CS-6500 A6

Tavish Burnah

January 2019

Most of my experience is in back end development, so while I know what email is, and have a general idea of how it works, I fear my model will be simplistic to others because I don't have the web server experience outside of school that others do. That being said, the clear pattern to use initially is Client-Server. We will have a server sending and receiving messages, and clients accessing messages.

The Client side will be written in C++ using the Qt Framework. This means the User Interface will be written in QML. Developing in the Qt Framework allows for the application to be written once (including all of the User Interface) and be published on all major platforms. It is also gives high-performance because it is written at a low level on the application side (not interpreted), and the User Interface is on its own thread.

For the Client Architecture, the pattern that made sense to me was a Layers pattern. There will be a User Interface Layer, an Application Logic layer, and a Database Layer. I guess that does sound a lot like the MVC model as well. Within the application layer, I chose to use multiple modules with specialized tasks: a Communication Manager to handle communication to and from the server, a Cryptography library which contains the SSL, Encryption and Decryption support, a Background Service which is responsible for notifying users when the app is not currently in the foreground, and a Database Manager module to handle creating, destroying, updating and choosing the right database for each user. To tie them all together, there was a Core module. This could act like the Supreme Commander pattern for the Application Logic Layer.

The application should rely on ACID to ensure that messages are sent. The messages will be stored locally in the databases at all times, and will be marked as unsent until the send is committed. There will be one database per user, per device. The data in the database will be encrypted using the Cryptography module. This will ensure security within the client system. In addition, the client will have to log in with a user-name and password. The password is hashed and compared against a value stored in the cloud. If the hash is identical, it is downloaded to the local database so that the user can log in "offline".

On the server side, I chose to use a number of Azure services: one to host the application, one to host the DNS and one to host the Database. Using Microsoft Azure provides reliability, concurrency and high availability. Because of Azure, the server will be written in C#. This is the most compatible and simple language to use for Azure development.