CHAT SERVER CONSOLE APPLICATION

Version 2, December 2023

A. PREFACE

The Project is designed around using the TCP transport protocol. Injecting networking calls into a real time application can be detrimental to its performance in multiple ways. For one, networking API calls are usually blocking by default.

This is designed around the asynchronous behavior of your computer network devices running alongside most of your program on the CPU. In summary, if you request a networking operation to happen, the active thread that the call is being made on will block until the conditions are fulfilled or fail for whatever reason. If you already had a single threaded program and you were to add blocking related networking between some of the logic, the program would only be as responsive as the networking performance. If the program requests data, it cannot continue until the data it requested arrives. This is why a simple fix for this problem can be to make the network calls not to block. However, this introduces a problem from the opposite angle. If a networking call was blocking, it is sure to deliver the request as soon as a low-level interrupt resumes your application thread when delivering the results of the request to your application. When the network call blocking is turned off, if the network call request can't be completed, then the control is instead returned to the application making the call.

The application can then go about processing any other logic that needs to be done such as updates and real time rendering calls. The other angle issue here is that every time an incomplete net operation happens, it must wait for other processing before it can attempt to make a request again. This is problematic for Page 2 software that wants to show the results of networking as soon as it can. A real time networked program using non-blocking TCP calls would also have to split up partial messaging on a frame-to-frame basis, because doing the full partial message loop described to you so far would almost be as detrimental as blocking, causing unstable frames and an inconsistent flow of the other real time operations. The solution to these problems you might have guessed is to place the networking calls on separate threads and retain their ability to block.

In fact, blocking is good when dealing with other threads, because they are de-scheduled when the blocking conditions aren't met. This allows the CPU to focus performance to other threads and processes running. When any network condition is met, any operations that need to interface with the unhinged software running on the original thread can be immediately instigated. The TCP transport protocol provides guaranteed in order delivery of the data. This is often important for certain types of data in games such as communication related operations like chat functionality. The UDP transport protocol doesn't hold these features over the data transport which can cause performance gain provided

guaranteed in order delivery is not required. If UDP was being used and data needed to be guaranteed, your application layer would need to provide the understanding of this and potentially handle extra network operations to get the required data. Doing so can sometimes enact a performance degradation greater than the built in TCP protocol being performed at a lower level, especially during times of unstable network conditions, when networking that requires guarantees will need to play catch up.

This project is also designed to give an insight when attempting to choose the right transport related protocols depending on the networking problem. TCP is chosen because chat messages should not be dropped and should be delivered in order.

B. PROJECT OBJECTIVES

- Gain Profound Expertise in C/C++ through Socket Programming.
- Integrate Multithreading and In-thread Multiplexing for Service Handling
- Revive C/C++ Programming Knowledge.
- Apply Project and Time Management Skills.

C. PROJECT OVERVIEW

In this networking project, you will be developing a **server console application** in C/C++ to facilitate and mediate communication between clients.

You will be using SpaghettiRelay program[1] as the client for testing the requirements of this project. You will be incorporating the same logic used in Spaghetti Relay for both sending and receiving messages to guarantee a successful exchange of data between your server and Spaghetti Relay client instances. The prescribed message format involves a single-byte length indicator followed by the actual message content.

The project is divided into three phases, each with specific deliverables. The final product is a chat server with the following features:

- In-thread multiplexing
- Multithreading
- UDP broadcast

[1] The executable file of the client is provided in the zipped folder in section 1.4 on FSO.

- User registration
- User authentication
- Hashtable[2]
- Commands[3] execution
- Messaging
- Logging

D. PROJECT FEATURES SUMMARY

Here is a detailed summary of the features you need to implement for each phase of the project, along with guidelines on how to implement them using C++.

Phase 1: Server Setup and User Registration (26 Points)

Feature	Description and Guidance	Points
1.1 Server Setup	Setup the TCP server (listening socket, masterSet and readySet). Prompt the user for TCP Port number, chat capacity, and the command character (default is ~) Obtain server host IP (IPv4 and IPv6) and port using the **gethostname() and getaddrinfo()** functions. The information should be displayed on the server console upon startup. Implement **select()** for in-thread multiplexing. Implement TCP framing (1B length + message) for sending and receiving messages. This step is crucial for the successful exchange of data with SpaghettiRelay client programs.	5 0 2 8 2
1.2 Welcome Message	Upon successfully establishing a connection with the client, the server is required to transmit a welcome message, displaying the default character (~) designated for issuing commands. This welcome message is intended to appear on the	2

[2] Hashtables provide constant-time average-case complexity for search operations, Collision Handling and fast Insertion and Deletion. Retrieving a username or password based on a key (e.g., username) which is generally very fast, making authentication processes efficient.

[3] Commands are distinguished from regular messages using a special character. In this document it is assumed to be the character (~), you can select a character of your choice.

Feature	Description and Guidance	Points
	SpaghettiRelay GUI immediately after the connection is established.	
1.3 Help Command	**~help** command: it is intended to provide active clients with the list of available commands on the server. This feature serves to enhance user experience by providing clear and accessible information about the functionalities and operations supported by the server.	2
1.4 User Registration	**~register** command: it is designed to register a user on the server. The server's response may indicate success, failure, or decline if the server's capacity is reached. The required format for this command is **~register username password.** To maintain uniqueness, usernames are stored in a hashtable along with their corresponding credentials. Note: Registration does not automatically activate the user. To engage in sending and receiving messages, users must log in after registration.	5

Phase 2: Client Authentication and Basic Commands (20 Points)

Feature	Description and Guidance	Points
1.5 Client Authentication	**~login** command: it is intended to initiate the login process for registered users. It expects the following format: **~login username password.** Upon execution, the system will provide responses indicating success, user not found, or incorrect password. It is important to note that if a user is already logged in, the client program cannot register or log in with a different user until a logout command is executed. This ensures that only one user is active at a time, maintaining security and preventing unauthorized access.	5
1.6 logging and Basic Commands	Create two separate log files: one for user commands and another for public messages (excluding direct messages or DMs). **~getlist** command: it is intended to furnish the list of active clients, i.e., those currently logged in and to transmit it to the client who initiated the request. **~logout** command: it handles the user disconnection. This command is intended to gracefully log out the user and initiate the TCP termination handshake (FIN/ACK/FIN/ACK) to disconnect the client sockets. **~logout** command: it is intended to transmit the content of the log file that includes the public messages to the client who initiated the request.	2 2 2 4
1.7 Message Relay	The default behavior of the server is to relay messages received from a client to all active clients. However, this can be overridden by using the command **~send username.** For example, in a scenario where the server has five connected clients (c1, c2, c3, c4, and c5), if c1 sends a message (e.g., "Hello" followed by an enter key), the server will relay this message to c2, c3, and c4, ensuring that all connected clients receive the communication. If c1 specifically wants the message to be sent only to c3, then it should use the command **~send c3 message.**	5

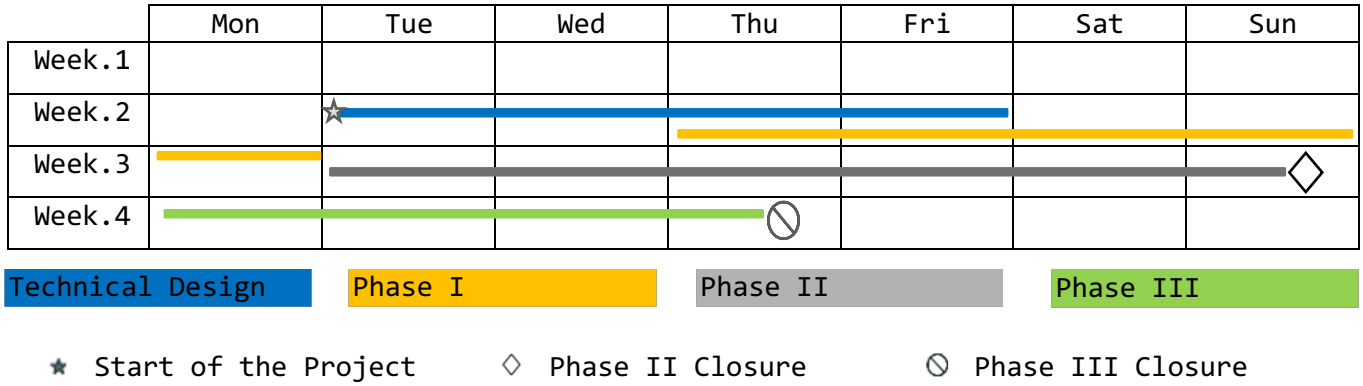
Phase 3: Broadcast Server Information (10 Points)

Feature	Description and Guidance	Points
1.8 Broadcast Server Information	Establish the UDP connection on a separate thread within the server. Construct the broadcast address structure. Compose the broadcast message. Regularly dispatch the broadcast message every x seconds using the sendto() function.	5 1 1 2

E.GENERAL GUIDELINES FOR IMPLEMENTATION

1. **Code Structure**: Follow modular and organized coding practices. Break down the functionality into functions or classes for better readability and maintainability.
2. **Error Handling**: Implement detailed error handling throughout the code. Use descriptive error messages or log entries for better debugging and troubleshooting.
3. **Documentation**: Provide inline comments for complex code sections. Document functions, variables, and any critical decisions made during the implementation.
4. **Testing**: Thoroughly test each feature and phase before moving to the next. Include edge cases and handle them gracefully.

F.PROJECT IMPLEMENTATION PLAN



G. REMARKS:

1. **SO_REUSEADDR:**

    - The **SO_REUSEADDR** option should be enabled to allow a socket to bind forcibly to a port that is already in use by another socket. The second socket should invoke **setsockopt** with the **optname** parameter set to **SO_REUSEADDR** and the **optval** parameter set to a boolean value of **TRUE** before calling **bind** on the same port as the original socket.

2. **SO_BROADCAST:**

    - The **SO_BROADCAST** option should be enabled to allow outgoing broadcasts from a socket.

3. **UDP Socket Binding:**

    - For UDP sockets that expect to receive messages, it is recommended to explicitly bind the socket to INADDR_ANY. This ensures that the application can receive UDP messages on any available network interface, and it avoids having the operating system automatically assign a port number.

H.WHAT AND WHEN TO SUBMIT?

At the conclusion of each implementation phase, you are required to submit specific project files, adhering to the outlined schedule.

Project Initiation (Week 2 Day 5 - Friday)

- **Submission:** **Technical design document** (.docx or .pdf)

Phase I (Week 3 Day 2 - Tuesday)

- **Submission:** Project **source and solution files** (.h, .cpp, .sln) for features 1.1, 1.2, and 1.3.

Phase II Closure (Week 3 Day 7 – Sunday)

- **Submission:**
    1. Project **source and solution files** including **(.h, .cpp, .sln)** incorporating Phase I and Phase II requirements (features 1.1 to 1.6).
    2. **Video recording** demonstrating your project work. For details on the specific demo scenario and content to cover, please refer to **Section J** of this documentation. Ensure clear narration and visibility of the screen throughout the demonstration, guiding viewers through each step.
    3. Wireshark **.pcap file** capturing the traffic generated by the chat project.
    4. A **screenshot** highlighting the packets exchanged during the TCP connection setup between the server and a client. The exchange should be clearly marked.
    5. A **screenshot** highlighting the packets illustrating the graceful connection termination between the client and server in response to the ~exit command. The exchange should be clearly marked.

Phase III Closure (Week 4 Day 4 – Thursday)

- **Submission:**
    1. Project **source and solution files** including (.h, .cpp, .sln) incorporating all project phases I, II and III.
    2. Wireshark **.pcap file** for the UDP traffic generated by the chat project.

3. A **screenshot** highlighting a packet containing the UDP broadcast, displaying its content IP: PORT (e.g., 127.0.0.1:31337). The message details should be clearly marked.

Note: Ensure that your submissions are organized, and file formats comply with the specified requirements.

I.RUBRIC

Project Initiation: Technical Document	10
Phase I Features	26
Phase II Features + User Experience	30
Phase III Features + Integration of Phases I, II and III	14
Phase II Closure	16
Demo recording	10
Wireshark .pcap + screenshots	6
Phase III Closure: Wireshark .pcap + screenshots	4
Total	**100**

Evaluation of User Experience

Intuitiveness: The interface is highly intuitive, allowing users to navigate effortlessly.	**2**
Responsiveness: The system responds promptly to user inputs, providing a seamless experience.	**2**
Visual Appeal: The design is visually pleasing, enhancing the overall user engagement.	**2**
Consistency: The user experience is consistent across different features and functionalities.	**2**
Error Handling: Clear and user-friendly error messages contribute to a positive experience.	**2**

J.INSTRUCTIONS FOR THE VIDEO RECORDING DEMO

1. **Run the Server (Set Chat Capacity to 3):**
    - Begin by launching the server application, setting the chat capacity to accommodate three users.
2. **Wireshark Broadcast Message:**
    - Demonstrate the Wireshark tool capturing and displaying the broadcast message sent by the server.
3. **Run 4 Spaghetti Relay Client Instances:**
    - Start four instances of the Spaghetti Relay client on the demonstration environment.
4. **Client Registration and GetList Commands:**
    - Register each client sequentially and execute the "getlist" command after every registration to showcase the dynamic client list update.
5. **Registration Decline for Client 4:**
    - Attempt to register the fourth client to illustrate the registration decline process.
6. **Client Exit and Login Test:**
    - Initiate an exit operation on one of the connected clients and then demonstrate the login command functionality.
7. **Message Relay Functionality Test:**
    - Showcase the message relay functionality by sending chat messages from one client to another.
8. **Send<Username> Command Test:**
    - Test the "send username" command from a selected client to validate direct message sending.
9. **Client Disconnection and GetList Update:**
    - Execute the "logout" command on all clients except one and demonstrate that the client list stays updated after disconnection.
10. **Server and Client Log Files:**
    - Display the server and client1 log files in the demonstration folder to provide insights into the system's operation.
11. **GetLog Command Test:**
    - Execute the "getlog" command on the remaining client and exhibit the content of the client log file using a folder view.