SOCKET PROGRAMMING AND CONCURRENCY

ACCA6028

Ryan Michael Hopkins (2221956)

Contents

Introduction 2
Application architecture 2
Network and protocol design 2
Multithreading 3
Features 3
Best Practice and difficulties 4
Appendix 4

Introduction

This report discusses the design and development of a chat room application built in C++ using the WinSock2 library. The application uses a client-server architecture, allowing multiple users to connect and interact simultaneously.

Application architecture

The application is designed using a modular approach. This means the program is divided into smaller parts, each with a specific function. These parts are stored in separate header files, which keeps the code organized and easier to understand.

Each header file is responsible for a different feature:

- Log.h handles logging, creating a unique session file and saving it to the disk.
- Network.h manages the network connection, setting up the server using sockets and connecting it to port 54000. It also includes functionality to safely shut down the server.
- ClientManager.h keeps track of all connections. It also provides shared functions such as sending messages to all users (also known as broadcasting) and removing users who disconnect.
- Commands.h processes any commands entered by users and directs them to the correct function

This structure keeps the main server file simple and easy to read, as most of the functionality is handled in these separate files.
The client application follows a similar structure. The network connection logic is in a separate file called ClientNetwork.h, while ClientUI.h handles the user interface, including timestamping messages and handling message reception. For file configuration in Visual Studio, see Figure 1.

Network and protocol design

The application uses TCP (Transmission Control Protocol) for communication between the client and the server. TCP was chosen instead of UDP (User Datagram Protocol) because it is more reliable, ensuring that no data is lost during transmission and that data is delivered in the correct order.

The server is built using the Winsock2 library, which is the standard way to handle network communication on Windows.

Setting up the server involves multiple steps:

1. A socket is created using socket(). This almost acts like the endpoint for communication.
2. The socket is linked to port 54000 using bind(), which tells the operating system to listen for connections.
3. The server starts listening for incoming clients using listen().
4. When a client attempts to connect, accept() is used to establish the connection.

On the client side, the connect() function connects to the server. Once connected, messages are sent as plain text between the client and server using send() and recv()
When a client first connects, they set their nickname, and the server stores this name to identify the user during the session.

Multithreading

Multithreading is used in the server to allow multiple clients to connect and communicate at the same time. Without it, the server would only handle one client at a time, meaning that other clients would have to wait. This is commonly known as blocking I/O.

To solve this, the server creates a new thread for each client using std::thread. Each thread runs independently and manages communication with its assigned client, allowing multiple users to interact with the server.

However, multiple threads accessing shared data can cause issues, for example, if two threads try to modify data at the same time, it can lead to errors or crashes. This is called a data race.

To prevent this, a std::mutex is used to ensure that only one thread can access shared data at a time. A std::lock_guard is used to manage the mutex safely by automatically releasing the lock when it is not needed.

The server also uses std::atomic<bool> variables, such as ServerRunning, to ensure safe values between threads. Without this, multiple threads accessing the same variable could cause a data race, leading to unpredictable behavior such as crashes. A specific crash is mentioned in the Best Practice and Difficulties section.

Features

The application includes several features and commands to connect users together. All commands are identified using a forward slash or / and are handled in the Commands.h header file.

When a user connects, they are prompted to enter a nickname which identifies them throughout the session. Once connected, they can send messages which are broadcast to all other connected users with a timestamp of when the message was sent.

The available commands are:

- /list — displays a list of all currently connected users. See figure 3
- /nick — allows the user to change their nickname. The server checks that the new nickname is not already taken before applying the change. See figure 4
- /stats — displays the number of messages each connected user has sent during the session, see figure 5
- /help — displays a list of all available commands. See figure 6
- /quit — disconnects the user cleanly from the server. See figure 7

The server also includes an idle shutdown timer, if no clients are connected for 20 seconds, the server automatically shuts down, see figure 8 for this. Each server session generates a unique log file named after the session's date and time, which records all messages and connection events.

Best Practice and difficulties

Throughout development, best practices were applied to keep the application stable and maintainable.

Resource management follows the RAII or resource acquisition is initialisation principles. This ensures resources such as sockets, files, and handles are properly released when no longer needed. An example of this is when a client disconnects, their socket is closed and removed from the client list. On server shutdown, all sockets are closed, and WSACleanup() is called to release Winsock resources.

Error handling is applied to all socket operations, including socket(), bind(), listen(), accept() and connect. Return values are checked, and if an error occurs, the program outputs a clear message and exits safely instead of continuing in an unstable state.

To improve memory safety, the receive buffer was implemented using std::vector<char> instead of a fixed-size array. This prevents buffer overflow issues that could lead to crashes.

One notable challenge was implementing a clean server shutdown. Initially, exit(0) was used, which caused crashes by terminating the program abruptly. This was replaced with an atomic bool flag (ServerRunning). When set to false, the server exits its main loop naturally and performs a proper cleanup.

Another issue encountered was a lock caused by incorrect mutex usage. This was resolved by ensuring locks were not held when calling functions that also require the same mutex.

Appendix

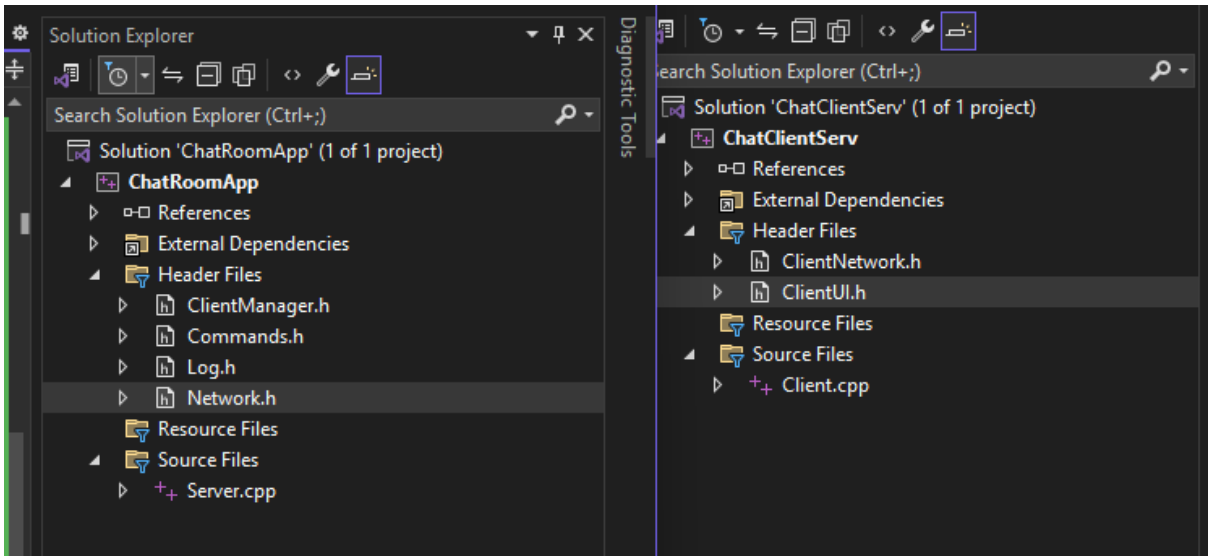

Figure1: architecture and layout within file visual studio for both files

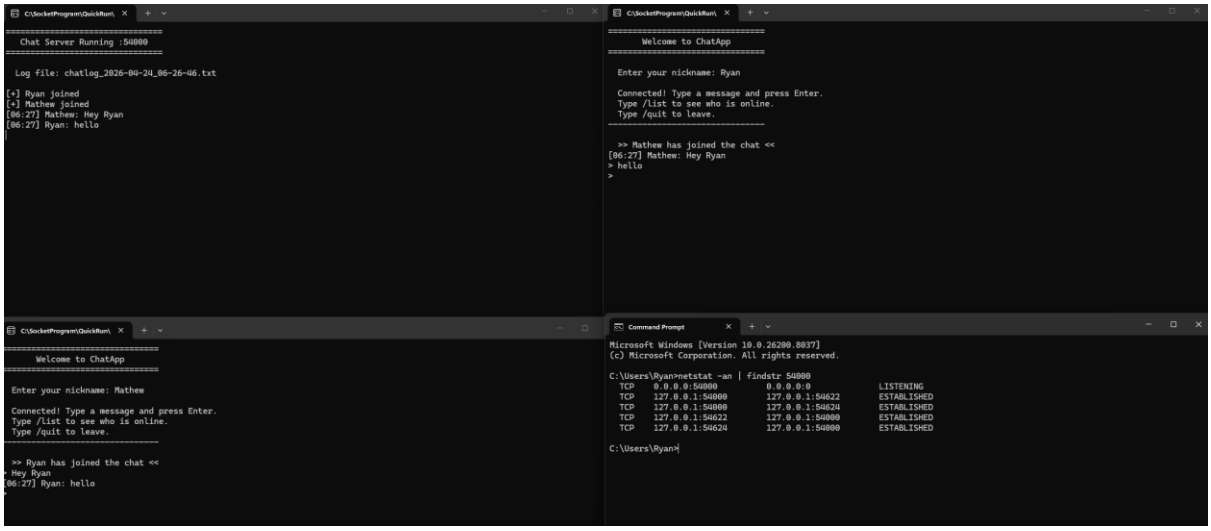

Figure2: Networking between 3 clients and specific finding what ports address to port 54000

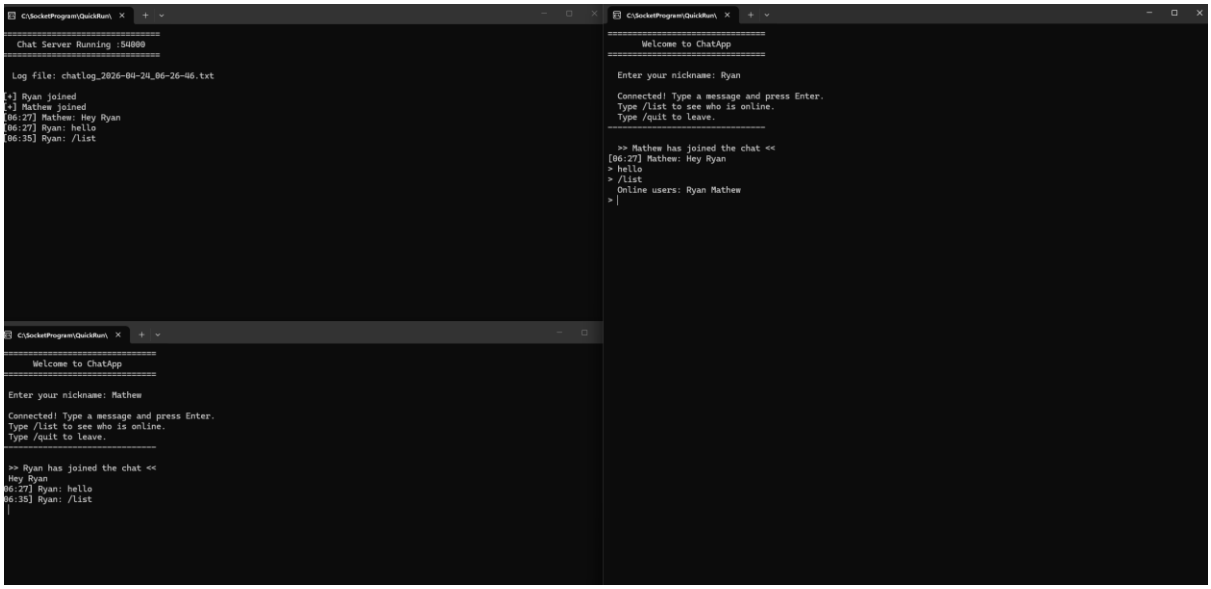

Figure 3: the user calling the /list command which lists all available users.

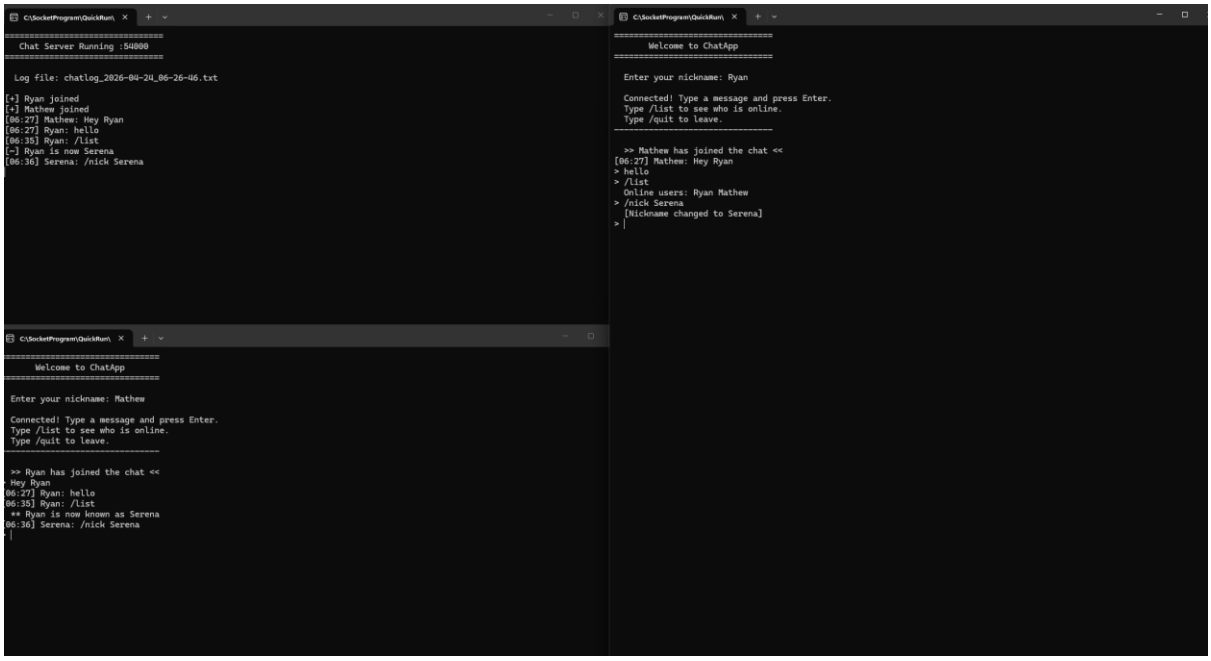

Figure 4: changing nickname left hand side is the other users and server, the right hand side changes their nickname

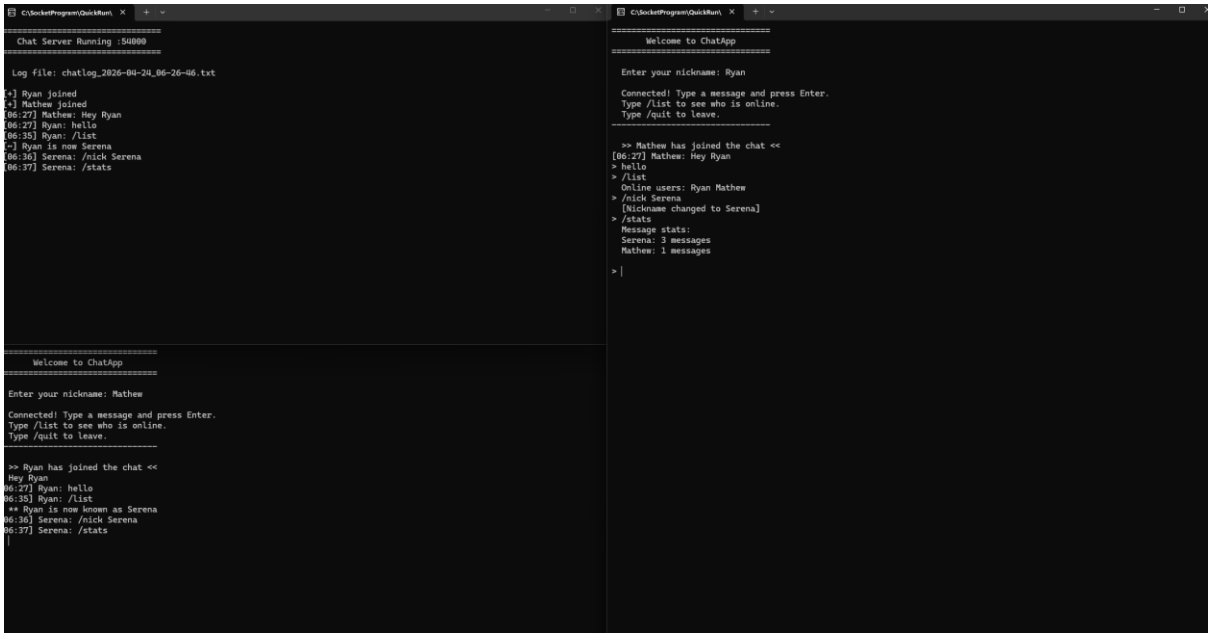

Figure 5: calling the stats function, displaying the session stats, user on the right calls it on the left is the other users and server view

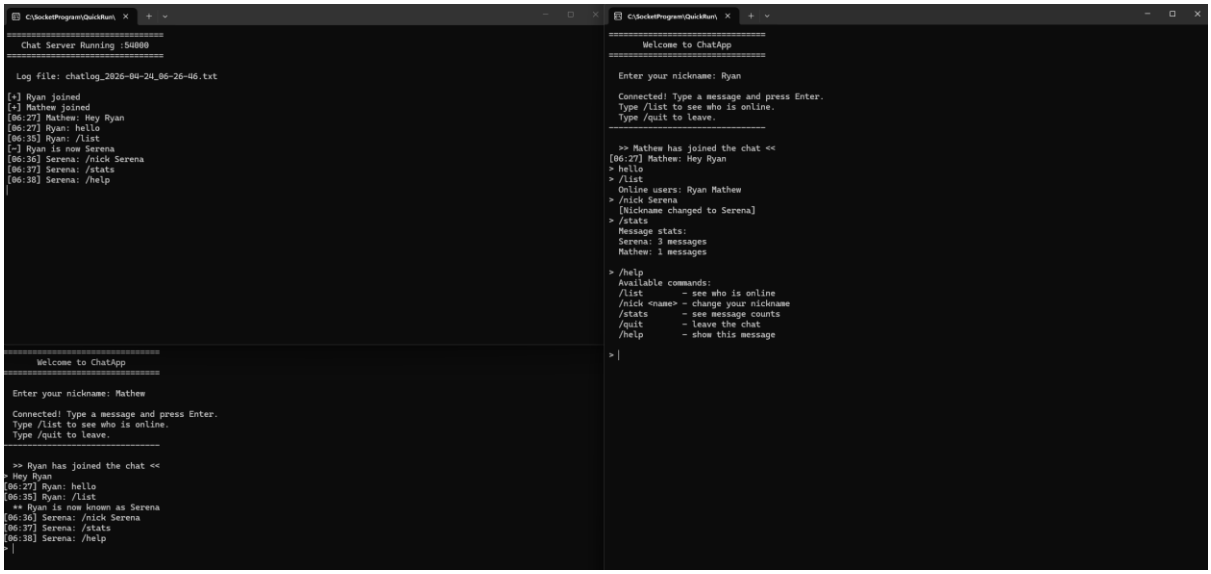

Figure 6: calling the help command. The right hand side calls the command, the left hand side is the server and other users

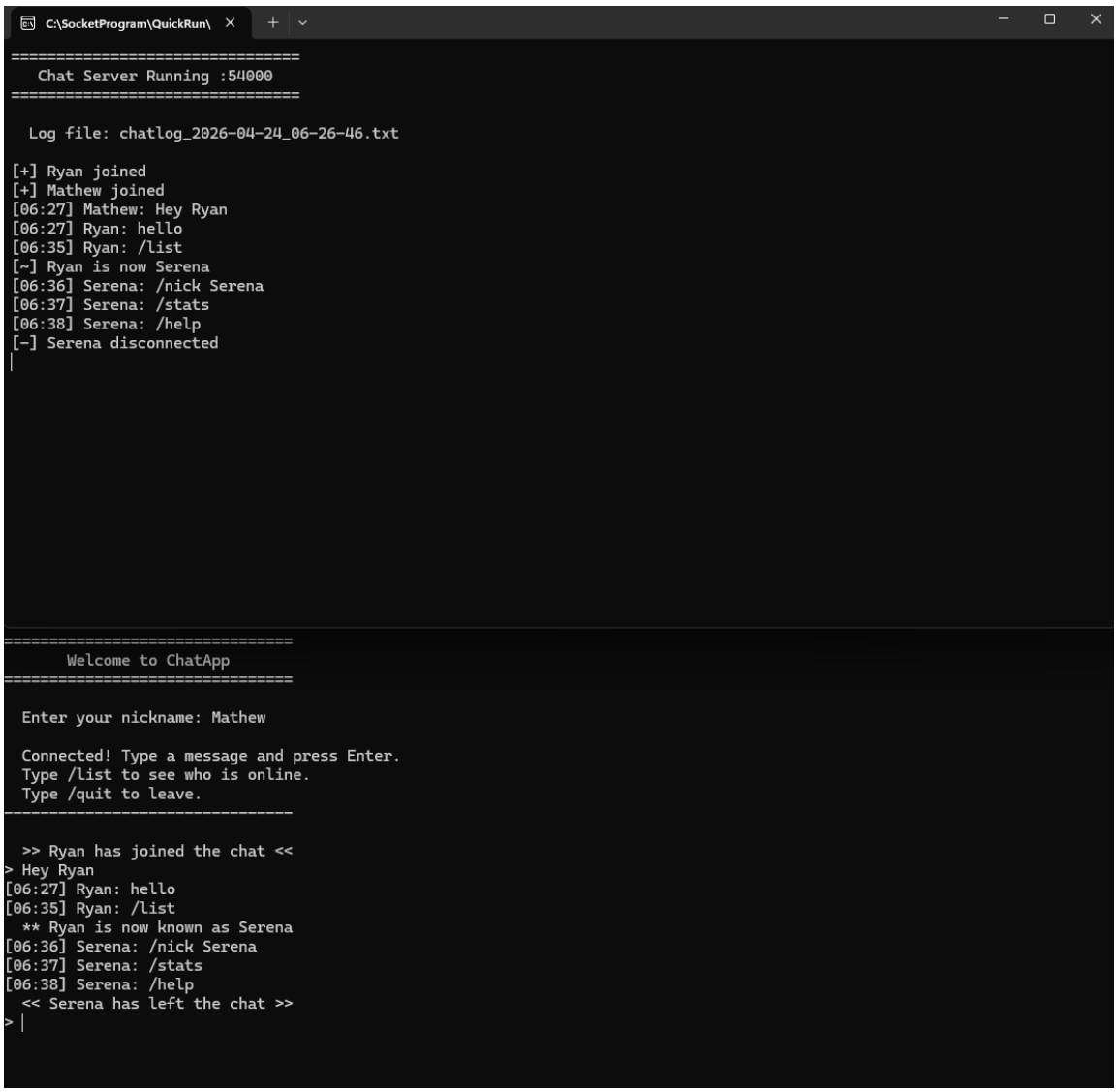

Figure 7: a user disconnecting from the session

File Edit View

```
[2026-04-24 07:21:01] === Server session started ===
[2026-04-24 07:21:22] === Server session ended ===
```

Figure 8: after 20 seconds the server shut down