A Prototype Peer to Peer Contact Sharing Mobile Application

A Paper Submitted to the Department of Engineering and Society

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia - Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

By

James Foster

Spring, 2022

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Signed:________ [illegible] ______________ Date 20 April 2022

Introduction

Exchanging digital information has become a chore, an arduous and tedious task. Too often are people using email as a means to transfer files from device to device when it was never designed with that purpose in mind. Modern day alternatives to this are AirDrop and similar technologies that use Bluetooth and WiFi local area networks to transfer data between devices. The problem is that these solutions are not cross-platform at all, i.e. it is impossible to AirDrop a file from an iPhone to an Android device. This Technical project aimed to answer this calling by providing a cross platform solution for exchanging digital information, starting with easily shareable contact information. This report will explain the development process of this prototype application and the impetus behind many of its design choices.

Part I: Client-server Architecture

The first epoch of development was based around building a client-server architecture. The mobile app would communicate with an API server that then communicated with a database that would house all of the user data needed in the app. The data would then be sent from the database, through the API server, and delivered to the mobile application. The mobile platform chosen for the prototype was Android and the code was written in Kotlin that translated to Java. Since data would be stored online in a database, users needed to make an account and log in to the service to gain access to that data. Multiple single-sign-on, or SSO, providers were tested from a self-hosted solution, to AWS Cognito, to Google Firebase, with the last being chosen due to its drop-in SDK for Android development. These providers store user login information securely and make OAuth flows much easier to implement for developers. At first, a Proof Key Code Exchange, or PKCE, server was written to work as a self hosted authorization medium, however this was tedious to create and was not worth the time to flesh out. Therefore, Firebase was chosen and preliminary login tests worked well with the prototype application. Firebase also allowed for Anonymous user accounts which could allow users to test drive the application before committing and creating an actual account. Since login implementation was covered and users were able to create accounts and sign in to the application it was now time to develop the actual back-end of the service.

The database was the first point of business. Tables were originally designed to store all user information. There was a private schema for the user login information and a public schema for user transaction information and basic info. The private schema only had a "salts" table that store users' hashed password salt. The public schema had tables titled "users", "swap_conns", "bump_conns", and "webs" representing user data, swap transactions between users, bump transactions between users, and groups of users respectively. These were SQL database tables that were stored in a PostgreSQL database server. This database server choice was due to PostgreSQL having a very good reputation among SQL databases. The only entity allowed to log in to this server was the API server. For role-based access control, or RBAC, only individuals with the "admin" role could view the "salts" table and other private user information. A "generic" role was created to allow the API server to insert, update, and delete items from the public schema tables. Therefore, any malicious actor trying to use the API server to extract data from the database could only see the public, and less sensitive, data tables.

The API server is just a medium to translate user requests from the mobile application into database queries. It was written using the Rust programming language and the Actix web framework crate. Rust was chosen as the developing language due to its high execution speed and modern syntax. The Actix web framework was chosen because it is a highly rated Rust web framework that would work well as a representational state transfer, or REST, server. To handle logins without using an SSO provider, as mentioned above, a PKCE server was made also using Actix web framework that took login credentials and returned an authorization token and refresh token to the user if their credentials were valid. This server was scrapped after migrating to SSO providers starting with AWS Cognito and then finally Google Firebase. After programming, the REST server worked as intended, but setting up

"URL-esque" endpoints is not easy when you are developing an API server rather than a website. Therefore it was decided to use the Google Remote Procedure Call, or gRPC, protocol for data transfer instead of REST. This protocol abstracted the API endpoints away from URLs to concise JSON calls. This made it a lot more intuitive to design new API calls and determine what data would be expected at each endpoint. Actix web doesn't support gRPC so the API server instead used a Rust crate called "tonic". After the gRPC API server was completed the Android app successfully queried for data using it and the complete concept of the mobile app service was slowly coming together.

The mobile application prototype was only developed for Android due to time concerns, but future releases will be developed for both Android and iOS. Android development has been evolving rapidly and many different approaches were attempted when designing the user interface, or UI, of this application. The first approach was using XML source files to define the UI which are then referenced in the Kotlin code. This is the oldest method for creating Android user interfaces and it is cumbersome and tedious, but makes up for this with its reliability. A new up and coming library called Jetpack Compose involves designing user interfaces using functions called Composables. These can be nested and can actually contain all the application logic if so desired. An iteration of this prototype app was created that was purely Composables, but the logic behind the UI became difficult to handle. The final decision was to go with a hybrid UI that incorporated the logic used in the XML approach and the Composables. It was possible to place Composables in a specific area of the normal code that allowed them to comprise the UI rather than XML files. This hybrid approach also introduced some complex logic, but since the application was not completely made of Composables, this logic was much easier to handle.

When the prototype application using this client-server model was finally complete, ethical concerns were risen. It did not seem to make sense to force users to trust a remote server to store there information indefinitely, so the "users" table was removed from the database. The database would now only store the transactions between users, and no personal information regarding them. The necessity of a database seemed to be waning and alternative architectures were being looked into. Eventually, it was deemed worthwhile to attempt a peer to peer solution of the application.

Part II: Peer to Peer Architecture

Mobile peer to peer applications and academic papers were in the mainstream a decade ago. For some reason interest in this field seemed to die down rapidly as more and more businesses defaulted to the client-server model which was tried and tested. The client-server model allows businesses to acquire petabytes of information on their users to sell, a highly valuable aspect of them. Peer to peer models decentralize an application and allow users, or peers, to directly send information from their device to another peer's. This provides enhanced security and execution speed, but created much more difficulty for long distance communication and big data collection. Peer to peer was chosen to be attempted because of its novelty in the mobile space and because mobile devices are relatively powerful and have the ability to send the data using multiple robust protocols. Peer to Peer also allows users to keep all of their personal information they would like to share on their device and not on any remote server this service is hosting. This architecture is volatile and difficult to implement correctly, so it has only really been used for niche cases in the mobile space.

The AirDrop service on iOS devices is one example of an application that uses a Peer to Peer architecture. It first detects other devices in the proximity of the user's device using Bluetooth Low Energy, or BLE, and lists them for the user to select. The user clicks on a name and then a WiFi LAN connection is spun up so that they can transfer their files using the higher bandwidth of WiFi rather than BLE. The first iteration of this project's Peer to Peer version was a pure BLE concept, where BLE proximity would be used to find nearby devices using the app as well as transfer the data between the devices. A low level BLE Android library was used to implement the BLE functionality and was very time consuming to work. A Bluetooth General Attribute Server Structure, or GATT, had to be designed and implemented, specifying what data would be transferred using this GATT server profile. Indeed this is a server and this is a problem with trying to develop a Bluetooth only Peer to Peer app. The Bluetooth GATT servers are brain dead and do not even recognize they has been connected to. It can only respond to requests from clients that have forcibly connected to them or send notifications. This made the data transfer logic much more complex than it needed to be and other solutions were looked into. The next solution was using the AirDrop formula where BLE detects nearby devices and WiFi LAN pipes the data between them. Initially, these protocols were going to be designed and implemented in-house so that they worked specifically with this app, but since this app was designed to be a prototype for this technical project SDKs implementing this functionality were used instead. The SDK of choice was one of a small start-up named Ditto. Like AirDrop, Ditto used BLE and WiFi LAN but it had a strange quirk of being designed as a database that can sync all of its data with peers. For instance, if two users both have a table named "cars" that one updates by inserting a new car, and the other user has subscribed to watch changes to that table, then the latter user's "cars" table will also insert the new car when it sees that the other user's "cars" table has. This is great functionality for niche application types but not for this prototype app. This quirk had to be taken into consideration when developing the transactions functions for the app. Basically, the sync functionality had to be translated to distinct transactions. To do this a helper class was made with logically titled function names that pretended to be transactions but had the Ditto sync logic under the hood. This helper class worked well for a time until progress on it halted and transaction logic was sloppily dropped into the main source files which made it very hard to tell what was going on. This is the current state of the prototype, where it is usable and functions well but the underlying code is positively unreadable for someone unfamiliar with the Ditto SDK. As mentioned earlier, an in-house solution is greatly preferred and will most likely be implemented if this project ever takes off and is well funded.

Conclusion

The Technical project resulted in a prototype Android mobile application that was able to transfer data peer to peer using Bluetooth Low Energy and WiFi LAN. The app is a basic contact sharing app for now, where users "bump" when they meet to add each other to their address books and then users can "swap" data thereafter. The data they swap for now is limited to basic contact information, but an AirDrop alternative that can transfer any data or file will be implemented later. The chief goal of this project was to create a mobile app that could send significant amounts of data locally between devices, without using the internet at all. This prototype has proven that the concept works and just needs to be fleshed out. The development of this project will not stop after the end of this semester and the release of a minimum viable product is on the horizon for this summer. The prospective impacts hoped for this application are for it to encourage people to make and reinforce physical relationships, to remind people that their personal data is their own property, to bring mobile Peer to Peer back into the zeitgeist, and to make data and file transfer between any devices extremely easy.