**Telnet Client Implementation Report**

**Introduction to Telnet**

Telnet is a network protocol used to provide a command-line interface for communication with a remote device or server. It operates over the TCP/IP protocol and allows users to log in to remote systems and execute commands as if they were local. Despite its historical importance, Telnet has been largely replaced by more secure protocols such as SSH (Secure Shell) due to its lack of encryption, which exposes sensitive information during transmission. However, Telnet is still useful for testing and debugging network applications.

**Implementation Approach**

The Telnet client is implemented in C using the Winsock library for socket programming. The implementation includes several key components:

1. **Socket Creation**: The program begins by creating a socket using the socket() function. A TCP socket is established with the AF\_INET address family, indicating the use of IPv4, and the SOCK\_STREAM socket type for TCP communication. Error handling is included to ensure that if the socket creation fails, an appropriate message is displayed, and the program exits.
2. **Setting Server Address**: After creating the socket, the server address is set using the struct sockaddr\_in structure. This structure is populated with the server's IP address (converted from a string using inet\_addr()), the address family (set to AF\_INET), and the port number (converted to network byte order using htons()). This step prepares the client for the connection to the specified server.
3. **Establishing Connection**: With the server address configured, the client attempts to connect to the server using the connect() function. This function takes the socket, a pointer to the server address structure, and the size of that structure as parameters. If the connection attempt fails, the program displays an error message indicating the failure and exits.
4. **Reading User Input**: The client enters a loop designed to continuously read user input from the console. A non-blocking method is employed to allow the program to monitor the socket for incoming messages while simultaneously accepting user commands. This ensures the client remains responsive to user actions.
5. **Sending User Input to Server**: Once user input is captured, the client sends this data to the server using the send() function. This function takes the socket, a pointer to the user input buffer, and the length of the input as parameters. Error handling is implemented to check for issues during data transmission.
6. **Receiving Output from the Server**: The client waits for a response from the server using the recv() function. This function reads the incoming data into a buffer, allowing the client to handle the server's responses effectively. The received data is then printed to the console for the user to see.
7. **Handling the "quit" Command**: Throughout the loop, the program checks for a specific user command, "quit." If the user enters this command, the client exits the loop and proceeds to clean up resources before closing the connection to the server. This provides a clean exit strategy for users.
8. **Cleanup**: After exiting the loop, the program ensures that the socket is closed and that Winsock is cleaned up using the closesocket() and WSACleanup() functions, respectively. This final step is crucial for freeing system resources and preventing memory leaks.

**Challenges Faced**

1. **Handling Telnet Protocol Specifications**: One of the main challenges was understanding and implementing the Telnet protocol, particularly regarding command negotiation. The Telnet protocol involves various commands and options that require careful handling to ensure proper communication.
2. **Socket Programming Errors**: Encountering various socket-related errors during development, such as connection failures and data transmission issues, was challenging. Each error necessitated thorough debugging and an understanding of the Winsock API.
3. **Non-blocking Input Handling**: Implementing non-blocking input handling to allow simultaneous data reception and user input posed some difficulties, particularly ensuring that user commands could be processed without interrupting the data flow from the server.