1. Composition (or Aggregation):
     + Both `Server` and `Client` classes use components like

`JTextArea`, `JTextField`, `JButton`, etc., which are composed within the class to create the GUI.

* 1. File Handling:
     + The code involves reading and writing files, demonstrating how objects interact with external resources. For example, in

`sendImage()` method, `FileInputStream` is used to read a file.

* 1. Exception Handling:
     + The program uses try-catch blocks to handle potential exceptions that may occur during file handling and socket communication. This is crucial for robustness.
  2. Multithreading:
     + Both the server and client applications use threads. They have a separate thread for receiving messages to ensure that the GUI remains responsive while waiting for incoming messages.
  3. Method Overriding:
     + `sendMessage`, `sendImage`, `receiveImage`, `displayImage` methods are overridden in both `Server` and `Client` classes to define their specific behavior.
  4. Constructor:
     + Both `Server` and `Client` classes have a constructor method that is responsible for initializing the objects when they are created.

Overall, this program makes use of several key Object-Oriented Programming concepts to organize and implement the functionality of a chat application with image sharing.

# SOURCE CODE:

SERVER SIDE:

package server;

import java.awt.\*; import java.awt.event.\*; import java.io.\*; import java.net.\*;

import javax.imageio.ImageIO; import javax.swing.\*;

import java.util.Base64;

public class Server extends JFrame { private ServerSocket serverSocket; private Socket clientSocket; private BufferedReader input; private PrintWriter output; private JTextArea chatArea; private JTextField messageField; private JButton sendImageButton; private JFileChooser fileChooser;

public Server() { setTitle("Server"); setSize(600, 400);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

chatArea = new JTextArea(); chatArea.setEditable(false);

JScrollPane scrollPane = new JScrollPane(chatArea); add(scrollPane, BorderLayout.CENTER);

messageField = new JTextField(); messageField.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) { sendMessage();

}

});

add(messageField, BorderLayout.SOUTH);

sendImageButton = new JButton("Send Image"); sendImageButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) { sendImage();

}

});

add(sendImageButton, BorderLayout.NORTH);

fileChooser = new JFileChooser(); fileChooser.setFileSelectionMode(JFileChooser.FILES\_ONLY);

try {

serverSocket = new ServerSocket(3333); clientSocket = serverSocket.accept();

input = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

output = new PrintWriter(clientSocket.getOutputStream(), true);

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

}

Thread messageReceiver = new Thread(new Runnable() { public void run() {

while (true) { try {

String receivedMessage = input.readLine(); if (receivedMessage != null) {

if (receivedMessage.startsWith("Image:")) {

// Handle image received from

client receivedMessage.substring(6);

String encodedImage = receiveImage(encodedImage);

receivedMessage + "\n");

} else {

chatArea.append("Client: " +

}

}

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

}

}

}

});

messageReceiver.start();

}

private void sendMessage() {

String message = messageField.getText(); chatArea.append("Server: " + message + "\n"); output.println(message); messageField.setText("");

}

private void sendImage() {

int result = fileChooser.showOpenDialog(this);

if (result == JFileChooser.APPROVE\_OPTION) {

File selectedFile = fileChooser.getSelectedFile(); String fileName = selectedFile.getName();

// Check if the selected file is a JPG image if (fileName.toLowerCase().endsWith(".jpg")) {

try {

FileInputStream fis = new FileInputStream(selectedFile);

byte[] buffer = new byte[(int) selectedFile.length()];

fis.read(buffer); fis.close();

String encodedImage = Base64.getEncoder().encodeToString(buffer);

chatArea.append("Server: Sending JPG image '" +

fileName + "'\n");

output.println("Image:" + encodedImage); // Notify the client that an image has been sent

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

}

} else {

chatArea.append("Server: Please select a JPG image

to send.\n");

}

}

}

private void receiveImage(String encodedImage) { try {

byte[] decodedImage = Base64.getDecoder().decode(encodedImage);

JFileChooser imageFileChooser = new JFileChooser(); int result = imageFileChooser.showSaveDialog(this);

if (result == JFileChooser.APPROVE\_OPTION) { File savedFile =

imageFileChooser.getSelectedFile();

FileOutputStream fos = new FileOutputStream(savedFile);

fos.write(decodedImage); fos.close();

chatArea.append("Client: Received image '" + savedFile.getName() + "'\n");

displayImage(savedFile);

}

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

}

}

private void displayImage(File imageFile) { try {

Image image = ImageIO.read(imageFile); AbstractButton imageLabel = null; imageLabel.setIcon(new ImageIcon(image));

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

}

}

public static void main(String[] args) { SwingUtilities.invokeLater(new Runnable() {

public void run() {

new Server().setVisible(true);

}});}

}

# CLIENT SIDE:

package client; import java.awt.\*;

import java.awt.event.\*; import java.io.\*; import java.net.\*;

import javax.imageio.ImageIO; import javax.swing.\*;

import java.util.Base64;

public class Client extends JFrame { private Socket socket;

private BufferedReader input; private PrintWriter output; private JTextArea chatArea; private JTextField messageField; private JLabel imageLabel; private JButton sendImageButton; private JFileChooser fileChooser;

private byte[] buffer = new byte[1024];

public Client() { setTitle("Client"); setSize(600, 400);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

chatArea = new JTextArea(); chatArea.setEditable(false);

JScrollPane scrollPane = new JScrollPane(chatArea); add(scrollPane, BorderLayout.CENTER);

messageField = new JTextField(); messageField.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) { sendMessage();

}

});

add(messageField, BorderLayout.SOUTH);

imageLabel = new JLabel(); add(imageLabel, BorderLayout.NORTH);

sendImageButton = new JButton("Send Image"); sendImageButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) { sendImage();

}

});

add(sendImageButton, BorderLayout.NORTH);

fileChooser = new JFileChooser(); fileChooser.setFileSelectionMode(JFileChooser.FILES\_ONLY);

try {

socket = new Socket("127.0.0.1", 3333); input = new BufferedReader(new

InputStreamReader(socket.getInputStream()));

output = new PrintWriter(socket.getOutputStream(),

true);

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

}

Thread messageReceiver = new Thread(new Runnable() { public void run() {

while (true) { try {

String receivedMessage = input.readLine(); if (receivedMessage != null) {

if (receivedMessage.startsWith("Image:")) {

// Handle image received from

server receivedMessage.substring(6);

String encodedImage = receiveImage(encodedImage);

receivedMessage + "\n");

} else {

chatArea.append("Server: " +

}

}

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

}

}

}

});

messageReceiver.start();

}

private void sendMessage() {

String message = messageField.getText(); chatArea.append("Client: " + message + "\n"); output.println(message); messageField.setText("");

}

private void sendImage() {

int result = fileChooser.showOpenDialog(this);

if (result == JFileChooser.APPROVE\_OPTION) {

File selectedFile = fileChooser.getSelectedFile(); String fileName = selectedFile.getName();

// Check if the selected file is a JPG image if (fileName.toLowerCase().endsWith(".jpg")) {

try {

FileInputStream fis = new FileInputStream(selectedFile);

byte[] buffer = new byte[(int) selectedFile.length()];

fis.read(buffer); fis.close();

String encodedImage = Base64.getEncoder().encodeToString(buffer);

chatArea.append("Client: Sending JPG image '" +

fileName + "'\n");

output.println("Image:" + encodedImage);

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

}

} else {

chatArea.append("Client: Please select a JPG image

to send.\n");

}

}

}

private void receiveImage(String encodedImage) { try {

byte[] decodedImage = Base64.getDecoder().decode(encodedImage);

JFileChooser imageFileChooser = new JFileChooser(); int result = imageFileChooser.showSaveDialog(this);

if (result == JFileChooser.APPROVE\_OPTION) { File savedFile =

imageFileChooser.getSelectedFile();

FileOutputStream fos = new FileOutputStream(savedFile);

fos.write(decodedImage); fos.close();

chatArea.append("Server: Received image '" + savedFile.getName() + "'\n");

displayImage(savedFile);

}

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

}

}

private void displayImage(File imageFile) { try {

Image image = ImageIO.read(imageFile); imageLabel.setIcon(new ImageIcon(image));

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

}

}

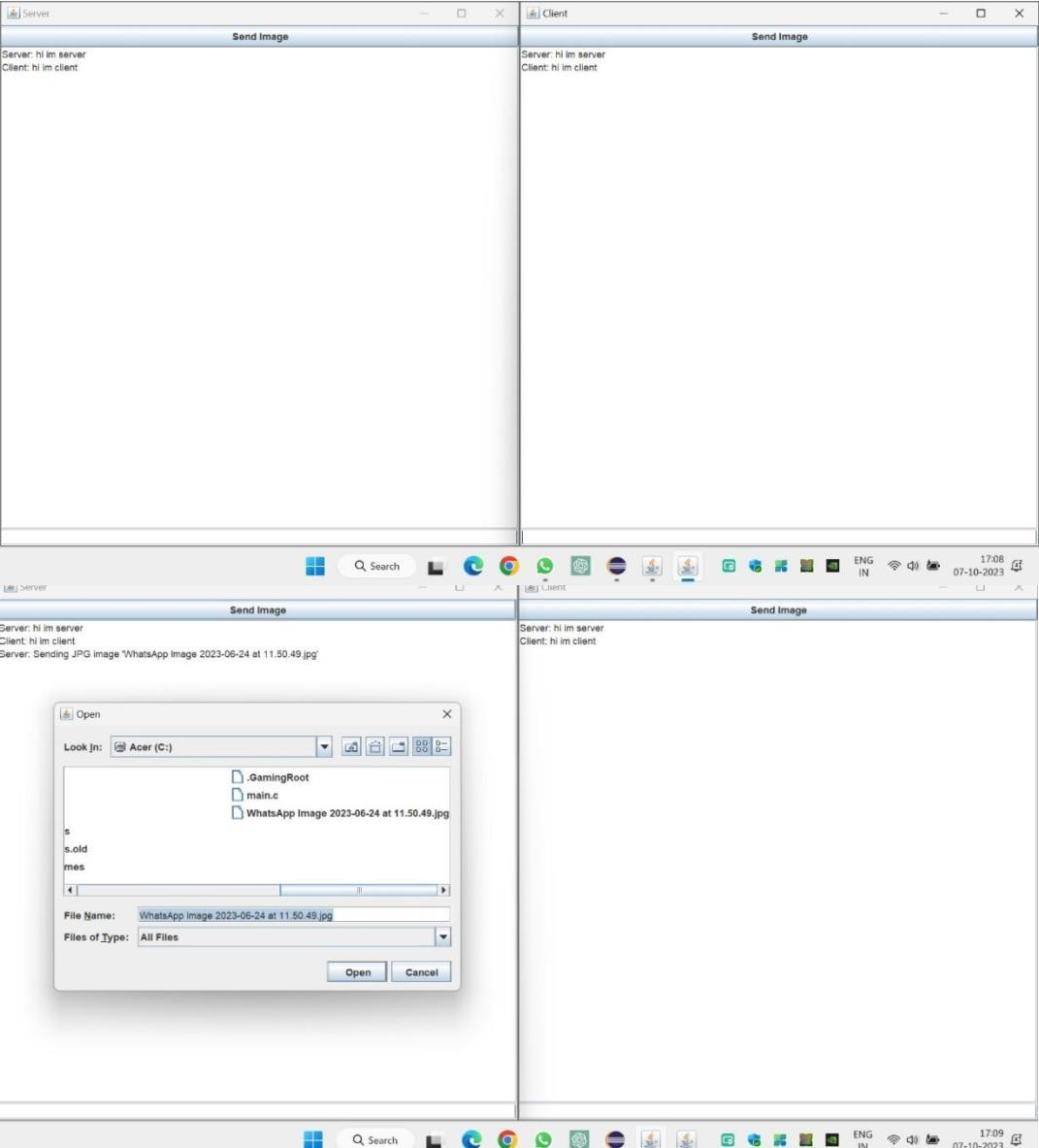
public static void main(String[] args) { SwingUtilities.invokeLater(new Runnable() {

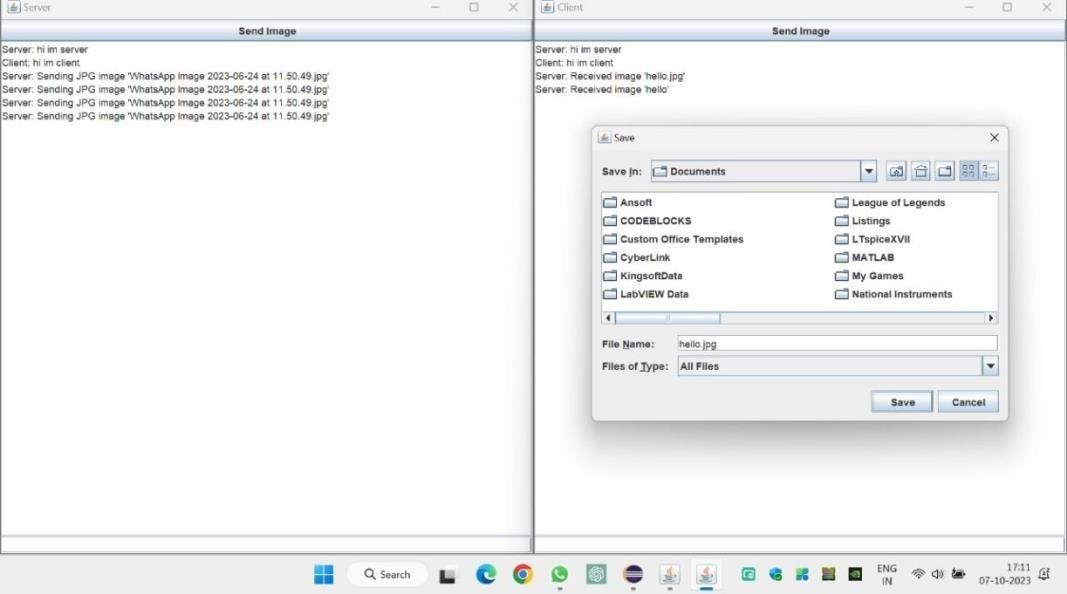
public void run() {

new Client().setVisible(true);

}});}

}

SCREENSHOTS:-



# RESULT:-

In this project, TCP (Transmission Control Protocol) is used for communication between the server and the client

1. Socket Communication (Networking):
   * The code uses Java's built-in `java.net.Socket` and

`java.net.ServerSocket` classes for establishing socket connections. These classes are part of Java's networking capabilities.

1. Swing (GUI):
   * The code utilizes the Swing framework for creating the graphical user interfaces (GUIs) in both the server and client applications. Swing is a set of GUI components and utilities provided by Java.
2. Multithreading (Concurrency):
   * Both the server and client applications use threads to handle incoming messages in real- time. This demonstrates the use of Java's multithreading capabilities for concurrent execution.
3. Exception Handling:
   * The code incorporates try-catch blocks to handle potential exceptions that may occur during file handling and socket communication. This showcases Java's exception handling mechanism.
4. File I/O (Input/Output):
   * The code demonstrates file input and output operations using

`FileInputStream` and

`FileOutputStream` to read and write image files. This is a core Java I/O concept.

1. Event Handling (AWT and Swing):
   * Event handling is implemented using Java's AWT (Abstract Window Toolkit) and Swing libraries. The `ActionListener` interface is used to handle button clicks, which is a fundamental Java GUI concept.

# Reason to choose TCP for the implementation of the chat server

1. Reliability and Order:
   * In a chat application, it's crucial that messages are reliably delivered and received in the correct order. TCP provides this guarantee.
2. Real-time Communication:
   * TCP is well-suited for real-time applications where low latency and consistent delivery are important. It ensures that messages are promptly delivered.
3. File Transfer (Image Sharing):
   * Since the application includes image sharing, it's important to have a reliable protocol. TCP ensures that images are transmitted accurately without loss of data.
4. Stability in Unreliable Networks:
   * TCP is designed to work in networks that may have packet loss, jitter, or varying latency. This is especially important for maintaining a stable connection in the context of online communication.
5. Guaranteed Delivery:
   * TCP guarantees that data sent by one side is received by the other. This is crucial for applications where missing messages or data corruption could lead to misunderstandings.

Overall, TCP is chosen for its reliability, order preservation, and suitability for real-time communication, all of which are essential for a chat application with image sharing functionality.