# Engineering Notebook - Chat Application

CS 262 HW1

Hanying Feng and Vera Yang

## Functionalities and Front End User Interaction Logic

We decided to design the front-end user interface for the Simple Chat Application first before implementing the back-end functionality. Our approach began with writing a design document detailing the interaction logic of the interface and ensuring that all required functionalities were incorporated into the logic chain.

For user data storage, we chose to use JSON format on the server side. Each user is identified by a unique username, which serves as the key, while the corresponding value is a dictionary containing:

* “password”: The user’s hashed password for authentication.
* “read\_messages”: A list of messages that the user has already read.
* “unread\_messages”: A list of messages the user has not read.

e.g.

*accounts* = {

“user”: {“password”: hashed\_password,

“read\_messages”: [{“from”: sender, “message”: message}],

“unread\_messages”: [{“from”: sender, “message”: message}]}

}

The following design document outlines the interaction logic:

Page 1 (***requirement 1: Creating an account***):

Title: Log in/Sign up

Display a note “(create a new account if the username has not been signed up)”

* Username: The user should enter a valid username (should not be empty or include space character).
* “Next” button: Goes to Page 2.

Page 2:

Title: Log in/Sign up

Display a note “(If the username does not exist, enter the password to sign up.)”

* Password: User should enter password. The password will be hashed before sending to the server.
* “Log in/Sign up” button:
  + If the username is in the account list, check if the password is correct. If it is correct, go to Page 3. If not, display the error message “Incorrect password.” Stay on Page 2. (***requirement 2: Log in to an account***)
  + Otherwise, create the account and add the information to the account list. Goes to Page 3. (***requirement 1: Creating an account***)
* “Back” button: Goes to Page 1.

Page 3:

Title: Account - username

* Display a note “(# unread messages)” where # represents the number of unread messages (# is the length of the list that is saved in “unread\_messages” key). This page will be refreshed every second, so the user can receive new messages that are sent to them immediately when they are logging in. (***requirement 2: Log in to an account & 4: Send a message to a recipient. If the recipient is logged in, deliver immediately.***)
* “List accounts” button (***requirement 3: List accounts***):
  + Goes to a new page and displays a note “Search for existing accounts”
    - The user should enter the *query* to search
      * If the *query* is empty, display all the usernames in the account list.
      * Otherwise, display all the usernames that contain the *query*.

(We created a scroll bar on this page, so all the usernames can be displayed comfortably.)

* + - “Back” button: Goes back to Page 3.
  + “Back” button: Goes back to Page 3.
* “Send messages” button: Goes to Page 4.
* “Read messages” button: Goes to Page 5.
* “Delete account” button (***requirement 7: Delete an account***):
  + Pop out a window and display “Do you want to delete this account? This will delete all your messages.” with “Yes” and “No” buttons.
    - “Yes” button: Removes the account from the account list and removes all the messages sent from this deleted username. This is to avoid confusion if the same username is signed up by another person in the future. Goes to Page 1.
    - “No” button: Goes back to Page 3.
* “Log out” button: Goes to Page 1.

Page 4 (***requirement 4: Send a message to a recipient***):

Title: Send messages

* Recipient: The user should enter a valid username.
* Message: The user should enter a message.
* “Send” button:
  + If the “Recipient” is not a valid username in the account list, display the error message “Invalid recipient. Please enter a valid username.” Stay on Page 4.
  + Otherwise, append a dictionary that contains the sender username and message ({“from”: sender, “message”: message}) to “unread\_messages” in the recipient’s account. Display “Send message successfully.” Stay on Page 4.
* “Back” button: Goes back to Page 3.

Page 5 (***requirement 5: Read messages***):

Title: Read messages

* Pop out a window and display “How many messages would you like to read? (Please enter an integer)”. The user should enter a number and save it as “per\_page”.
  + If the number is not an integer, display “Invalid number. Please enter again.” and stay on the same page.
  + Otherwise, save it as “per\_page”.
* If “unread\_messages” is not an empty list, go to page 5.1.
* If “unread\_messages” is an empty list, go to page 5.2.

Page 5.1 (***requirement 5: Read messages***):

Title: Read messages

* List the first “per\_page” number of messages from “unread\_messages” of the current user. Remove the displayed messages from “unread\_messages” and append them to “read\_messages”.
* “Back” button: Goes back to Page 3.

Page 5.2 (***requirement 5: Read messages & 6: Delete a message or set of messages***):

Title: Read messages

Display a note “(Click on the message to delete.)”

* Display all the messages from “read\_messages” of the current user. There is a scroll bar on this page, so all the messages can be displayed comfortably.
* If the user clicks on a message, pop out a window that displays “Do you want to delete this message?” with “Yes” and “No” buttons.
  + “Yes” button: remove this message from “read\_messages” in the account. Goes back to Page 5.2. The page should no longer display the message that has been deleted.
  + “No” button: Goes back to Page 5.2.
* “Back” button: Goes back to Page 3.

After finalizing the interaction logic, we decided to first implement the **JSON wire protocol** to establish communication between the client and server. This was a crucial step to ensure the application could function properly and support all required features.

## Implementation of JSON Wire Protocol

We implemented the core functionality by creating two primary files:

* *server.py*: Defines the Server class, which manages user information and implements all functionalities.
* *client.py*: Contains two classes:
  + ChatApp: Implements the Tkinter-based GUI that follows the previously defined interaction logic.
  + Client: Manages communication with the server.

We followed a structured approach to implement each functionality:

1. Front-End Interface (ChatApp Class): Designed and implemented the user interface for the functionality in the ChatApp class. We used ChatGPT and DeepSeek to learn and write code with Tkinter for creating the interface. For example, we implemented the scrollbar with the help of AI tools.
2. Client-Side Communication (Client Class): Developed functions to send requests and data to the server and process responses.
3. Server-Side Implementation (Server Class): Implemented server-side logic for handling requests and updating user data.

We implemented all the functionalities one by one in the order of the assignment requirements order.

The communication between the client and server follows a JSON wire protocol.

1. Each function in the Client class creates a dictionary containing the action type, username, and other required data for the specific function.

* Code: data = {“action”: “login”, “username”: user, “password”: hashed\_password}

1. The client encodes the data as a JSON string and sends it to the server.

* Code: json.dumps(data).encode('utf-8')

1. The server receives the message, decodes it, and converts it into a dictionary.

* Code: message = client.recv(1024).decode('utf-8')

data = json.loads(message)

1. After processing the request, the server encodes the response as a JSON string, sends it to the client, and the client decodes it using the same process.

After completing the implementation, we verified that all functionalities worked correctly using the JSON wire protocol. Then we can design our customized wire protocol and test it, which ensures that any bugs in the chat application must originate from the newly written code in the wire protocol section.

## Implementation of Our Customized Wire Protocol

After we have a working application using the JSON protocol, we now want to build a second implementation with our customized wire protocol only using sockets. To design this protocol, there are several questions we want to consider:

1. We are storing data in JSON format, but we might need to send data in utf-8 format through the network, so how should we encode and decode data between these two formats?
2. When the server tries to receive data from the client, how does it know how long the message will be? How can we inform the receiver how much data to receive from the sender?
3. The actual information contained in the message may have different formats as well. For instance, we might only send a success message, which is a continuous string itself, but what about when we want to send over lists of accounts or lists of dictionaries of messages?

With these considerations in mind, we came up we the following design for our wire protocol:

* The ‘data’ we want to send over between the server and the client will be a dictionary in the format {‘version’: VERSION, ‘operation’: OPERATION, ‘info’: “”}.
  + ‘version’: 1 byte, the version number that should be consistent between client and server
  + ‘operation’: 2 bytes, the 2-digit code that we define for each type of operation. For instance, server-side operations may include “SUCCESS” and “FAILURE”, while client-side operations may include “LOGIN” and “SEND\_MESSAGE”.
  + ‘info’: variable length, the actual information to be communicated, the format of which will depend on the type of operation.
* To enable sending data over in utf-8 format, we define serialization and deserialization methods to encode and decode the data.
  + Serialization: takes in a data dictionary, converts each of the three entries in the dictionary into utf-8, and combines them into a single byte string.
  + Deserialization: takes in a byte string in utf-8 format, converts the first byte into version number, the second byte into operation, and the rest of the byte string into a python string.
* We need to inform the receiver of the length of the message to be sent over. This requires us to send over a message length before sending the actual message. This is how it should work:

1. Sender prepares the data dictionary (version, operation, info) and serializes it into a byte string.
2. Sender calculates the length of the byte string and converts this length into utf-8. Use padding to ensure this piece of information has length 64 bytes.
3. Receiver receives 64 bytes and converts it to a length that represents how long the actual message will be.
4. Receiver receives data in length of the actual message.
5. Receiver deserializes the received data into a data dictionary and responds accordingly.

* How to handle different types of operations? We may be sending ‘info’ in different formats depending on the operation. Here are some cases to consider
  + Case 1: When the ‘info’ is just a string, e.g. “Send message successfully”, it can be easily encoded and decoded to and from utf-8.
  + Case 2: When we have more than one piece of information, e.g. when we send both username and password from the client to the server, we can join these strings by ‘\n’. When decoding, just split by ‘n’.
  + Case 3: If ‘info’ is a list of strings, e.g. list of usernames, we can join each username by ‘\n’ to turn the list into a continuous string. When decoding, just split by ‘\n’.
  + Case 4: ‘info’ could be a list of dictionaries. For instance, we store the unread messages for each user in a list of dictionaries named ‘unread\_messages’, where each dictionary looks like {‘from’: username, ‘message’: message}. In this case, when we send the unread messages from the server to the client, we first turn each dictionary into the format (username+’ ‘+message) and then join all these messages by ‘\n\t\n’. When decoding on the client side, first split by ‘\n\t\n’ and then split on the first ‘ ‘.

This protocol design achieved efficient communication between the client and the server. The customized wire protocol application has the same user interface and functionalities as the JSON protocol version.

## Comparison between JSON Protocol and Our Customized Wire Protocol

Our customized wire protocol efficiently serializes and deserializes messages into a compact byte format that includes the version number, the type of operation, and the actual information. This compact format can reduce the overhead associated with transmitting data over the network, and thus can result in lower latency and bandwidth usage than the JSON protocol. In addition, since we use a fixed-length header to indicate message sizes, the protocol reduces the complexity of delimiting or parsing messages.

The JSON wire protocol offers greater flexibility and integrability and is easier to understand than our customized protocol, as JSON is a widely adopted format for structured message transmission. JSON messages have key-value structures that are very self-explanatory, which increases readability and makes it easier for debugging. However, this structure also increases the message size due to added characters like brackets and whitespaces, which increases the cost of data transmission and makes it less efficient than our customized wire protocol.

1. Tests, Comments, and README

After completing the code for both the JSON wire protocol and the customized wire protocol, we began writing unit tests for both. We implemented comprehensive tests for all features, including setting up the server and client, logging in (new user, existing user, wrong password), listing accounts (retrieving all usernames and searching with a query), sending messages (valid and invalid recipients), reading messages (all messages and unread messages), deleting messages, and deleting an account. Both applications successfully passed all the tests. Finally, we added comments and docstrings to all the code. We also wrote a README.md file to document and explain all the functions.