HW1 README:

1. Overview

This project implements a simple client–server protocol over TCP.
The server supports multiple concurrent clients using select() (no threads).
Communication is line-based (\n at the end of each message), and the client must wait for a response from the server before sending another request.

2. Usage

The server's script must run **before** running client's script.

Server -

./ex1_server.py users_file [port]

users_file – a text file containing username<TAB>password pairs

port – optional, defaults to 1337, must be an integer

Client -

./ex1_client.py [hostname [port]]

hostname - optional, default is localhost, can be either an IP address or a string host name

port – optional, defaults to 1337, must be an integer

Note: A port cannot be supplied without a hostname.

3. Protocol Description

3.1 Overview

Upon execution, the client initiates a TCP connection to the server. The interaction begins with the server sending a "Welcome" message. The protocol then proceeds to a login phase: the client locally prompts the user with "User: ", reads the input, and sends the full string (prefix included, e.g., User: <input>) to the server. Immediately after, it prompts "Password: " and sends the full string to the server as well.

If the credentials match an entry in the users_file, the server responds with a "Hi <username>..." message, and the session is established. Otherwise, the server reports a failure and allows the client to retry the login process immediately (with no limit on the number of attempts).

Once logged in, the communication is synchronous (stop-and-wait): the client sends one command at a time (triggered by the Enter key) and must wait for the server's response before allowing the user to send the next command. Pipelining requests (chaining multiple commands with \n) are not supported.

3.2 Connection details

When a client connects, the server sends:

Welcome! Please log in

3.3 Login details

After getting the "welcome" message, the client must send to the server exactly:

User: <username>
Password: <password>

The client must send these messages in that order, with no other input in between.

To ensure that, we print "User: ", waiting for the client's input and sending it to the server, then we print "Password: ", waiting for the client's input and sending it to the server as well.

If the credentials match the users file, the server sends to the client:

Hi <username>, good to see you

Otherwise, it sends:

Failed to log in.

If a message does not follow this sequence, or if the credentials don't match the users file, the server sends to the client "Failed to log in." and let it try again.

4. Supported Commands (after successful login)

a. parentheses: X

Checks whether X is a balanced sequence of (and).

Server responds:

the parentheses are balanced: yes

or

the parentheses are balanced: no

Note: The server ignores non-parentheses characters, so strings like "(X)" are acceptable.

b. lcm: X Y

Computes the least common multiple of integers X and Y (using abs(X*Y) // gcd(X,Y)).

Server responds:

the lcm is: R

Note: The LCM in case X = 0 or Y = 0 is 0.

c. caesar: plaintext X

Applies a Caesar cipher shift of X to plaintext (result is lowercase).

For a valid input X, the output will be: "the ciphertext is: Y" (Y is ciphered X).

If plaintext contains characters other than English letters or spaces: "error: invalid input".

Note: caesar cipher output is always lowercase.

d. quit

Client requests to disconnect.
The server closes the connection immediately (no response).

e. For any other input at this point, the server closes the connection.

5. Error Handling

Before login: any incorrect message → the server lets to try to log in again.

After login: any malformed command → server disconnects.

Client also validates commands and disconnects on invalid input.

6. Implementation Notes

Transport: TCP via socket(AF_INET, SOCK_STREAM).

We chose TCP because the application demands reliable and ordered data delivery. Critical operations like user login and command processing require every message to arrive correctly and in sequence to maintain the session state and logic.
TCP's built-in error checking and sequence numbering guarantee this essential reliability, making it the most robust and efficient choice over UDP for this specific protocol.

Server concurrency: handled using select() on the listening socket and all active client sockets.

Implementation Detail for Disconnection: when the server decides to disconnect a client (due to a 'quit' command or a malformed command), the server immediately closes the connection using the close() function on the client's socket. Then, the server removes this closed socket from the set of active sockets monitored by the select() system call to ensure it is no longer waiting for events from that client.

Protocol is line-based (each message from both sides ends with \n) and each client maintains a small login state machine. In addition, the client must wait for a response from the server before sending another request.