Yapp

-Design & Protocol-

- **Concurrency Strategy**

The concurrency strategy outlined for the chat application involves a combination of multiple approaches designed to manage high levels of concurrent operations, ensure data integrity, maintain system responsiveness, and optimize performance in real-time scenarios. The primary strategy revolves around an event-driven architecture that leverages the asynchronous nature of Node.js.

By using the built-in EventEmitter class, the application can handle various chat events, such as new messages or user actions, in a non-blocking manner. This event-driven model is ideal for managing concurrency as it allows multiple parts of the system to operate independently, responding to events as they occur. This setup ensures that multiple users can send and receive messages concurrently without causing delays or blocking other operations, which is crucial for a seamless user experience in a chat application. In other words, an **Event-Driven Architecture** was used in this communication application.

Another critical aspect of the concurrency strategy is the use of **Asynchronous Operations** to handle I/O tasks. By utilizing async/await or Promises, the application ensures that operations such as sending messages, connecting to a database, or querying user information do not block the main execution thread. This non-blocking I/O model allows the server to remain responsive, even under high load, as it can continue processing other tasks while waiting for I/O operations to complete. This approach is particularly important in real-time applications like chat systems, where responsiveness and low latency are essential for user satisfaction.

Concurrency Control is further enhanced through the use of WebSockets for real-time communication between clients and the server. WebSockets provide a persistent connection that allows for bi-directional communication, enabling multiple clients to interact with the server simultaneously without interfering with each other's messages. This setup facilitates high concurrency, as each WebSocket connection is managed independently, allowing for smooth handling of numerous simultaneous connections. To prevent abuse and maintain server stability, rate limiting is implemented, ensuring that no single user can overwhelm the server with excessive requests in a short period.

To support scalability and manage a large number of concurrent users, the strategy includes **Load Balancing** and horizontal scaling. By deploying multiple

instances of the chat server behind a load balancer, the system can distribute incoming connections and messages across various servers. This distribution helps prevent any single server from becoming a bottleneck, allowing the system to handle more users concurrently and improving overall performance. Additionally, session management is implemented using a shared session store, ensuring consistent user experiences across different server instances.

Error Handling plays a vital role in maintaining concurrency without degrading the user experience. Implementing graceful degradation allows the system to manage and recover from issues, such as connection drops or failed message deliveries, without affecting other users. A retry mechanism ensures that failed operations, like sending messages, can be retried, enhancing reliability and maintaining a smooth user experience despite occasional failures.

By using **Client-Side Considerations**, the strategy employs optimistic concurrency control and robust state management techniques to handle concurrent messages and updates. By updating the user interface optimistically, the client can provide immediate feedback to the user while managing potential conflicts gracefully. This approach ensures a responsive and fluid user experience, even in the face of network delays or server issues. Moreover, state management tools, such as Redux in a React application, help handle concurrent state changes and message queuing, ensuring that the display of chat messages remains accurate and up-to-date.

Finally, the concurrency strategy includes comprehensive **Testing and Monitoring** to ensure that the system can handle high concurrency scenarios effectively. Stress testing simulates high traffic and concurrent connections to identify performance bottlenecks and potential weaknesses. Monitoring tools are used to track the performance and health of the chat server in real-time, providing valuable insights into response times, error rates, and resource usage. This proactive approach helps maintain optimal performance and quickly address any issues that arise under load, ensuring the system remains robust and responsive at all times.

Overall, the concurrency strategy for the chat application combines event-driven design, asynchronous operations, real-time communication, load balancing, and robust error handling to create a highly scalable, responsive, and reliable system capable of handling a large number of concurrent users efficiently. This way, the YAPP application is designed to become an communication application worthy of being more than just functional, but also adaptable and flexible with accordance to the users personal preference.

- **UI Sketches**

Designs and aesthetics are just as important as the functionality of the application itself. With the functionality of the application secured and is well running, the UI that

YAPP will use is rather simple than complicated. There will be 3 main consoles that will make YAPP fully functional, so we'll be providing 3 UI sketches as well according to the main consoles.

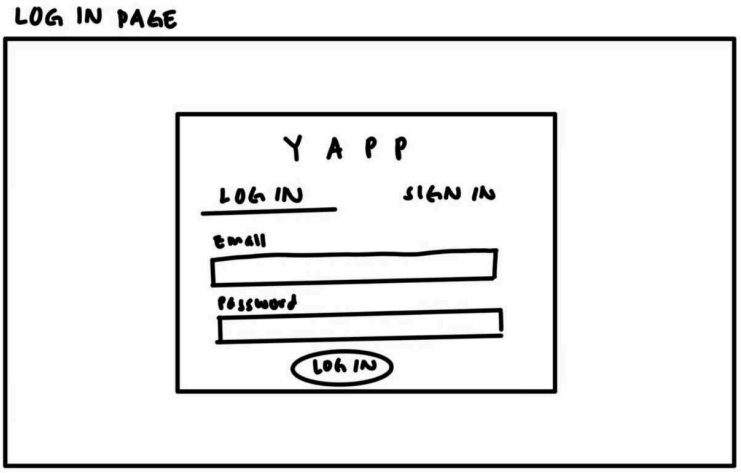

Image 1
Log In Page

The UI sketch for the LogIn Page and SignIn Page would be somewhat that of above. The LogIn Page provides a simple Email and Password bar, a login button, and an option to select either LogIn or SignIn into the Main Chat Console. The simplicity of the LogIn Page provides a good impression towards the users of YAPP as all of the things needed for the app to work and run are already provided all in a box, right in the middle of the app.

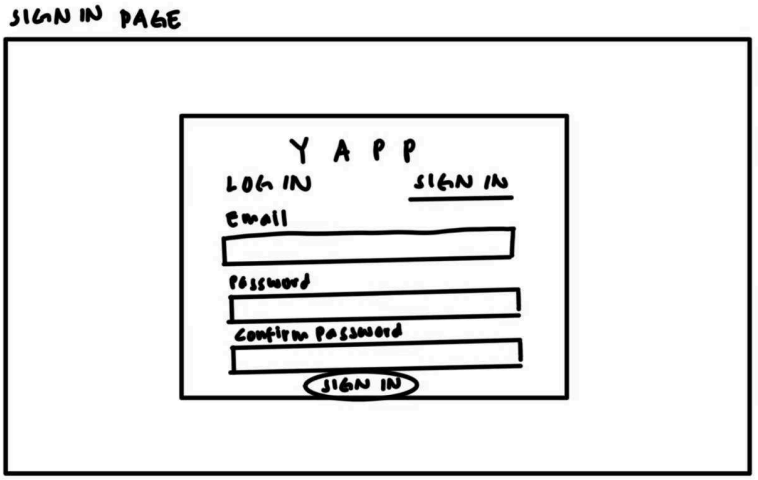

Image 2
Sign In Page

While the SignIn Page provides an Email bar, Password bar, and a Confirm Password bar, a sign in button and an option to switch between SignIn and LogIn which leads to the Main Chat Console. All of these features would be isolated into one box, thus creating a visual contrast in between the background and the functions

that the page provides. Similar to the LogIn Page, the simplicity of the SignIn Page provides a good impression towards the users of YAPP as all the functions provided by YAPP are all accessible right in the middle of the app.

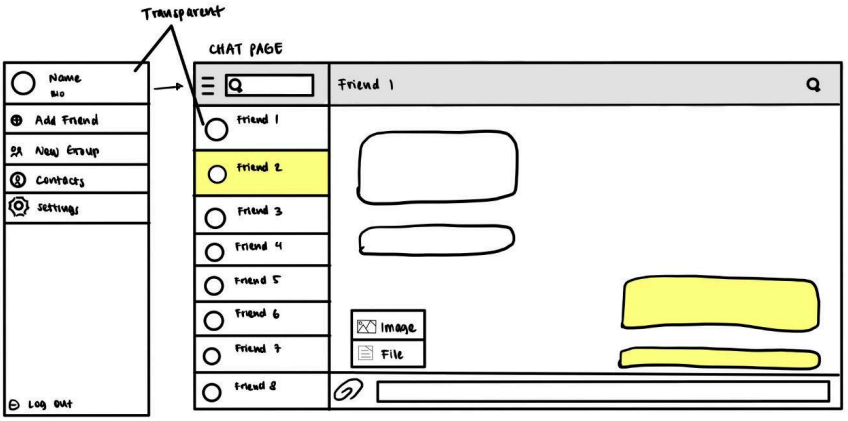

Image 3
Main Chat Console UI

The main chat console will be the UI that provides the most functionality, where the user can expand the more options tab, the friends list & its search bar, and the chat console itself, whether it being between one or more users where the users can send images or files. Here the UI of the YAPP application would be using the same concept as the LogIn and SignIn Page, which is a simple design.

The Chat Bubble will be a well rounded chat bubble, here the active users chat bubble would have a yellow color palette where the other users that's being communicated to will have a rather monotone color, a gray or white color. The contrast provided between the main chat console and the bar on the left side would be an important way to put the focus on the main chat console. The main chat console would have a solid background while the bar on the left side would have a transparent background, this way the users can see the contrast that both of these parts of the app have 2 different functions. With the solid background being the part mostly used as it is for communication and the transparent background being used for selectable options.

- **References**

Image 4
Reference Used for the UI provided

The YAPP application takes inspiration from a rather well known and used UI design, based from the messages application that is usually used by the MacOS devices or in other words Apple products. The design that Apple crates for their Messages application is rather easy to use and is simplistic, therefore giving the users ease of use too, the same thing that was envisioned for the YAPP application.

- **Testing Strategy**

A comprehensive testing strategy for a communication app that operates over the same network must address several critical aspects to ensure the app functions seamlessly, securely, and efficiently under various conditions. First, **functional testing** is crucial for verifying that all core features of the app, such as messaging, sending images, and sending files, operate as expected. This involves testing each feature individually and ensuring they work together cohesively. Functional testing also includes testing different user scenarios, such as initiating a call while sending a message or switching between audio and video modes, to ensure smooth transitions and functionality.

Performance testing is another essential component of the strategy. This type of testing evaluates the app's behavior under various network conditions, such as low bandwidth, high latency, and network congestion. The goal is to ensure the app maintains its responsiveness and quality of service, providing a smooth user experience even in challenging network environments. Performance testing should include stress testing to determine the app's breaking point and identify potential bottlenecks that could impact usability. Additionally, **load testing** simulates multiple users accessing the app simultaneously to evaluate its scalability and ability to handle a high number of concurrent connections without degradation in performance or user experience.

Given the sensitive nature of communication apps, **security testing** is imperative to protect user data and ensure privacy. This involves testing the app for vulnerabilities, such as data breaches or man-in-the-middle attacks, and verifying that all communication is encrypted end-to-end. Security testing should also assess the app's ability to withstand various cyber threats and ensure compliance with relevant data protection regulations. Furthermore, **interoperability testing** ensures that the app can function seamlessly across different devices, operating systems, and network configurations. This is particularly important in a diverse user environment where the app must deliver consistent performance and functionality regardless of the platform or device being used.

Lastly, an effective testing strategy should include **continuous monitoring and updates** to adapt to evolving network environments and user needs. This means regularly updating the app to address newly discovered vulnerabilities, improve performance, and introduce new features that enhance user experience. Automated testing tools can be employed to streamline this process, enabling rapid identification and resolution of issues as they arise. By adopting a comprehensive and adaptive testing strategy, developers can ensure that their communication app provides a reliable, secure, and high-quality experience for all users, regardless of the network conditions.

Ultimately, to ensure the condition and well being of YAPP, there will be multiple layers of testing involved. The main way to achieve such things is by doing field testing. We activate the server on end while having 2-3 users active from multiple devices as clients of the server. From there, testing will be done. Messages, files, and images will be sent from one user to another (Private Messaging) or from one user to multiple users (Group Messaging). This way we'll get to see how the app performs under stress. As for the other selectable options, it can be tested isolatedly, and should be a rather simple task in comparison with the actual communication and texting function itself.