**Simple Chat Application**

The project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology

In

Computer Science and Engineering

School of Engineering and Sciences­­

A logo of a tree

Description automatically generated

**Under the Guidance of:**

**SUBHANKAR GHATAK**

**SRM University–AP**

**Neerukonda, Mangalagiri, Guntur**

**Andhra Pradesh – 522 240**

**November 2023**

**Submitted by:**

**(AP22110011623) Jithsungh sai**

**Certificate**

Date: 11/20/2023

This is to certify that the work present in this Project entitled “SIMPLE CHAT APPLICATION” has been carried out by jithsungh sai

The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology/Master of Technology in School of Engineering and Sciences.

Supervisor

(Signature)

Prof. SHUBHANKAR GHATAK

**Acknowledgements**

We would like to express my special thanks of gratitude to our professor Mr. Shubhankar Ghatak , who gave us the golden opportunity to do this wonderful project of ‘Simple Chat Application’.

We extend our heartfelt gratitude to all individuals who contributed to the completion of this project report on bidirectional TCP communication via Winsock in C++. Our sincere appreciation goes to for their invaluable guidance, support, and expertise throughout this endeavor.

We would like to express our gratitude to [Supervisor/Instructor's Name] for their continuous encouragement, insightful feedback, and unwavering support, which significantly enriched the development of this report.

Additionally, we acknowledge the contributions of our peers and colleagues who offered their insights and assistance, enhancing the depth and quality of this project report.

This project report stands as a collective effort, and we acknowledge and appreciate the support, guidance, and contributions of everyone involved in making this endeavor a success.

**Table Of Contents:**

Certificate 2

Acknowledgements 3

Table of Contents 4

Abstract 5

Statement of Contributions

Introduction 6

Methodology 8

Discussion 11

Conclusion 14

Future Work 15

**Abstract:**

This Simple Chat Application is a communication protocol that provides full-duplex communication channels over a single [TCP](https://www.pubnub.com/guides/tcp-ip/) connection. It enables real-time, event-driven communication between a client and a server.

Unlike traditional HTTP, which follows a request-response model, WebSockets allow bi-directional communication. This means that the client and the server can send data to each other anytime without continuous polling.

WebSockets are used for real-time, event-driven communication between clients and servers. They are particularly useful for building applications requiring instant updates, such as real-time chat, messaging, and multiplayer games.

In traditional HTTP, the client sends a request to the server, and the server responds with the requested data. This request-response model requires continuous polling from the client to the server, which can result in increased latency and decreased efficiency.

On the other hand, WebSockets establish a persistent connection between the client and the server. This means that once the connection is established, the client and the server can send data to each other at any time without continuous polling. This allows realtime communication, where updates can be sent and received instantly.

**Introduction:**

This project report encapsulates the essence of bidirectional TCP communication via Winsock in C++. Exploring server-side establishment and client-side interaction, the report delves into socket initialization, message exchange, and error handling.

It elucidates the intricacies of networking fundamentals, elucidating the process of connecting, sending, receiving, and terminating interactions. By dissecting these code segments, the report provides a comprehensive insight into basic networking paradigms, showcasing their practical application in C++ programming with Winsock for seamless communication between server and client entities.

This project delves into the implementation of TCP communication using Winsock in C++, showcasing bidirectional interaction between a server and client.

The report explores two fundamental code segments: the server-side code establishing connections, receiving and transmitting messages, and the client-side code initiating connections and engaging in message exchange.

Emphasizing network communication fundamentals, the report navigates through socket initialization, data transmission, error handling, and graceful termination procedures. By dissecting the intricacies of TCP-based communication, this project report offers a comprehensive understanding of basic networking principles applied in practical C++ programming with Winsock.

The main theme of the project is the web socketing in a very basic manner.

**How does this chat work :**

Before a client and server exchange data, they must use the TCP (Transport Control Protocol) layer to establish the connection. Using their WebSocket protocol, webSockets effectively run as a transport layer over the TCP connection.

Once connected through an HTTP request/response pair, the clients can use an HTTP/1.1 upgrade header to switch their connection from HTTP to WebSockets. WebSocket connections are fully asynchronous, unlike HTTP/1.1, however. WebSocket connection is established through a WebSocket handshake over the TCP. During a new WebSocket handshake, the client and server also communicate which subprotocol will be used for subsequent interactions. After this is established, the connection will run on the WebSocket protocol.

It is important to note that when running on the WebSocket protocol layer, WebSockets require a uniform resource identifier (URI) to use a “ws:” or “wss:” scheme, similar to how HTTP URLs will always use a “http:” or “https:” scheme.

**Methodology:**

**Technical Overview:**

1.1 **Technologies Used:** The project relies on the utilization of C++ as the primary programming language, known for its efficiency and versatility in handling system-level programming tasks. Additionally, it integrates the Winsock library, specifically designed for Windows systems, enabling robust and efficient networking functionalities. This combination forms a sturdy foundation for implementing socket programming, facilitating seamless communication between server and client entities.

1.2 **System Architecture:** The architecture is structured around a straightforward client-server model, embodying the core functionalities of network communication. The server component of this architecture takes the forefront, initiating the process by initializing the Winsock library, creating a socket, binding it to a predefined address and port, and actively listening for incoming connections. Upon establishing a connection with a client, it facilitates message exchange, emphasizing the server-side operations within this communication model.

1.3 **Key Features:** The implemented server boasts a series of essential features critical to its functionality within the client-server paradigm. These include:

• Winsock Initialization: The project handles the initialization of Winsock, a pivotal step to ensure the correct setup of the networking environment.

• Socket Creation and Configuration: It involves the creation and configuration of a socket, defining its characteristics and settings for communication purposes.

• Listening for Connections: The server actively listens for incoming connections, awaiting client requests to establish communication.

• Accepting Incoming Connections: Once a client seeks to connect, the server accepts the incoming connection, paving the way for bidirectional communication.

• Continuous Message Exchange: The implemented server incorporates a loop mechanism to facilitate continuous message exchange with connected clients, enabling seamless interaction and data transmission.

**2. Implementation Details:**

**2.1 Winsock Initialization:** The project begins with the initialization of the Winsock library using the **WSAStartup** function. This step is crucial for setting up the necessary environment for socket communication.

**2.2 Socket Creation and Configuration:** The server creates a socket using the **socket** function, specifying the address family (AF\_INET), socket type (SOCK\_STREAM for TCP), and protocol (0 for default protocol). The server's socket is then configured with a specific IP address and port using the **sockaddr\_in** structure.

**2.3 Listening for Connections:** Once the socket is configured, the server enters a listening state using the **listen** function. This allows the server to wait for incoming connections from clients.

**2.4 Accepting Connections:** The server accepts incoming connections using the **accept** function, which creates a new socket dedicated to communication with the connected client.

**2.5 Message Exchange Loop:** The core of the server lies in a loop where it continuously receives and sends messages with the connected client. The server prompts the user to type a message, sends it to the client, receives the client's response, and displays it. The loop continues until either party types "exit."

**2.6 Error Handling:** Throughout the implementation, robust error handling mechanisms are in place. The code checks for errors after each significant operation, such as socket creation, binding, listening, and accepting connections. Proper error messages are displayed, and the program terminates gracefully in case of errors.

**3. Testing and Validation**

**3.1 Test Scenarios:**

An extensive array of test scenarios was meticulously designed and executed to validate and ensure the robust functionality of the server. The test scenarios were strategically crafted to assess the server's capabilities under diverse conditions:

**Multiple Client Testing:** The server's ability to handle multiple simultaneous client connections was tested to gauge its scalability and concurrent communication capabilities.

**Message Variability:** Various messages of different lengths, formats, and content types were sent and received to ascertain the server's efficiency in processing diverse message inputs.

**Error Induction:** Deliberate introduction of errors, such as unexpected disconnections, invalid message formats, or abrupt terminations, was conducted to observe and analyze the server's response in handling and recovering from such error-prone scenarios.

Each test scenario aimed to validate specific functionalities while stressing the server's performance and reliability under various real-world conditions, ensuring its robustness in practical usage scenarios.

**3.2 Results**:

The comprehensive testing phase yielded conclusive results, affirming the server's competence and effectiveness in fulfilling its designated functionalities:

**Connection Establishment:** The server adeptly established connections with multiple clients, validating its ability to handle concurrent connections seamlessly.

**Message Exchange**: Diverse messages were successfully exchanged between the server and clients, demonstrating the server's capability to process varied message types efficiently.

**Error Handling:** Upon inducing intentional errors, such as unexpected disconnections or invalid message formats, the server demonstrated appropriate error handling mechanisms, recovering gracefully and maintaining operational stability.

**The results** corroborated the server's expected behavior, showcasing its reliability, efficiency, and adaptability in handling diverse scenarios. It validated the server's resilience and effectiveness in fulfilling its intended functionalities within the client-server communication model.

**Object-Oriented Programming (OOP) Concepts in the Context of Server-Client Communication Code:**

**1.Encapsulation:**

Explanation: Encapsulation involves bundling related data and functionalities into a single unit (a class), thereby restricting access to the inner workings of the class and allowing access only through well-defined interfaces (public methods or properties).

**Application to the Code:** In the absence of explicit class structures, encapsulation can be simulated by logically grouping functionalities. For example, creating classes or modules for managing socket connections, handling data transmission, or processing incoming messages can encapsulate related operations within these designated units. This helps in organizing and compartmentalizing the codebase, improving readability and maintainability.

**2. Abstraction:**

**Explanation:** Abstraction involves hiding the complex implementation details and exposing only the essential features or functionalities. It allows users to interact with a simplified interface, reducing complexity and enhancing usability.

**Application to the Code:** Abstracting socket operations by creating higher-level functions or classes that encapsulate Winsock functions can simplify the usage and comprehension of socket communication. By offering simplified interfaces to interact with sockets (such as connecting, sending, or receiving messages), abstraction shields the user from the intricacies of low-level socket handling.

**3. Modularity:**

**Explanation:** Modularity refers to breaking down a system into smaller, self-contained units (modules or functions) that perform specific tasks. Each module or function focuses on a single responsibility, enhancing maintainability, and facilitating reusability.

**Application to the Code:** Enhancing modularity involves structuring the codebase into smaller, distinct modules or functions, each responsible for a particular task. For instance, segregating functionalities such as socket initialization, connection handling, data transmission, and error handling into separate modules promotes cleaner code organization, making it easier to understand, modify, and debug.

**4. Polymorphism (Potential):**

**Explanation:** Polymorphism allows objects of different types to be treated as objects of a common base type. It enables flexibility by providing a way to perform a single action in various forms.

**Potential Application to the Code:** While not explicitly present in the provided snippet, future code enhancements could leverage polymorphism for handling different communication protocols or socket types (beyond TCP). By abstracting common operations into an interface or base class, diverse socket types can be managed uniformly while allowing each specific type to implement its specialized functionalities. For example, implementing TCP, UDP, or WebSocket protocols as subclasses of a base socket class could facilitate polymorphic behavior for handling various communication scenarios.

**5. Inheritance (Potential):**

**Explanation:** Inheritance involves creating new classes (derived or child classes) that inherit properties and behaviors from existing classes (base or parent classes). It promotes code reuse and allows for hierarchy-based structuring of classes.

**Potential Application to the Code:** Inheritance could be utilized if there were multiple socket types sharing common functionalities. A base socket class could define generic socket operations, while specific socket types (e.g., TCP, UDP) could inherit these common behaviors while implementing their specific functionalities. This hierarchical structure enables code reuse, maintains consistency, and facilitates easier extension of functionality for different socket types.

**Discussion:**

Algorithm ServerSideTCP():

Initialize Winsock

Create a socket for the server

Bind the socket to a specific IP address and port

Listen for incoming connections

Accept a client connection

while True do:

Receive message from the client

If received message is "exit":

Break the loop

Display received message from the client

Prompt server user for input

If server user input is "exit":

Break the loop

Send server user input to the client

Continue loop

Close the client and server sockets

Clean up Winsock resources

Exit the program

Algorithm ClientSideTCP():

Initialize Winsock

Create a socket for the client

Set up the server's address structure

Connect to the server

while True do:

Prompt client for input

If client input is "exit":

Break the loop

Send client input to the server

Receive message from the server

If received message is "exit":

Break the loop

Display received message from the server

Continue loop

Close the client socket

Clean up Winsock resources

Exit the program

**Conclusion:**

This project report serves as an insightful exploration into bidirectional TCP communication via Winsock in C++, providing a comprehensive understanding of server-side establishment and client-side interaction. It meticulously dissects socket initialization, message exchange, error handling, and termination procedures, elucidating the nuances of networking fundamentals in a practical context.

**Key Insights:**

Networking Fundamentals Unveiled:

The report delves into the intricate process of connecting, sending, receiving, and terminating interactions, offering a detailed exploration of basic networking paradigms using C++ programming with Winsock.

Comprehensive Code Segments:

Two fundamental code segments are showcased: the server-side code for establishing connections, receiving/transmitting messages, and the client-side code for initiating connections and engaging in message exchange.

Emphasis on Practical Implementation:

Emphasizing network communication fundamentals, the report navigates through the practical implementation of TCP-based communication, illustrating its application through Winsock in C++.

The Project's Focus:

The core theme of this project revolves around web socketing in a foundational manner. It highlights the practical application of bidirectional communication between server and client entities, laying the groundwork for understanding TCP communication using Winsock in C++.

In summary, this project report not only provides a comprehensive understanding of basic networking principles but also demonstrates their practical implementation in C++ programming with Winsock. It serves as an insightful exploration into the realm of web socketing, offering a foundational understanding of bidirectional communication protocols.

**Future Works**:

1. Concurrency and Scalability: Enhance the code to support multiple simultaneous client connections. Utilize threading, asynchronous I/O, or a non-blocking I/O model to allow the server to handle multiple clients efficiently.
2. Security Enhancements: Implement robust security measures. Incorporate encryption protocols (SSL/TLS) to secure data transmission and integrate authentication mechanisms to verify client-server interactions.
3. Protocol Diversity: Extend the code to support different protocols beyond TCP. Incorporating UDP or WebSocket protocols can enable applications with specific requirements such as real-time communication or broadcasting.
4. Error Handling and Recovery: Strengthen error handling mechanisms to manage unexpected scenarios effectively. Implement graceful error recovery strategies to maintain the stability of communication.
5. Cross-Platform Compatibility: Make the codebase platform-independent by abstracting platform-specific components or utilizing cross-platform libraries. This enables its use across different operating systems.
6. Performance Optimization: Optimize code efficiency to minimize latency and improve overall performance. Strategies like data compression, optimized data transmission, and resource utilization can enhance efficiency.
7. Modularity and Extensibility: Refactor the code into modular components to allow easier maintenance and future updates. Encapsulate functionalities into reusable modules for improved extensibility.
8. Integration with Higher-Level Frameworks: Integrate the code with higher-level networking libraries or frameworks to leverage additional functionalities without compromising performance.
9. Adaptation to Specific Use Cases: Tailor the code to cater to specific industry or application requirements, such as IoT devices, online gaming networks, or real-time collaborative tools.
10. AI and Machine Learning Integration: Explore integrating AI capabilities into the communication system for adaptive networking decisions, predictive analysis for optimizing data transmission, or anomaly detection.
11. Comprehensive Testing and Documentation: Conduct extensive testing to ensure the code's reliability and document it thoroughly to aid developers in understanding its functionalities and implementation details.

**Statement of contributions** :

Sahasra: assisited in writing the algorithm and report writing

Vivek: assisted in writing the code and debugging

Jithsungh: contributed significantly with the core concepts in writing the code

Bhavith Kalyan: helped with library functions and

Rachana acharya: developed the idea of web socketing concept.

Moussa demble: helped with report writing.