

Ethan Shellem-Prince Assignment 2 Report

by Ethan Shellem-Prince

Submission date: 14-May-2025 02:50PM (UTC+0100)

Submission ID: 259232845

File name: 31725_Ethan_Shellem-Prince_Ethan_Shellem-Prince_Assignment_2_Report_664977_1276832958.docx (1.07M)

Word count: 895

Character count: 5143

Socket Programming and Concurrency Report – Assignment 2

Module Code: ACCA6028

Author: Ethan Shellem-Prince

Table of Contents

1. Introduction	3
2. Design Overview	3
3. Server Architecture	3
4. Client Functionality	5
5. Key Features Implemented	5
6. Development Decisions	5
7. Testing and Validation	6
8. Limitations and Future Improvements	7
9. Conclusion	7
10. References	8

Table of Figures

1. Figure 1	3
2. Figure 2	3
3. Figure 3	3
4. Figure 4	5
5. Figure 5	5
6. Figure 6	5
7. Figure 7	6

1. Introduction

This report documents the development of a networked chat application that was built using C++ and Winsock on the Windows platform [1]. The goal was to demonstrate a working knowledge of socket programming, concurrency, and client-server communication by implementing a basic but functional multiple chat system. The application allows multiple clients to connect to a server, send and receive real-time messages, and disconnect easily.

2. Design Overview

The chat system follows a standard client-server architecture. A central server listens on a fixed port (54000) and handles multiple simultaneous client connections. Each client connects to the server using TCP and is managed on the server through a dedicated thread. This ensures that blocking operations from one client do not affect others. This design choice ensures responsiveness and scalability without requiring a complex system.

3. Server Architecture

The server is implemented using Winsock's socket API [2]. On startup, it initializes the Winsock library and binds a socket to a specific port (54000). It then enters a loop, continuously accepting incoming client connections.

Figure 1: Server accepts loop and thread-per-client model for handling incoming connections.

```
int main() {
    WSADATA wsaData;
    WSASocket(2, 2, &wsaData);

    SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in serverAddr{};
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    serverAddr.sin_port = htons(54000);

    bind(serverSocket, (sockaddr*)&serverAddr, sizeof(serverAddr));
    listen(serverSocket, SOMAXCONN);

    std::cout << "Server started on port 54000.\n";

    while (true) {
        sockaddr_in clientAddr{};
        int clientSize = sizeof(clientAddr);
        SOCKET clientSocket = accept(serverSocket, (sockaddr*)&clientAddr, &clientSize);

        {
            std::lock_guard<std::mutex> lock(clientMutex);
            clients.push_back(clientSocket);
        }

        std::cout << "New client connected.\n";
        std::thread clientThread(HandleClient, clientSocket);
        clientThread.detach();
    }

    closesocket(serverSocket);
    WSACleanup();
    return 0;
}
```

For each new client, the server creates a new thread to handle communication with that client independently. Messages received from any client are broadcast to all other connected clients.

To manage concurrency, a `std::mutex` is used to guard shared access to the list of connected clients. Each client socket is stored in a vector, and the server iterates through this list to send messages. Mutex protection prevents race conditions when multiple threads attempt to modify the list at the same time [3].

Figure 2: BroadcastMessage() function showing mutex-protected message broadcasting to all clients except the sender.

```
void BroadcastMessage(const std::string& message, SOCKET sender) {  
    std::lock_guard<std::mutex> lock(clientMutex);  
    for (SOCKET client : clients) {  
        if (client != sender) {  
            send(client, message.c_str(), message.length(), 0);  
        }  
    }  
}
```

The server also listens for a special `/exit` message from any client, allowing them to disconnect easily. When receiving this, the server closes the client's socket and removes it from the vector.

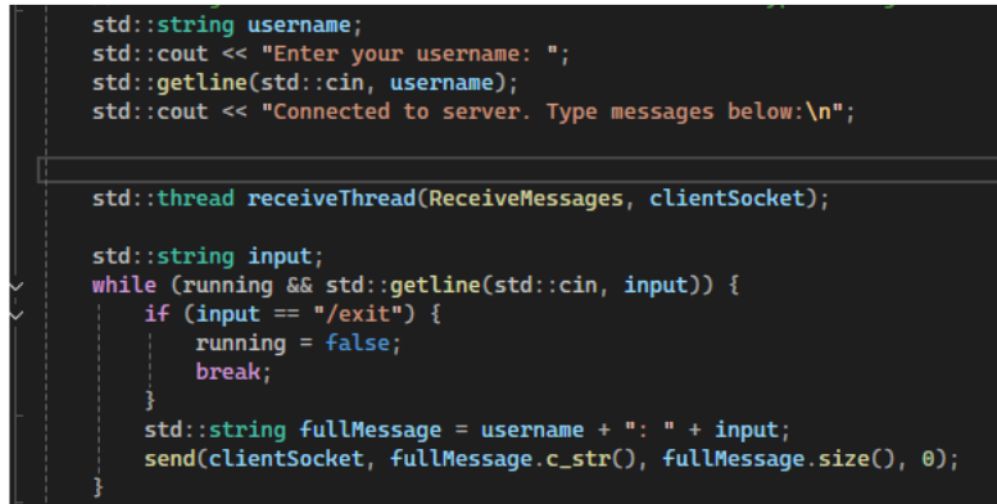
Figure 3: HandleClient() receives messages, handles /exit, and removes disconnected clients.

```
void HandleClient(SOCKET clientSocket) {  
    char buffer[1024];  
    while (true) {  
        int bytesReceived = recv(clientSocket, buffer, sizeof(buffer), 0);  
        if (bytesReceived <= 0) break;  
  
        std::string msg(buffer, bytesReceived);  
  
        std::cout << "[RECEIVED] " << msg << std::endl;  
  
        if (msg.find("/exit") != std::string::npos) {  
            std::cout << "[DISCONNECT COMMAND] Client requested to exit.\n";  
            break;  
        }  
  
        BroadcastMessage(msg, clientSocket);  
    }  
  
    std::lock_guard<std::mutex> lock(clientMutex);  
    closesocket(clientSocket);  
    clients.erase(std::remove(clients.begin(), clients.end(), clientSocket), clients.end());  
    std::cout << "Client disconnected." << std::endl;  
}
```

4. Client Functionality

On startup, the client prompts the user to enter a username, which is then prepended to every message sent.

Figure 4: Client prompts for a username and formats messages with a name prefix.



```
std::string username;
std::cout << "Enter your username: ";
std::getline(std::cin, username);
std::cout << "Connected to server. Type messages below:\n";

std::thread receiveThread(ReceiveMessages, clientSocket);

std::string input;
while (running && std::getline(std::cin, input)) {
    if (input == "/exit") {
        running = false;
        break;
    }
    std::string fullMessage = username + ": " + input;
    send(clientSocket, fullMessage.c_str(), fullMessage.size(), 0);
}
```

The client attempts to connect to the server's IP (localhost, 127.0.0.1) and port 54000. Once connected, it launches a separate thread to receive messages from the server asynchronously while the main thread manages user input.

Users can type messages freely and see responses from other users in real time. If the user types "/exit", the client shuts down the connection and exits cleanly.

5. Key Features Implemented

- Multi-threaded server with thread-per-client model
- Thread-safe message broadcasting using `std::mutex`
- Username capture and message prefixing
- Graceful disconnection via "/exit" command
- Use of blocking `recv()` and `send()` calls in separate threads to simulate non-blocking behavior

6. Development Decisions

Using Winsock on Windows allows access to low-level socket operations and full control over the networking layer. The decision to use a thread-per-client model was made for

simplicity and clarity, given the limited scale of the application. For larger-scale systems, an asynchronous model or use of I/O completion ports would be much more efficient, but for this case, threads provided a reasonable concurrency mechanism.

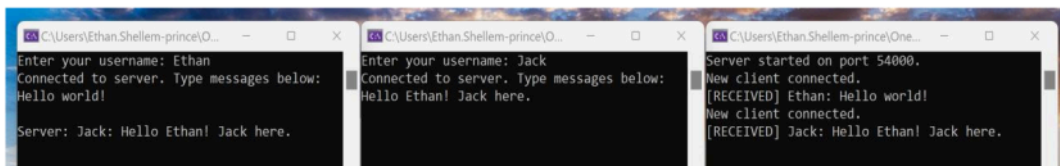
All sockets were managed using TCP, ensuring reliable data transmission and built-in message ordering. Mutexes were preferred over more advanced concurrent containers due to their clarity and suitability for educational purposes.

7. Testing and Validation

The server was run in a dedicated console window, and multiple instances of the client were executed in parallel to simulate multiple users. The following tests were performed:

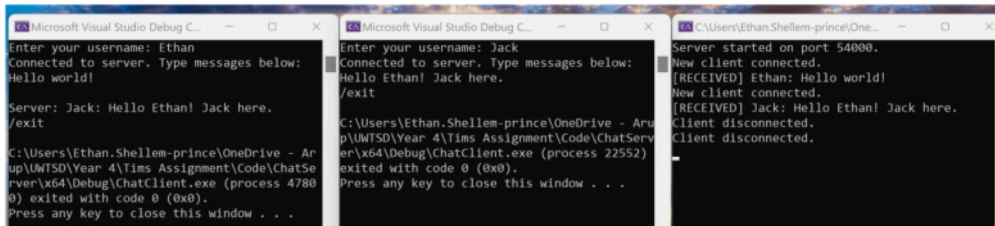
- Connecting two or more clients simultaneously
- Sending messages between clients and verifying correct broadcast

Figure 5: Testing 1



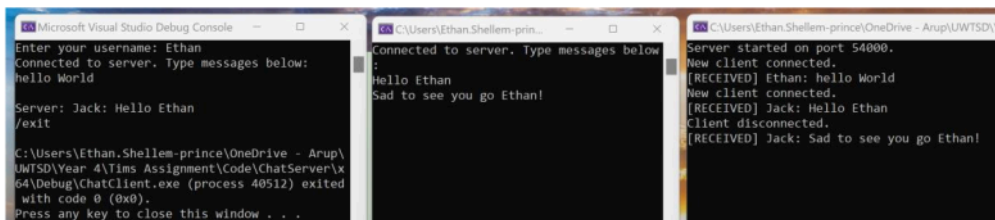
- Entering "/exit" and ensuring the server logs the disconnect

Figure 6: Testing 2



- Observing that remaining clients continue to operate without interruption

Figure 7: Testing 3



No crashes or data corruption were observed during normal usage. The application remained stable with up to 5 clients at once.

8. Limitations and Future Improvements

The current system does not include features such as:

- Displaying a list of currently connected users
- Private messaging or rooms
- Message history or logging
- Encryption or authentication

In future iterations, these features could be added by extending the protocol or integrating a GUI framework. An event-driven architecture could also replace the threading model for better scalability.

9. Conclusion

The chat application successfully meets the requirements outlined in the assignment brief. It demonstrates key skills in network programming, threading, and client-server architecture. All features were built and tested manually using Visual Studio 2022 on Windows. This project provided valuable experience in synchronisation, socket handling, and managing concurrent client connections in C++.

10. References

- [1] Microsoft Docs - Winsock Programming Guide. <https://learn.microsoft.com/en-us/windows/win32/winsock/>
- [2] MSDN - Windows Socket Functions. <https://learn.microsoft.com/en-us/windows/win32/api/winsock2/>
- [3] cppreference.com - C++ Standard Library. <https://en.cppreference.com>

Ethan Shellem-Prince Assignment 2 Report

GRADEMARK REPORT

FINAL GRADE

70/100

GENERAL COMMENTS

A good effort - a limited chat application implemented,

and sensible future work identified. A good application of the core principles.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8
