



Open Ended Lab: TCP Chatroom in Information Security

By

Zia Ur Rehman

engrziurrehman.kicsit@gmail.com

in LinkedIn,  GitHub, ResearchGate,  PyPI



December 18, 2024

You can find the Open Ended Lab on GitHub at the following link: [Open Ended Lab on GitHub](#).

Contents

1	Introduction	3
2	What is TCP?	3
2.1	Key Features of TCP	3
2.2	Why TCP is Secure	3
2.3	Mathematical Foundations of TCP	4
2.4	Applications	4
2.5	Scenario	5
3	Objectives	5
4	Step-by-Step Guidance	5
4.1	Setup Environment	5
4.2	Server Code Implementation	5
4.3	Code Blocks and Explanation	5
4.4	Flow of Code and Approach	7
4.5	Client Code Implementation	8
4.6	Running the Application	10
5	How It Works	10
6	Dry Run Example	10
6.1	Scenario: Three Clients (Alice, Bob, and Charlie)	10
7	Applications	11
8	Conclusion	11

1 Introduction

In the modern digital era, secure communication is a cornerstone of information security. Chatrooms, built on the TCP (Transmission Control Protocol), serve as excellent learning tools for understanding how reliable, ordered, and error-checked communication is achieved over networks. This open-ended lab explores the design and implementation of a TCP-based chatroom system to highlight the practical applications of secure communication protocols and concurrent programming.

2 What is TCP?

Transmission Control Protocol (TCP) is a core protocol of the Internet Protocol Suite. It provides reliable, ordered, and error-checked delivery of data between applications communicating over a network. TCP is widely used in various real-world applications, including web browsing, email, file transfer, and chat systems.

2.1 Key Features of TCP

- **Reliability:** Ensures all packets are delivered and reassembled in the correct order. Example: Consider sending a file divided into three parts. Even if parts 2 and 3 arrive before part 1 due to network delays, TCP ensures the receiver reassembles them as part 1, part 2, part 3 before processing.
- **Error Checking:** Uses checksums to detect and correct errors during data transmission. Example: During transmission, if a packet's data is altered due to noise, the receiver calculates the checksum and compares it with the sender's checksum. Mismatches trigger a retransmission request.
- **Flow Control:** Manages data flow to prevent overwhelming the receiver. Example: If a server sends data faster than a client can process, TCP adjusts the speed dynamically by limiting the window size of unacknowledged packets.
- **Congestion Control:** Adjusts the transmission rate based on network traffic conditions. Example: In a busy network, TCP slows down data transmission to reduce congestion, similar to vehicles reducing speed in heavy traffic to avoid collisions.
- **Connection-Oriented:** Establishes a connection between the sender and receiver before transmitting data. Example: Before sending a message, TCP completes a three-way handshake to ensure both the sender and receiver are ready for data exchange.

2.2 Why TCP is Secure

TCP incorporates several mechanisms to enhance security and reliability:

- **Three-Way Handshake:** Ensures a connection is properly established before data transmission. Example: When Alice initiates a connection with Bob, the handshake follows these steps: Alice sends a SYN (synchronize) packet, Bob responds with a SYN-ACK, and Alice replies with an ACK to confirm readiness.

- **Acknowledgments:** Confirms receipt of data packets. Example: If Bob receives a data packet from Alice, he sends an acknowledgment (ACK) back to Alice to confirm successful delivery. If no ACK is received, Alice retransmits the packet.
- **Sequence Numbers:** Maintains the order of packets, preventing data corruption or duplication. Example: For a message split into packets with sequence numbers 101, 102, and 103, the receiver uses these numbers to arrange them correctly, even if they arrive out of order.
- **Error Detection:** Detects transmission errors using checksums. Example: If a checksum for a packet calculated by Bob does not match the checksum sent by Alice, Bob discards the packet and requests a retransmission.

2.3 Mathematical Foundations of TCP

TCP uses various mathematical concepts to ensure reliable communication:

- **Checksums:** TCP computes a checksum by summing all 16-bit words in a segment and verifying it at the receiver. Example: For a data segment "1011" and "1100" (in binary), the checksum is computed as:

$$1011 + 1100 = 10111 \quad (\text{wrap-around sum} : 0111)$$

The checksum "0111" is appended to the data. At the receiver, adding data and checksum should yield zero, confirming integrity.

- **Sequence Numbers:** Represent the byte order of data, ensuring packets are re-assembled correctly. Example: If sequence numbers 1, 2, and 3 correspond to "Hello", "World", and "!", the receiver uses the numbers to combine them into "Hello World!", even if they arrive in the order 3, 1, 2.
- **Round-Trip Time (RTT):** Measures the time taken for a signal to travel to the receiver and back, helping in adjusting the retransmission timeout. Example: If Alice sends a packet to Bob and receives an acknowledgment in 200ms, the RTT is 200ms. TCP uses this to set a timeout slightly higher than RTT to avoid premature retransmissions.

2.4 Applications

- **Corporate Communication:** TCP-based chat systems are used for secure internal communication within organizations.
- **Customer Support:** Live chat support systems for troubleshooting and assistance.
- **Real-time Collaboration:** Used in collaborative tools for simultaneous interaction across remote teams.
- **Gaming:** Enables real-time multiplayer communication in online games.

2.5 Scenario

Consider an organization that needs a simple yet secure chatroom for internal discussions. Each participant should have a unique nickname, and messages exchanged must be reliable and delivered in the correct order. The organization requires a scalable system capable of handling multiple participants simultaneously while maintaining efficient communication.

3 Objectives

1. Learn to build a TCP-based chatroom server and client.
2. Understand the role of threading for handling multiple clients concurrently.
3. Explore the basics of reliable communication protocols like TCP.
4. Analyze message broadcasting and its efficiency.
5. Experiment with different use cases and potential improvements.

4 Step-by-Step Guidance

4.1 Setup Environment

Before proceeding, ensure you have Python installed on your system along with a text editor or IDE.

4.2 Server Code Implementation

Create a file named `Server.py` and use the following code: The server code is divided into logical blocks, each serving a specific purpose. Below, we provide the code sections, explain their roles, and describe the overall flow and approach.

4.3 Code Blocks and Explanation

1. Importing Libraries and Setting Up Connection Data

Listing 1: Importing Libraries and Setting Connection Data

```
import socket
import threading

# Connection Data
host = '127.0.0.1'
port = 55555
```

Purpose: This section imports the required libraries. The `socket` library is used to create the network connection, while `threading` enables handling multiple clients simultaneously. The variables `host` and `port` define the server's address and the port it listens on.

2. Starting the Server

Listing 2: Starting the Server

```
# Starting Server
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((host, port))
server.listen()

clients = []
nicknames = []
```

Purpose: Here, a TCP socket is created using `socket.AF_INET` (IPv4) and `socket.SOCK_STREAM` (TCP). The server is bound to the `host` and `port`, and `listen()` makes it ready to accept incoming connections. The `clients` and `nicknames` lists are initialized to store active client connections and their nicknames.

3. Broadcasting Messages

Listing 3: Broadcasting Messages

```
def broadcast(message):
    for client in clients:
        client.send(message)
```

Purpose: This function takes a `message` as input and sends it to all connected clients by iterating over the `clients` list. This ensures that every participant in the chatroom receives the message.

4. Handling Individual Clients

Listing 4: Handling Individual Clients

```
def handle(client):
    while True:
        try:
            message = client.recv(1024)
            broadcast(message)
        except:
            index = clients.index(client)
            clients.remove(client)
            client.close()
            nickname = nicknames[index]
            broadcast(f"{nickname} left!".encode('ascii'))
            nicknames.remove(nickname)
            break
```

Purpose: The `handle()` function manages communication with a specific client. It continuously listens for incoming messages using `recv(1024)` (buffer size of 1024 bytes). If an error occurs (e.g., client disconnects), the client is removed from the `clients` list, and a notification is broadcast to others.

5. Receiving New Connections

Listing 5: Receiving New Connections

```
def receive():
    while True:
        client, address = server.accept()
        print(f"Connected with {str(address)}")

        client.send('NICK'.encode('ascii'))
        nickname = client.recv(1024).decode('ascii')
        nicknames.append(nickname)
        clients.append(client)

        print(f"Nickname is {nickname}")
        broadcast(f"{nickname} joined!".encode('ascii'))
        client.send('Connected to server!'.encode('ascii'))

        thread = threading.Thread(target=handle, args=(client,))
        thread.start()
```

Purpose: This function waits for new clients to connect using `accept()`. Once connected, the client is asked for a nickname, which is stored in the `nicknames` list. The client socket is added to `clients`, and a new thread is created to handle communication with the client.

6. Starting the Server

Listing 6: Starting the Server

```
print("Server is listening...")
receive()
```

Purpose: This starts the `receive()` function, allowing the server to begin accepting connections and managing clients.

4.4 Flow of Code and Approach

1. **Initialize Server:** The server socket is set up and starts listening for connections.
2. **Handle New Connections:** When a client connects, they are prompted for a nickname and added to the active client list.
3. **Concurrent Communication:** Each client is managed in a separate thread using the `handle()` function, ensuring real-time interaction.
4. **Broadcast Messages:** Messages from clients are sent to all participants using the `broadcast()` function.
5. **Error Handling:** Disconnects and errors are gracefully managed by removing the client and notifying others.

This structured approach ensures scalability, reliability, and ease of managing multiple clients simultaneously.

4.5 Client Code Implementation

Create a file named `Client.py` and use the following code:

1. Importing Required Modules

This block imports the necessary modules for networking and multithreading.

Listing 7: Importing Required Modules

```
import socket
import threading
```

- `socket`: Allows for network communication (establishing a TCP connection).
- `threading`: Enables parallel execution of functions (in this case, handling receiving and sending messages concurrently).

2. Setting up Client Socket

This block sets up the client-side socket to connect to the server.

Listing 8: Setting up Client Socket

```
nickname = input("Choose your nickname: ")

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('127.0.0.1', 55555))
```

- `nickname`: Prompts the user to input a nickname for identifying them during the conversation.
- `client`: Creates a socket object using IPv4 (`AF_INET`) and TCP (`SOCK_STREAM`).
- `connect()`: Establishes a connection to the server running on `127.0.0.1` (localhost) on port `55555`.

3. Receive Function (Threaded)

This block handles receiving messages from the server. It runs in a separate thread.

Listing 9: Receive Function (Threaded)

```
def receive():
    while True:
        try:
            message = client.recv(1024).decode('ascii')
            if message == 'NICK':
                client.send(nickname.encode('ascii'))
            else:
                print(message)
        except:
            print("An error occurred!")
            client.close()
            break
```


- `receive()`: Continuously listens for messages from the server.
 - `client.recv(1024)`: Receives a message from the server, up to 1024 bytes.
 - `decode('ascii')`: Decodes the received message from byte format to a string.
- If the message is 'NICK', it sends the user's nickname to the server for identification.
- Any other message is displayed in the terminal.
- **except**: If any error occurs (e.g., server connection loss), it prints an error message and closes the socket.

4. Write Function (Threaded)

This block handles sending messages to the server. It runs in a separate thread as well.

Listing 10: Write Function (Threaded)

```
def write():
    while True:
        message = f"{nickname}: {input(' ')}"
        client.send(message.encode('ascii'))
```

- `write()`: Continuously prompts the user to input messages and sends them to the server.
 - `input(' ')`: Waits for the user to enter a message.
 - `message.encode('ascii')`: Encodes the message into byte format before sending it to the server.

5. Starting Threads

This block initializes and starts the threads for receiving and writing messages concurrently.

Listing 11: Starting Threads

```
receive_thread = threading.Thread(target=receive)
receive_thread.start()

write_thread = threading.Thread(target=write)
write_thread.start()
```

- `threading.Thread()`: Creates separate threads for the `receive()` and `write()` functions.
- `start()`: Starts both threads, allowing the program to handle receiving and sending messages simultaneously.

Flow of the Program

- The client prompts the user for a nickname and establishes a TCP connection with the server.
- Two threads are created:
 - `receive_thread`: Continuously listens for incoming messages from the server and displays them.
 - `write_thread`: Allows the user to input messages and sends them to the server.
- The client sends its nickname if the server asks for it, and messages can be sent and received concurrently in real-time.
- This approach allows the client to interact with the server without blocking the input or the message receiving process, ensuring smooth communication.

4.6 Running the Application

1. Start the server by running `Server.py` in one terminal or command prompt.
2. Open multiple terminals or command prompts and run `Client.py` in each to simulate different clients.
3. Observe the message broadcasting and nicknames as clients join, communicate, and leave the chatroom.

5 How It Works

- The server listens for incoming connections and assigns a thread to each client for concurrent communication.
- Clients exchange messages with the server, which then broadcasts them to all connected clients.
- TCP ensures reliable delivery, ordered sequence, and error-free communication of messages.

6 Dry Run Example

6.1 Scenario: Three Clients (Alice, Bob, and Charlie)

1. **Alice connects:** Server prompts nickname and broadcasts "Alice joined!"
2. **Bob connects:** Server prompts nickname and broadcasts "Bob joined!"
3. **Charlie connects:** Server prompts nickname and broadcasts "Charlie joined!"
4. **Message Exchange:** Alice sends "Hello everyone!". Server broadcasts:
 - To Bob: "Alice: Hello everyone!"

- To Charlie: "Alice: Hello everyone!"

5. **Client Disconnects:** Charlie disconnects. Server broadcasts "Charlie left!"

7 Applications

- Real-time communication in secure environments.
- Educational tools to understand TCP and multi-threading.
- Basis for implementing more complex chat systems with encryption.

8 Conclusion

This lab provides hands-on experience with TCP-based chatroom systems, illustrating the reliability of TCP and the power of multi-threading. It lays a foundation for exploring advanced topics like secure communication, encryption, and scalability in networking.