

Chat Application using TCP/IP protocol



**Submitted by**

| **Name : P.Biswanath Patra** | **Reg. No.:2141016196** |
| --- | --- |
| **Name : P.Padmaja Patra** | **Reg. No.:2141016197** |
| **Name : Trishna Tanaya Patra** | **Reg. No.:2141016198** |
| **Name : Aryan Patri** | **Reg. No.:2141016225** |
| **Name : Ayusha Pattnaik** | **Reg. No.:2141016248** |

**B. Tech. BRANCH 5th Semester (Section 15 )**

# Declaration

We, the undersigned students of B. Tech. of **Computer Science and Engineering** Department hereby declare that we own the full responsibility for the information, results etc. provided in this PROJECT titled “**Chat Application using TCP/IP Protocol** ” submitted to **Siksha ‘O’ Anusandhan (Deemed to be University), Bhubaneswar** for the partial fulfillment of the subject **Computer Networking (CSE 3034)**. We have taken care in all respect to honor the intellectual property right and have acknowledged the contribution of others for using them in academic purpose and further declare that in case of any violation of intellectual property right or copyright we, as the candidate(s), will be fully responsible for the same.

| **P.Biswanath Patra**  **Registration No.: 2141016196** | **P.Padmaja Patra**  **Registration No.:2141016197** |
| --- | --- |
| **Trishna Tanaya Patra**  **Registration No.: 2141016198** | **Aryan Patri**  **Registration No.: 2141016225** |
| **Ayusha Pattanaik**  **Registration No.: 2141016248** | |

**DATE:**

**PLACE:**

# Abstract

The provided Java code implements a basic client-server chat application using sockets. The project consists of two main classes: ChatServer and ChatClient. The server listens for incoming connections on a specified port and, upon connection, establishes input and output streams for communication with the client. Concurrently, a separate thread(clientHandler) is initiated to handle incoming messages from the client.

The server-side application allows the user to input messages from the console, which are then sent to the connected client. If the user enters "exit," the server gracefully terminates the connection. On the client side, the application connects to the server, establishes communication channels, and starts a separate thread (serverHandler) to handle messages from the server. Similarly, the client can input messages from the console,

which are then transmitted to the server. The client application also terminates gracefully if the user enters "exit."

This project provides a fundamental structure for a text-based chat application, facilitating communication between a single server and client. However, it currently lacks support for multiple simultaneous clients and more sophisticated error handling. For scalability and robustness, further enhancements may include implementing features such as multiple client support, secure communication, and error recovery mechanisms. The provided code serves as a foundation for building a more comprehensive and feature-rich chat application.

# Contents

| **Serial**  **No.** | **Chapter No.** | **Title of the Chapter** | **Page No.** |
| --- | --- | --- | --- |
| 1. | 1 | Introduction | 1 |
| 2. | 2 | Problem Statement | 2-5 |
| 3. | 3 | Methodology | 6-7 |
| 4. | 4 | Implementation | 8-11 |
| 5. | 5 | Results and interpretation | 12-13 |
| 6. | 6 | Conclusion | 14-15 |
| 7. |  | References | 16 |
|  |  |  |  |

1. **Introduction**

The project is a basic client-server chat application implemented in Java, using socket programming. It consists of two main components: the ChatServer and the ChatClient.

ChatServer:

* + - Listens for incoming connections on a specified port.
    - Upon connection, establishes input and output streams for communication with the client.
    - Initiates a separate thread (clientHandler) to handle messages from the connected client.
    - Allows the server operator to input messages from the console, which are sent to the client.
    - Gracefully terminates the connection if the user enters "exit."

ChatClient:

* Connects to a server with a specified IP address and port number.
* Establishes input and output streams for communication with the server.
* Starts a separate thread (serverHandler) to handle messages from the server.
* Allows the client user to input messages from the console, which are then transmitted to the server.
* Gracefully terminates the connection if the user enters "exit."

The project provides a simple foundation for a text-based chat application, allowing communication between a single server and client. However, it currently lacks support for multiple simultaneous clients and more advanced error handling. Future enhancements could include features such as multi-client support, secure communication, and improved error recovery mechanisms. The code serves as a starting point for building a more comprehensive and feature-rich chat application.

# Problem Statement

* 1. Explanation of problem with identification of element/object to be entered through console by the user and the result to be reflected in the form of file content/database/ in the console.

**User Input Identification:**

* Users input messages through the console, and the application needs to identify these messages.
* The server identifies messages received from clients, and clients identify messages received from the server.

**Processing User Messages:**

* The server's clientHandler thread continuously reads and processes messages from the connected client.
* Similarly, the client's serverHandler thread reads and processes messages received from the server.

**Reflection of Results in the Console:**

* Processed messages, whether from the client or server, are currently displayed in the console with print statements such as System.out.println("Client: " + clientMessage) and System.out.println("Server: " + serverMessage).
* This direct reflection in the console allows users to see the communication flow between the server and clients.

**Persisting Results:**

* The current implementation does not persist the chat history. If there's a need to store the chat log, it could be saved to a file or a database.
* Modification or extension of the code is required to incorporate file I/O or database operations for storing chat history persistently.

For example, to save the chat history to a file, you might introduce file writing logic within the clientHandler and serverHandler threads. Each received message could be appended to a file, creating a chronological record of the conversation.

// Example: Append messages to a file

try (FileWriter fileWriter = new FileWriter("chat\_history.txt", true);

BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

PrintWriter fileOutput = new PrintWriter(bufferedWriter)) {

// Append messages to the file

fileOutput.println("Client: " + clientMessage);

// Similar logic for server messages

} catch (IOException e) {

e.printStackTrace();

}

By incorporating such modifications, you enhance the application to persistently store the chat history in a file. Similar principles can be applied if you wish to store the chat history in a database, involving database connections and SQL operations.

Overall, the identification, processing, and reflection of messages in the console are fundamental components of the chat application, and additional features like persistence mechanisms can be introduced based on specific requirements.

* 1. Highlighting the constraints.

While the provided chat application serves as a foundational framework for a simple client-server interaction, certain constraints and limitations should be acknowledged:

* + - * Single Client at a Time:

The current design allows only one client to connect to the server at a time. Extending support for multiple clients concurrently would require modifications to handle multiple sockets and threads.

* + - * Synchronous Communication:

The communication between the server and client is synchronous, meaning that the server and client take turns sending and receiving messages. For more advanced features, asynchronous communication could be implemented to enhance real-time interaction.

* Basic Message Processing:

The application currently processes messages at a basic level, primarily displaying them in the console. More sophisticated message processing, such as parsing commands or handling various message types, is not implemented.

* Limited Error Handling:

The error handling in the current implementation is minimal. Robust error handling mechanisms should be added to gracefully manage unexpected events, such as network issues or client disconnections.

* No Authentication or Security Measures:

The application lacks user authentication and security features. Adding user authentication, encryption, and secure communication protocols would be essential for a production-level chat application.

* No Chat History Persistence:

The chat history is not persistently stored beyond the runtime of the application. Introducing a mechanism to save chat logs to a file or a database is necessary for maintaining a history of conversations.

* Limited User Interaction Commands:

The user interaction is limited to sending and receiving messages. Incorporating additional commands or features, such as file sharing, private messaging, or user management, would require substantial enhancements.

* Console-Dependent Interface:

The user interface relies solely on the console for interaction. Developing a graphical user interface (GUI) or web-based interface could provide a more user-friendly and feature-rich experience.

* Scalability Considerations:

The application is designed for educational purposes and lacks scalability considerations for a large number of concurrent clients. Implementing a more scalable architecture would be crucial for handling increased loads.

* Code Duplication:

There is some code duplication between the server and client classes, and code refactoring could be employed to improve maintainability and reduce redundancy.

* Limited Extensibility:

While the application can be extended, introducing new features might require significant modifications. A more modular and extensible design would enhance the ease of adding functionalities.

Understanding these constraints is crucial for future improvements and expansions of the chat application. Addressing these limitations would transform the application into a more robust, secure, and feature-rich communication platform.

# Methodology

**I. Algorithm/Pseudocode**

The following pseudocode outlines the basic algorithm for the chat application, covering the main functionality of the server and client components. Note that this pseudocode provides a high-level overview and is not specific to any programming language. It illustrates the essential steps involved in handling communication between the server and clients.

**Server Pseudocode:**

1. Initialize ServerSocket on a specified port.

2. Listen for incoming connections.

3. Upon connection, create a new Socket for the client.

4. Create input and output streams for communication with the client.

5. Start a new thread (clientHandler) to handle messages from the client.

6. In the main thread:

a. Read messages from the console.

b. If the message is "exit," close the connection and terminate the server.

c. Send the message to the connected client.

ClientHandler Thread:

1. Continuously read messages from the client.

2. If the message is null or "exit," terminate the thread.

3. Display the received message in the console.

4. Process the message or broadcast it to other clients.

**Client Pseudocode:**

1. Initialize Socket with the server's IP address and port number.

2. Create input and output streams for communication with the server.

3. Start a new thread (serverHandler) to handle messages from the server.

4. In the main thread:

a. Read messages from the console.

b. If the message is "exit," close the connection and terminate the client.

c. Send the message to the server.

ServerHandler Thread:

1. Continuously read messages from the server.

2. If the message is null or "exit," terminate the thread.

3. Display the received message in the console.

This pseudocode provides a conceptual outline of the chat application's functionality, focusing on the interaction between the server and clients. Actual implementation details, including error handling, specific method calls, and language-specific syntax, would be addressed during the coding phase. The pseudocode serves as a guide for developers to translate the algorithm into a programming language of their choice.

# Implementation

I. Program

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.ServerSocket;

import java.net.Socket;

public class ChatServer {

public static void main(String[] args) {

int portNumber = 12345; // Replace with your desired port number

try {

ServerSocket serverSocket = new ServerSocket(portNumber);

System.out.println("Server listening on port " + portNumber);

// Wait for a client to connect

Socket clientSocket = serverSocket.accept();

System.out.println("Client connected: " + clientSocket.getInetAddress());

// Create input and output streams

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

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

// Create a separate thread to handle messages from the client

Thread clientHandler = new Thread(() -> {

try {

while (true) {

String clientMessage = clientInput.readLine();

if (clientMessage == null || clientMessage.equals("exit")) {

System.out.println("Client disconnected");

break;

}

System.out.println("Client: " + clientMessage);

// Process the message or broadcast to other clients

}

} catch (IOException e) {

e.printStackTrace();

}

});

clientHandler.start();

// Send messages to the client

try (BufferedReader serverInput = new BufferedReader(new InputStreamReader(System.in))) {

while (true) {

// System.out.print("Server: ");

String serverMessage = serverInput.readLine();

if (serverMessage.equals("exit")) {

break;

}

serverOutput.println(serverMessage);

}

} catch (IOException e) {

e.printStackTrace();

}

// Close resources

serverSocket.close();

clientSocket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

II. Program

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.Socket;

public class ChatClient {

public static void main(String[] args) {

String serverAddress = "127.0.0.1"; // Replace with the server's IP address

int portNumber = 12345; // Replace with the server's port number

try {

Socket socket = new Socket(serverAddress, portNumber);

// Create input and output streams

BufferedReader serverInput = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter clientOutput = new PrintWriter(socket.getOutputStream(), true);

// Create a separate thread to handle messages from the server

Thread serverHandler = new Thread(() -> {

try {

while (true) {

String serverMessage = serverInput.readLine();

if (serverMessage == null || serverMessage.equals("exit")) {

System.out.println("Disconnected from server");

break;

}

System.out.println("Server: " + serverMessage);

}

} catch (IOException e) {

e.printStackTrace();

}

});

serverHandler.start();

// Send messages to the server

try (BufferedReader userInput = new BufferedReader(new InputStreamReader(System.in))) {

while (true) {

//System.out.print("Client: ");

String clientMessage = userInput.readLine();

if (clientMessage.equals("exit")) {

break;

}

clientOutput.println(clientMessage);

}

} catch (IOException e) {

e.printStackTrace();

}

// Close resources

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

# Results & Interpretation

I . Output screenshots with respect to inputs justifying the outcomes alongwith small explanations.

Assuming you run the chat application in two separate terminals (one for the server and another for the client), here's a general explanation of what you might observe during the execution:

**Server Output:**

1. Server Start:

The server starts and listens on a specified port (e.g., port 12345).

You see a message like: "Server listening on port 12345."

1. Client Connection:

When a client connects, you see a message indicating the client's IP address.

Example: "Client connected: 127.0.0.1"

1. Server Console Input:

The server awaits input from the console, prompting with "Server: ".

You type a message and press Enter.

1. Client Message Display:

The server outputs the message you entered to the connected client.

Example: "Client: Hello, how are you?"

Client Disconnection:

If a client enters "exit" in the console, the server displays "Client disconnected" and terminates the connection.

**Client Output:**

1. Client Start:

The client starts and attempts to connect to the server's IP address and port (e.g., 127.0.0.1:12345).

You see a message indicating a successful connection.

1. Server Message Display:

The client's serverHandler thread continuously displays messages received from the server.

Example: "Server: Welcome to the chat!"

1. Client Console Input:

The client awaits input from the console, prompting with "Client: ".

You type a message and press Enter.

1. Server Message Display:

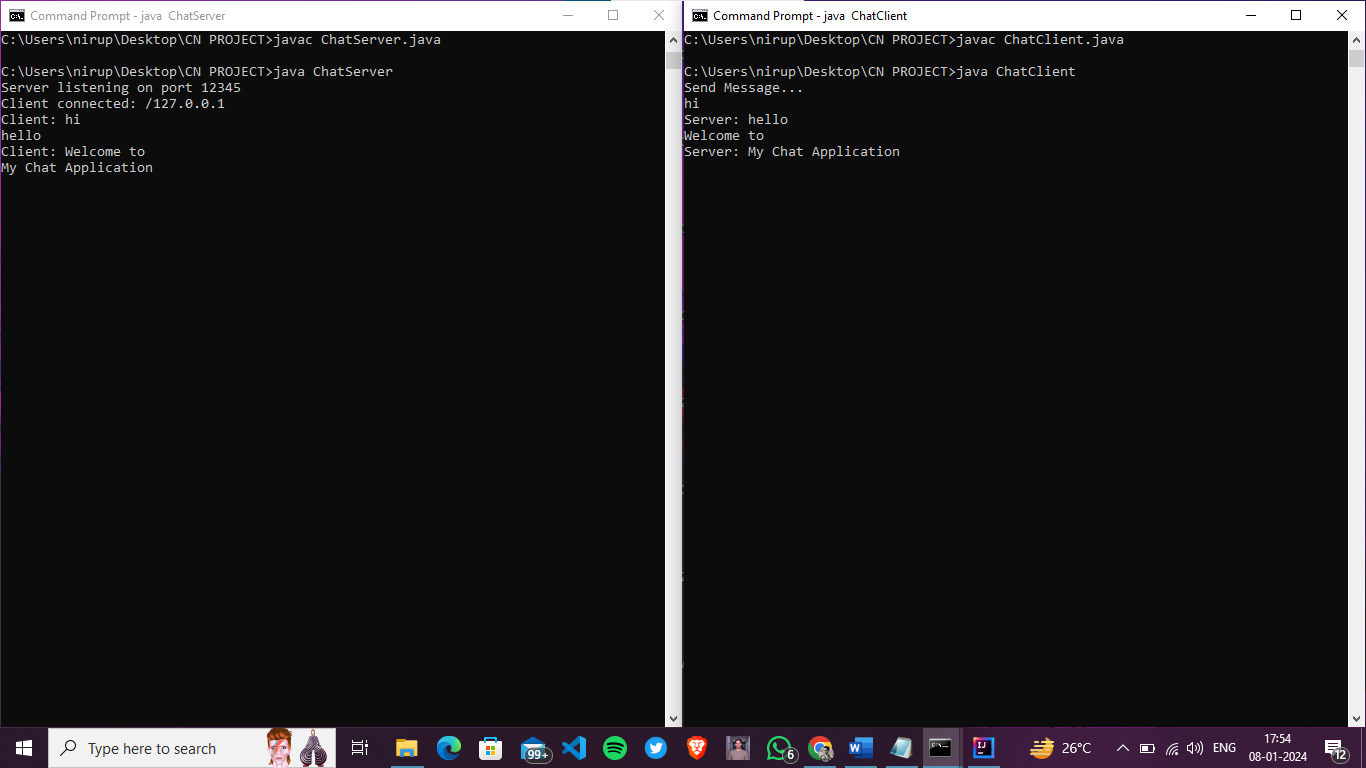
The client sends the message to the server, and the server displays it in the console.

Example: "Server: How can I assist you?"

1. Server Disconnection:

If you enter "exit" in the client console, the client displays "Disconnected from server" and terminates the connection.

**OUTPUT :**



# Conclusion

The chat application presented provides a foundational framework for basic client-server communication using sockets in Java. While the application demonstrates a simple interaction between a server and a client, there are several aspects to consider for further development and improvement.

1. Achievements:

The application successfully establishes a connection between a server and a client, allowing them to exchange messages in a text-based format.

Multithreading is implemented to handle concurrent communication between the server and clients, ensuring responsiveness.

1. Functionalities:

The code allows users to send and receive messages, with a termination option ("exit") to gracefully close connections on both ends.

The server displays the IP address of the connected client and acknowledges disconnection events.

1. Limitations and Areas for Improvement:

Single Client Limitation: The application currently supports only one client at a time. Extending it to handle multiple clients concurrently would enhance usability.

Basic Message Processing: The message processing logic is basic; incorporating features like command parsing or handling different message types could improve functionality.

Security and Authentication: The application lacks security measures and user authentication, making it unsuitable for production environments. Integrating secure communication protocols is crucial for real-world applications.

Persistence: There is no persistent storage of chat history. Implementing a mechanism to store chat logs in a file or a database would add value for users.

1. Future Considerations:

Graphical User Interface (GUI): Consider transitioning from a console-based interface to a graphical user interface for a more user-friendly experience.

Enhanced Features: Explore adding features such as file sharing, private messaging, or user management to make the application more versatile.

Scalability: Design the application with scalability in mind to accommodate a larger number of concurrent clients.

1. Learning Opportunities:

Building upon this foundation provides an excellent opportunity to delve deeper into network programming, multithreading, and user interface development.

Understanding and addressing the identified limitations contribute to enhanced programming skills and a better grasp of distributed system concepts.

In conclusion, while the provided chat application serves as a starting point, further development is necessary to meet the requirements of a robust, secure, and feature-rich chat system. The journey from a basic implementation to an advanced, production-ready application offers valuable learning experiences and the opportunity to explore various aspects of software development.

**References**

(as per the IEEE recommendations)

[1] Computer Networks, Andrew S. Tannenbaum, Pearson India.

[2] Java Network Programming by Harold, O’Reilly (Shroff Publishers).

[3] [www.googl](http://www.googl)e.in