Matt Haffner

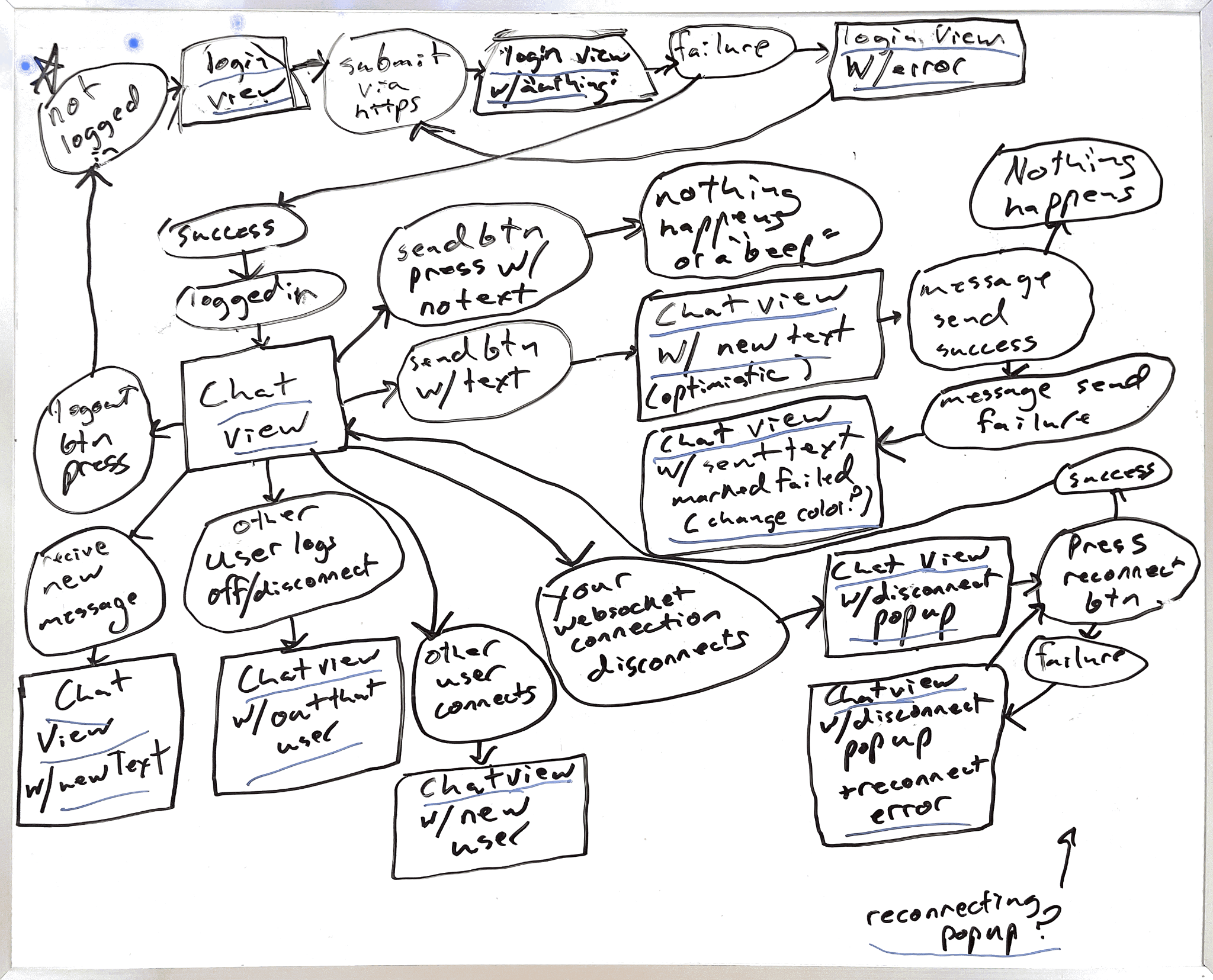
Daniel Pellowski

Ryan Gallagher

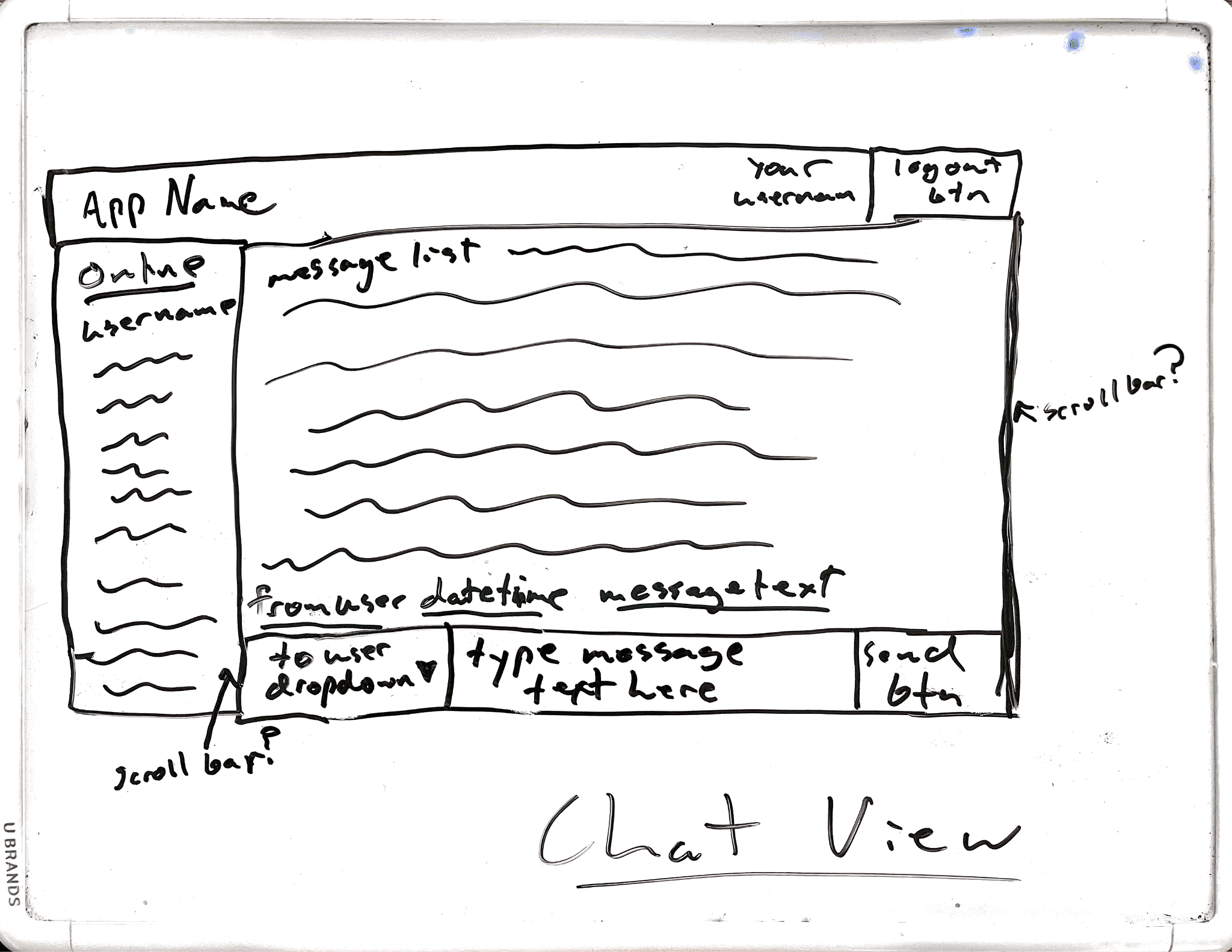
Dylan Skokan

Final Write Up

Our goal was to create a messaging application that can have clients connect to a server and can chat through the server. We wanted to require clients to have credentials for logging in, and clients can create an account for themselves if needed. We wanted clients to be able to send each other direct messages along with group messages to everyone. Below is our prototype state transition diagram for the logical flow of our overall program:

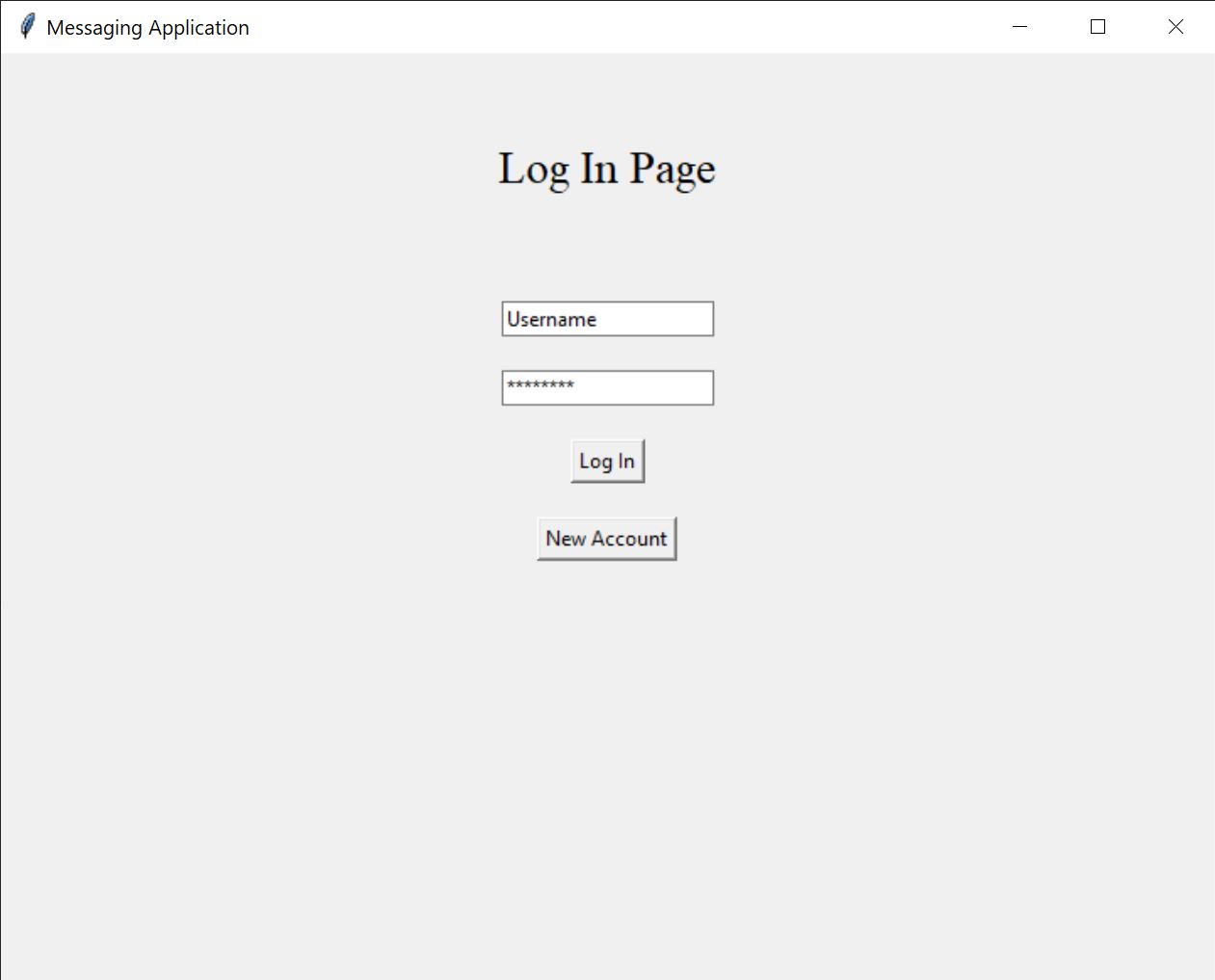


Initially, upon starting the client it prompts the user to enter in some login credentials. If the login credentials are valid, it takes the client to the main chat view. Here, the user can send messages, see everyone else who is online, and if they are an admin they can kick other users. We also want clients to have a GUI to be able to interact with the program easily. Here is an initial design concept for the client side GUI main chat page:



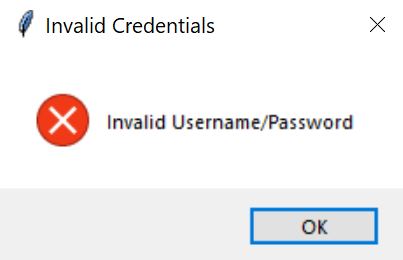
We decided to use the python library tkinter to build the GUI. Here are the GUI pages:

Login page:

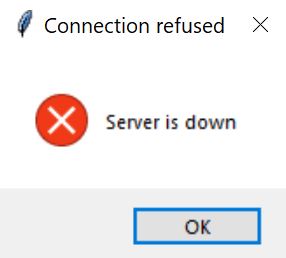


The login page allows for a user to enter in their username and password. The initial text “Username” and stars in the password field disappear when a user clicks on them. When the user pushes the login button it sends the information in the text fields to the server. If a user tries to login and it fails, we will get an error popup.

Error popups:

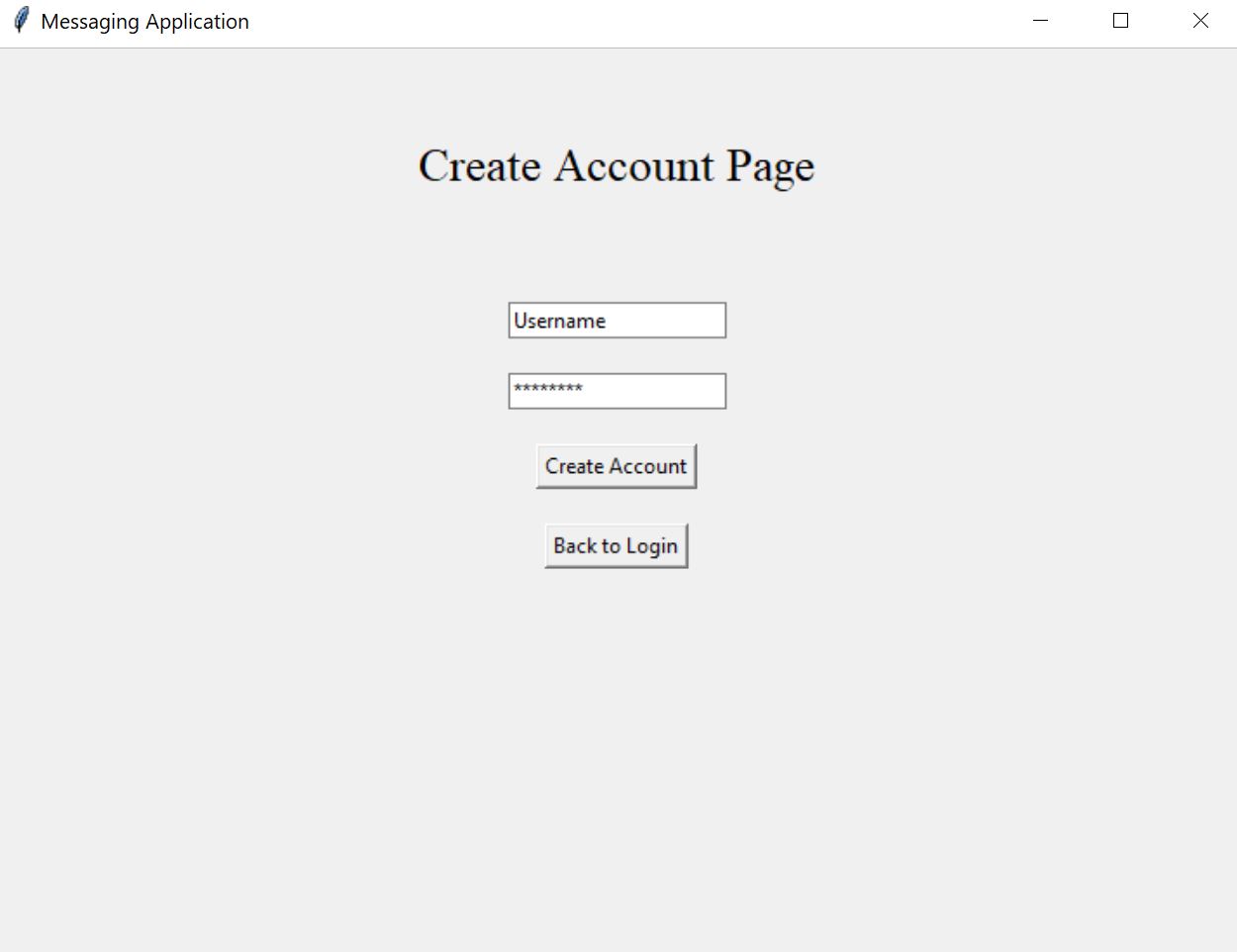


This popup will happen if the user either enters login information for an account that does not exist yet. This will also popup from the account creation page if the user enters an invalid name such as “group” or “admin”.

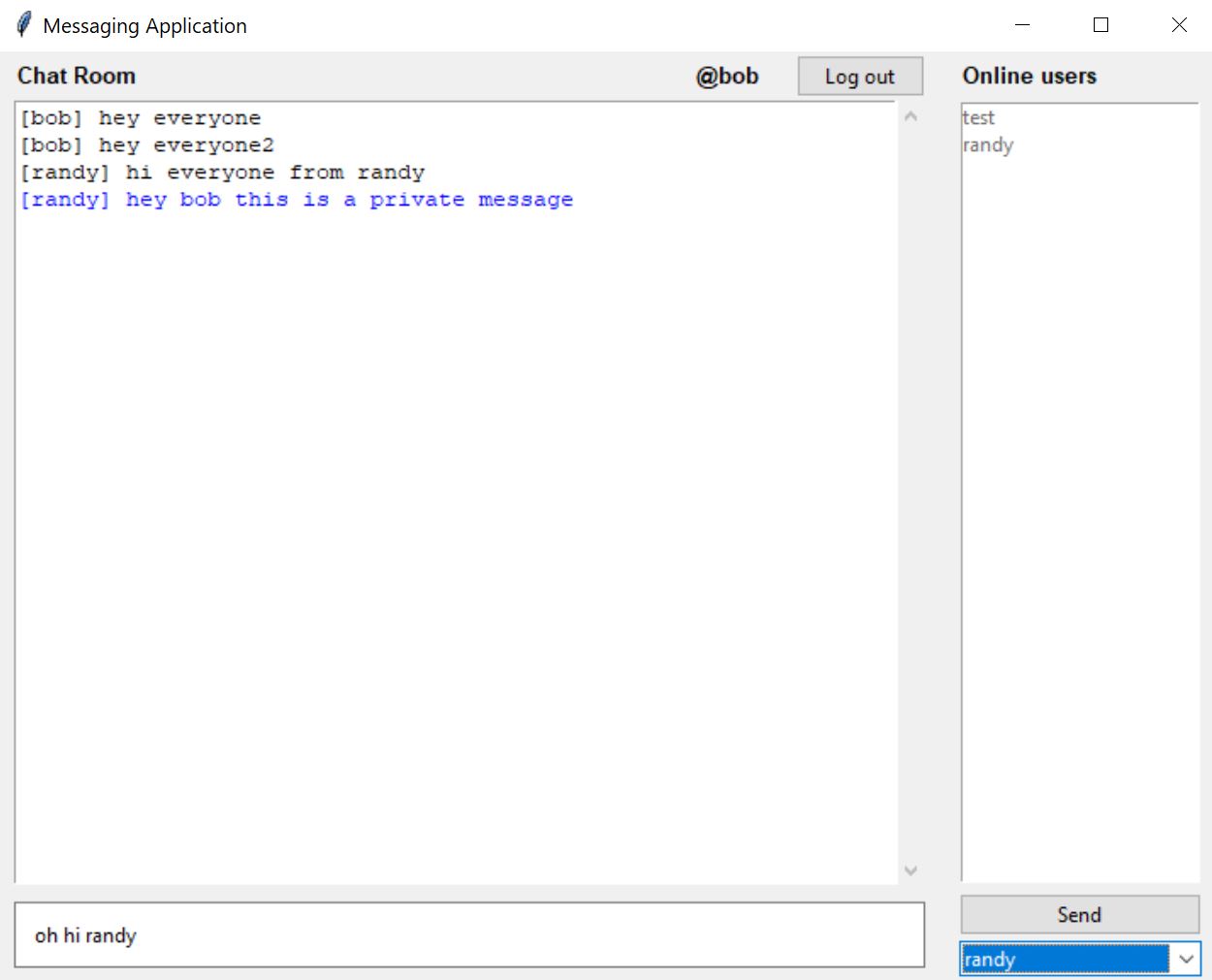


This popup will happen if the user tries to login and the server is down, regardless of credentials.

Create account page:



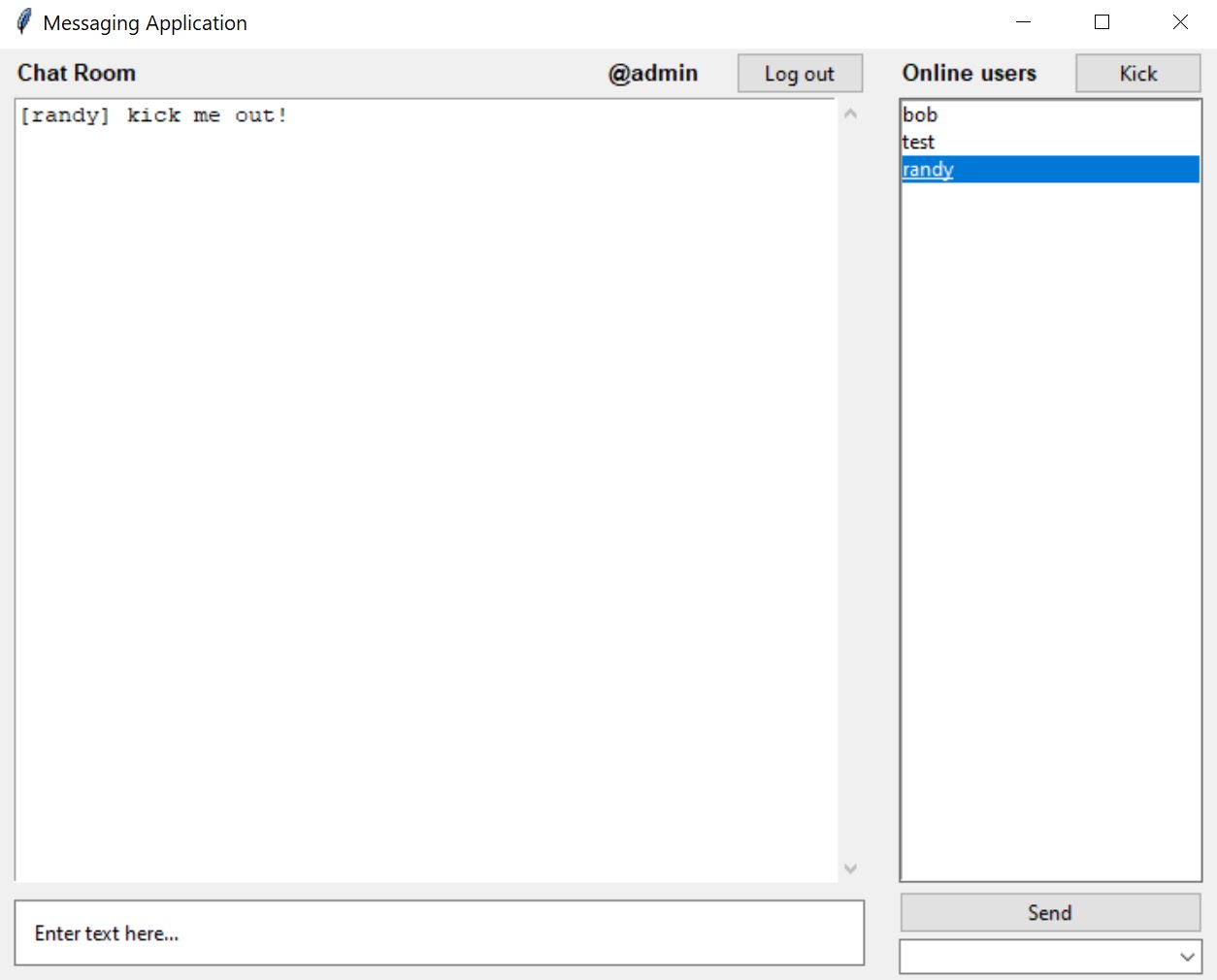
The create account page will allow users to create new accounts. When a user enters information to the text fields, the create account button will take the text from those fields and send it to the server. The user can also go back to the login screen if they want to go back.

Main chat page:

The user's name is shown at the top next to the logout button, which will log the user out and return to the login page. The “Chat Room” pane on the left shows messages to and from this user. We can see that along with group messages (messages to everyone) which are colored in black, there are also private messages which are colored in blue. The “Online Users” pane on the right shows all users who are currently online. When someone logs out or gets kicked, they are removed from this list. When someone logs on they are added.

At the bottom of the page we can see where the user can enter in their message and send it to other users. The dropdown box will show all users who are currently online as well as a “group” user. The “group” option is not actually a user, but it is the way to send the message to everyone.

When a user hits the “send” button and the text in the message field is not empty the client sends the message, the sender of the message, and who the intended receiver of the message to the server.

The admin version of the main chat page:

The admin account is created the first time the server is run. If someone logs in with admin credentials, they have a slightly different UI. Here we see that the admin has an option to kick a user from the online users. If the admin wanted to kick “randy”, they would have to click on his name on the right, highlighting it, then click the kick button. The kicked user gets disconnected and sent back to the login page.

Program Architecture and Design

All of the updates through the GUI are handled with the subscriber/publisher pattern. For example, the main chat page is subscribed to a user list object. This way, whenever there is a change in the user list, the “Online users” pane will be notified of the change and get updated. Similarly, the chat page is also subscribed to a message list object which listens for changes in the message list. When the message list changes, it notifies the chat room pane and updates it.

We chose the client-server architecture for our application because it fits the use case of our application the most effectively. A client-server architecture is employed to facilitate efficient and scalable communication between multiple clients. The server of our application is responsible for managing the state of the application, such as user accounts, messages, and user sessions. The clients are responsible for user interactions, rendering the user interface, and communicating with the server.

Our application also follows the Model-View-Controller (MVC) pattern to separate concerns and promote modularity. The MVC pattern consists of three main components:

* Model: Represents the data and the business logic of the application. In our application, the model classes include ‘User’, ‘UserList’, ‘Message’, ‘MessageList’, and ‘CurrentUser’.
* View: Manages the user interface and presentation of the data. The ‘ChatView’ class handles the rendering and updating of the user interface, including the login, account creation, and chat components.
* Controller: Acts as an intermediary between the model and the view, handling user input and updating the model and view accordingly. The controller in our application is responsible for communicating with the server and handling events, such as user login, account creation, and sending messages.

By following the MVC pattern, this chat application benefits from improved maintainability and modularity. The separation of concerns allows for easier updates to the individual components.

To store user information, we opted for a text-based database approach, which involves storing data in files. For this purpose, we specifically chose to use pickle files, which is a file format that allows for easy reading and writing of Python objects. We chose pickle files for data persistence for their simplicity and native support within Python.

We chose Python as the language of choice for our application due to prior familiarity with the language and we knew Flask and Tkinter would not be difficult to learn and there are a ton of great resources out there for those modules and the other modules we ended up using. We chose Flask because it is a lightweight Python Web Framework that allows developers to quickly build web applications. For our application, Flask is used to set up the server-side logic, handle user authentication, and manage the server endpoints. In addition to Flask, we also used Flask-SocketIO, which is an extension for Flask that provides support for Socket.IO.

We also used Tkinter which is a built-in Python library for creating graphical user interfaces. In our chat application, Tkinter is used to build the client-side UI components, such as the login screen, chat window, and user list. Tkinter's simplicity, native support, and cross-platform compatibility make it an excellent choice for building the UI of a small-scale application like this chat client.

Socket.IO is a real-time communication library that enables bi-directional communication between a server and clients. In the chat application, the Socket.IO library is used on the client-side to establish a real-time connection with the server, allowing the exchange of messages and user updates. Additionally, we used Requests, which is a popular Python library for making HTTP requests. In our chat application, it is used on the client-side to communicate with the Flask server for tasks like user authentication, account creation, and user management.

Server side we use HTTP requests and some Socket.IO. Our HTTP is for user registration and authentication; it also creates sessions. While Socket.IO does the heavy lifting, such as full duplex, low overhead communication between client/server Combination of WebSockets and HTTP long-polling. Socket.IO Allows for real time updates and an easy API to send messages to a group of users. Socket.IO is built on top of Engine.IO protocol and adds multiplexing via “namespaces”.