Pedro's Engineering Notebook

12/7:

Today, I began developing a chat application that uses Tkinter for the graphical user interface (GUI) and socket communication for sending and receiving messages between users. One of the first challenges I encountered was understanding how to properly manage frames in Tkinter. Initially, I did not realize that when switching between frames, the old frames were not being properly cleaned up, which led to unexpected behavior and unnecessary memory usage.

To address this, I started implementing a CentralState class to manage shared state across different parts of the application. This class was meant to store information about the logged-in user and active connections. However, I quickly learned that without proper cleanup and refresh mechanisms, the changes in state were not being reflected properly when transitioning between frames.

As part of the solution, I structured the application with three primary frames:

- Onboarding: This frame is responsible for user login and setup.
- Navigation: This serves as a main menu that allows users to switch between different functionalities.
- MessageDisplay: This frame is used to display messages and interact with chat conversations.

12/8:

Today, I worked on handling message display updates and ensuring that new messages were correctly reflected in the UI. Initially, I encountered an issue where messages were not updating in real-time when displayed in the MessageDisplay frame. The problem arose because the frame was not being explicitly refreshed when a new message was received.

At first, I attempted to manually update individual widgets, but this approach proved to be inconsistent. The breakthrough came when I realized that instead of trying to modify the existing frame dynamically, it was more effective to delete the frame and recreate it from scratch

whenever a user navigated to it. This approach ensured that the `__init__` function of the frame was called again, which properly reloaded the messages without requiring additional update logic.

Additionally, I learned more about parent-child relationships in Tkinter, where child widgets (such as labels, buttons, and text boxes) are bound to their parent frames. Understanding this helped me debug issues related to message display, as some elements were not updating because their parent frames were not being reconstructed.

Ultimately, the key lesson from today was that in Tkinter, dynamically modifying an existing frame is not always the best approach. Instead, completely reinitializing the frame upon navigation can simplify state management and ensure that all UI components are properly updated.

12/9:

I shifted focus to server communication and connection handling, addressing several issues related to how the client interacts with the server. One major problem was ensuring that the client correctly handled cases where the server was unavailable. Initially, the client would attempt to connect once at startup, but if the connection failed, it would never attempt to reconnect, leaving the user stuck without a functioning chat.

To fix this, I implemented a retry mechanism that continuously attempts to reconnect to the server at regular intervals if the initial connection fails. This was particularly important for handling cases where the server was temporarily down or unreachable due to network issues.

Another challenge was deciding between JSON vs. Wire Protocol for communication between the client and server. JSON messages were more readable and structured, making debugging easier, but I encountered an issue where certain return statuses (such as errors or confirmations) were missing, causing inconsistencies in message handling. To resolve this, I ensured that every server response included a clear success or error flag, making it easier for the client to determine the correct behavior.

I also improved error handling for disconnections. The client now actively checks whether the connection to the server is still active and, if not, attempts to reconnect before sending new

messages. This prevents the issue where the client would assume it was still connected and fail silently when trying to send a message.

One key realization from today was that socket connections are not persistent by default, meaning that a client must actively manage the connection state and handle reconnections when necessary.

12/10:

Today's focus was on enabling cross-device functionality so that users could communicate across different machines on the MIT network. One major issue I encountered was that my server was initially set to listen only on localhost, meaning it could only accept connections from the same machine it was running on.

I discovered that to allow external devices to connect, the server needed to bind to `0.0.0.0` instead of localhost. The difference is that:

- localhost (`127.0.0.1`) restricts connections to only the local machine.
- `0.0.0.0` acts as a wildcard address, allowing the server to listen on all available network interfaces, including external connections.

Once the server was correctly set up, the client also needed to be updated to connect to the server's actual IP address rather than `localhost`. Since my machine's IP address on the MIT network was 10.250.88.150, I had to modify the client so that it explicitly connected to this address.

Additionally, I updated the environment configuration files to store the server's IP address dynamically, ensuring that it could be modified without changing the core code.

The key takeaway from today was understanding how network binding works in socket programming. A server can be made accessible to external devices by listening on `0.0.0.0`, but clients must connect using the server's real IP address, not `localhost`.

12/11:

Today, I focused on fixing critical JSON message handling bugs that were causing messages to be lost or incorrectly displayed.

One of the biggest issues was that the server's JSON responses for RET (retrieve messages) did not include a clear success flag. This caused the client to incorrectly assume that messages were missing when, in reality, they had been received but were not being displayed properly. To fix this, I added a 'success' key in the server responses, ensuring that the client could verify whether the operation completed successfully.

I also discovered that SEL (message received) responses were sometimes failing to trigger message updates. This was due to a race condition between the receipt of messages and the display refresh. If a message arrived just before a UI refresh, it would sometimes be skipped. To resolve this, I ensured that every new message automatically triggered a UI update, preventing messages from being lost due to timing issues.

12/13:

Today was focused on testing, documentation, and final refinements to improve client-server communication and overall system efficiency. At this stage, the core functionality was working correctly—messages were being sent and received as expected, the UI was updating properly, and the server was handling client connections. However, while the system was functional, there were areas that could be optimized to reduce unnecessary resource usage and improve responsiveness.

One of the key improvements involved refining the server reconnection logic. Initially, the client was successfully reconnecting when the server connection was lost, but this process was handled in the main execution flow, which could sometimes cause the user interface to lag or freeze. To address this, I created a dedicated background thread responsible for managing server connection retries. This ensured that reconnection attempts happened asynchronously, keeping the UI smooth and responsive.

Additionally, I optimized how messages were fetched from the server by ensuring that:

- MessageDisplay only fetches messages once upon initialization, instead of continuously polling the server.

- Real-time updates now rely on SEL (select) messages, allowing the server to push new messages instead of the client constantly checking for them.
- Periodic LA (list accounts) updates were retained for the Chat and Login pages to keep user lists up to date.

These optimizations made the system more efficient while keeping all necessary functionality intact. After a final round of testing, I confirmed that all improvements worked as expected, completing the project with optimized performance, clear documentation, and a clean code structure.