Client-Server Architecture

This project implements a client-server architecture using the socket programming paradigm. The communication follows a request-response model:

- **Client**: The client is responsible for sending commands to the server and processing the responses. It connects to the server using a TCP socket.
- **Server**: The server listens for client connections on a specified port and processes commands such as LOGIN, SEND, LIST, READ, DEL, and QUIT. It interacts with a mail spool directory for storing and retrieving message data.

Used Technologies

- Programming Language: C++
- Libraries: POSIX sockets, LDAP, C++17 Filesystem
- Synchronization: Mutex for thread-safe operations

Development Strategy

- **Design**
  - Defined a simple protocol with commands like LOGIN, SEND, LIST, READ, DEL, and QUIT.
  - Established validation rules for input fields (e.g., username: max 8 alphanumeric characters; subject: max 80 characters).
  - Decided to use a directory-based storage system for messages, with each message stored in a separate file.
- **Implementation**
  - **Client**
    - Implemented functions for validating user inputs and handling commands.
    - Commands are converted to structured requests and sent to the server.
  - **Server**
    - Implemented handlers for each command, including LDAP authentication.

        - Used file I/O to store and retrieve messages.
        - Managed user-specific directories within the mail spool.

- **Testing**
    - Verified end-to-end communication for all commands.
    - Tested edge cases for input validation and error handling (e.g., invalid usernames, non-existent message files).

Needed Protocol Adaptations

- During development, the protocol was extended to include the LOGIN command for user authentication.
- The current implementation also creates a new thread for every incoming connection to handle client requests and responses. This approach ensures that multiple clients can be served simultaneously.

Synchronization Methods

To ensure consistency in accessing the mail spool and handling login attempts, mutexes were used. Since each client connection runs in its own thread, mutexes prevent race conditions and ensure thread-safe operations.

Handling of Large Messages

In our code, large messages are handled by collecting the full message content on the client side until a . is entered to indicate the end. The complete message, along with the SEND command, is then sent to the server in one request. On the server side, the message is processed and stored in a file within the recipient's directory.